

بسمه تعالیٰ



دانشگاه تهران

دانشکده مهندسی مکانیک

تمرین سری چهارم درس هوش مصنوعی

نام و نام خانوادگی:

محمدمهری تویسرکانی

شماره دانشجویی:

۸۱۰۶۰۳۰۰۵

تاریخ تحویل:

۱۴۰۴/۰۳/۱۵

نیمسال دوم سال تحصیلی ۱۴۰۴-۱۴۰۳

## فهرست مطالب

صفحه	عنوان
۱	اطلاعات مسئله
۵	حل مسئله
۵	۱) بخش اول: دادگان UCI HAR
۵	معرفی کتابخانه‌های استفاده شده
۶	۱-۱) آماده‌سازی داده‌ها
۶	۱-۱-۱) معرفی روش‌های مختلف نرمال‌سازی و بیان مزایا و معایب هر کدام
۷	۱-۱-۲) انتخاب یکی از روش‌های نرمال‌سازی و پیاده‌سازی آن
۸	۱-۱-۳) اختصاص ۰.۸۵٪ و ۰.۱۵٪ داده‌های نرمال شده به آموزش و آزمون و ذکر ابعاد هر بخش
۸	۲-۱) طراحی شبکه MLP
۱۰	۳-۱) طراحی شبکه CNN
۱۱	۴-۱) آموزش
۱۳	۵-۱) ارزیابی
۱۳	۱-۵-۱) رسم نمودارهای خطای Loss و دقت Accuracy برای داده‌های آموزش و آزمون و تحلیل هر کدام
۱۵	۱-۵-۲) دقت نهایی مدل روی داده‌های آزمون
۱۷	۱-۵-۳) رسم ماتریس آشفتگی برای داده‌های آزمون و تفسیر آن
۲۰	۶-۱) تحلیل
۲۱	۲) بخش دوم: دادگان NEU Surface Defects
۲۱	معرفی کتابخانه‌های استفاده شده

۲۲	۱-۲) آماده‌سازی داده‌ها
۲۲	۲-۱-۱) انتخاب یکی از روش‌های نرمال‌سازی و پیاده‌سازی روی داده‌ها
۲۳	۲-۱-۲) اختصاص ۸۵٪ و ۱۵٪ داده‌های نرمال شده به آموزش و آزمون و ذکر ابعاد هر بخش
۲۴	۲-۲) طراحی شبکه MLP
۲۶	۳-۲) طراحی شبکه CNN
۲۹	۴-۲) آموزش
۳۰	۵-۲) ارزیابی
۳۰	۱-۵-۱) رسم نمودارهای خطای Loss و دقت Accuracy برای داده‌های آموزش و آزمون و تحلیل هر کدام
۳۰	
۳۳	۲-۵-۲) دقت نهایی مدل روی داده‌های آزمون
۳۴	۲-۵-۳) رسم ماتریس آشفتگی برای داده‌های آزمون و تفسیر آن
۳۷	۲-۶-۲) تغییر های پر پارامترها
۳۷	۱-۶-۲) استفاده از لایه Dropout در شبکه CNN به جای Block Dropout
۴۴	۲-۶-۲) توضیح و پیاده‌سازی تجزیه فیلترها Kernel Factorization در شبکه CNN
۵۱	۷-۲) تحلیل
۵۳	۳) بخش سوم: یادگیری انتقالی Transfer Learning
۵۳	۳) معرفی کتابخانه‌های استفاده شده
۵۳	۱-۳) آماده‌سازی داده‌ها
۵۵	۲-۳) آماده‌سازی مدل
۶۴	۳-۳) مرحله‌ی آموزش
۶۷	۴-۳) ارزیابی نهایی

۱-۴-۳) رسم نمودارهای خطای داده‌های آموزش و آزمون و تحلیل هر کدام	۶۷
۲-۴-۳) دقت نهایی مدل روی داده‌های آزمون	۷۰
۳-۴-۳) رسم ماتریس آشفتگی برای داده‌های آزمون و تفسیر آن	۷۰
۴-۳) تحلیل	۷۲
گیت هاب	۷۴

## هوش مصنوعی و یادگیری ماشین (شبکه‌های عصبی)

### ❖ اطلاعات مسئله:

هدف این تمرین آشنایی با ساختار شبکه پرسپترون چندلایه (MLP) و شبکه پیچشی عمیق (CNN) و مقایسه عملکرد آن‌ها در کاربردهای طبقه‌بندی چندکلاسه است. برای این کار از دو مجموعه داده، یکی با ساختار برداری و دیگری در قالب تصویر استفاده می‌شود. این دو مجموعه داده در زیر معرفی می‌شوند.

#### ۱. دادگان تشخیص فعالیت‌های انسانی با گوشی (UCI HAR)

این دادگان که شامل ۱۰۲۹۹ داده برداری است حاصل پژوهشی است که می‌کوشد نوع فعالیت فیزیکی افراد (شامل شش فعالیت را در فتن، بالا رفتن از پله، پایین آمدن از پله، نشستن، ایستادن و دراز کشیدن) را براساس اطلاعات ضبط شده توسط گوشی هوشمند آن‌ها تشخیص دهد. اطلاعات ضبط شده پس از پردازش‌های اولیه به بردارهای  $561 \times n$  درایه‌ای تبدیل می‌شوند و به این ترتیب ورودی مدل مورد نظر یک ماتریس  $(n \times 561)$  خواهد بود که در آن  $n$  تعداد نمونه‌هاست. برچسب هر نمونه نیز نوع فعالیت فرد (یکی از شش فعالیت یاد شده) خواهد بود.

#### ۲. دادگان تشخیص عیوب سطحی قطعات فولادی (NEU Surface Defects)

این دادگان شامل ۱۸۰۰ تصویر خاکستری ( $200 \times 200$ ) پیکسلی از قطعات فولادی نورد شده است که هر یک دارای یکی از عیوب ترک‌های سطحی<sup>۱</sup>، ناخالصی<sup>۲</sup>، لکه<sup>۳</sup>، قلوه کن شدگی<sup>۴</sup>، پوسته‌گرفتگی<sup>۵</sup> و خراشیدگی<sup>۶</sup> است. دادگان مورد نظر کاملاً متوازن است، به این معنی که به ازای هر عیوب، ۳۰۰ نمونه تصویری وجود دارد.

به دلیل وجود نویزهای گوناگون در تصاویر و پیچیدگی بافت سطحی قطعات انتظار می‌رود که عملکرد شبکه‌های پیچشی در دسته‌بندی عیوب قطعات بهتر از عملکرد مدل‌های ساده‌ای همچون MLP باشد.

### بخش اول: دادگان UCI HAR

#### ۱. آماده‌سازی داده‌ها

- روش‌های مختلف نرم‌افزاری را معرفی کرده و مزایا و معایب هر یک را بیان کنید.
- یکی از این روش‌ها را انتخاب و پیاده‌سازی نمایید.
- ۸۵٪ داده‌های نرمال شده را به آموزش و ۱۵٪ آن‌ها را به آزمون اختصاص داده و ابعاد هر بخش را گزارش کنید.

#### ۲. طراحی شبکه MLP

<sup>1</sup> Crazing

<sup>2</sup> Inclusion

<sup>3</sup> Patches

<sup>4</sup> Pitted Surface

<sup>5</sup> Rolled-in Scale

<sup>6</sup> Scratches

یک شبکه MLP با ساختار دلخواه (حداکثر دو لایه پنهان با یا بدون لایه Dropout) طراحی کرده و توابع فعالسازی مورد استفاده برای لایه‌های پنهان و لایه خروجی را با ذکر دلیل انتخاب نمایید.

### ۳. طراحی شبکه CNN

یک شبکه پیچشی با معماری دلخواه (تعداد لایه‌های کانولوشن، لایه‌های Dropout، Pooling و توابع فعالسازی) به همراه نرمالسازی دسته‌ای (Batch Normalization) طراحی کرده و آموزش دهید. دلایل انتخاب این معماری را به صورت کامل بیان نمایید.

### ۴. آموزش

شبکه‌های طراحی شده را با استفاده از الگوریتم Adam و نرخ یادگیری  $0.001$  آموزش دهید.

### ۵. ارزیابی

- نمودارهای خطا (Loss) و دقت (Accuracy) را برای داده‌های آموزش و آزمون رسم و تحلیل کنید.
- دقت نهایی مدل را روی داده‌های آزمون گزارش کنید.
- ماتریس آشتقگی (Confusion Matrix) را برای داده‌های آزمون رسم و تفسیر کنید.

### ۶. تحلیل

با مقایسه نتایج مشخص کنید که کدامیک از دو مدل (CNN یا MLP) عملکرد بهتری در این مسئله دارد و علت آن را توضیح دهید.

## بخش دوم: دادگان NEU Surface Defects

### ۱. آماده‌سازی داده‌ها

- یکی از روش‌های نرمالسازی را انتخاب و روی داده‌های این بخش پیاده‌سازی نمایید.
- $85\%$  داده‌های نرمال شده را به آموزش و  $15\%$  آن‌ها را به آزمون اختصاص داده و ابعاد هر بخش را گزارش کنید.

### ۲. طراحی شبکه MLP

یک شبکه MLP با ساختار دلخواه (حداکثر دو لایه پنهان با یا بدون لایه Dropout) طراحی کرده و توابع فعالسازی مورد استفاده برای لایه‌های پنهان و لایه خروجی را با ذکر دلیل انتخاب نمایید.

### ۳. طراحی شبکه CNN

یک شبکه پیچشی با معماری دلخواه (تعداد لایه‌های کانولوشن، لایه‌های Dropout و تابع فعال‌سازی) به همراه نرمال‌سازی دسته‌ای (Batch Normalization) طراحی کرده و آموزش دهید. دلیل انتخاب این معماری را به صورت کامل بیان نمایید.

#### ۴. آموزش

شبکه‌های طراحی شده را با استفاده از الگوریتم Adam و نرخ یادگیری ۱٪ آموزش دهید.

#### ۵. ارزیابی

- نمودارهای خطا (Loss) و دقت (Accuracy) را برای داده‌های آموزش و آزمون رسم و تحلیل کنید.
- دقت نهایی مدل را روی داده‌های آزمون گزارش کنید.
- ماتریس آشتقگی (Confusion Matrix) را برای داده‌های آزمون رسم و تفسیر کنید.

#### ۶. تغییر هایپرپارامترها

- در معماری CNN، به جای استفاده از لایه Dropout از Block Dropout استفاده کنید. دلیل این جایگزینی را توضیح داده و پس از آموزش مجدد، نتایج را با نتایج حالت قبل مقایسه نمایید.
- مفهوم تجزیه فیلترها (Kernel Factorization) را توضیح داده و آن را در شبکه CNN پیاده‌سازی کنید. پس از آموزش مجدد، نتایج به دست آمده را با حالت قبل مقایسه و تحلیل نمایید.

#### ۷. تحلیل

مشخص کدام مدل در این مسئله عملکرد بهتری داشته است و علت برتری آن را توضیح دهید.

#### بخش سوم: یادگیری انتقالی (Transfer Learning)

استفاده از شبکه‌ای که قبلاً روی داده‌های مشابهی آموزش دیده است یکی از روش‌های متداول برای افزایش دقت و تعمیم‌پذیری شبکه است. در این روش که پیش‌تریت (pre-training) نامیده می‌شود ابتدا شبکه روی دادگان مشابهی آموزش داده می‌شود و سپس ضمن ثابت نگهداشتن ضرایب وزنی بقیه شبکه، یک یا چند لایه نهایی آن با لایه‌های (های) جدیدی جایگزین شده و تنها ضرایب وزنی این لایه‌های (های) جدید تعیین می‌شود. سپس در ادامه فرآیند آموزش به تدریج به لایه‌های قبلی نیز اجازه داده می‌شود که ضرایب وزنی خود را بروزآوری کنند.

در این بخش می‌خواهیم به کمک یادگیری انتقالی، عملکرد مدل تشخیص عیوب سطحی را بهبود ببخشیم. برای این کار از یک مدل از پیش‌آموزش دیده (ResNet-50) استفاده خواهیم کرد.

#### ۱. آماده‌سازی داده‌ها

با استفاده از روش‌های استاندارد، دادگان NEU Surface Defects را فراخوانی کرده و آن را برای ورود به شبکه ResNet-50 آماده نمایید. بهمنظور بهبود فرآیند آموزش، افزایش تعمیم‌پذیری مدل و ارتقاء مقاومت به نویز، از روش‌های مختلف داده‌افزایی (Data Augmentation) استفاده کنید. در پایان نیز بهصورت تصادفی از هر کلاس یک نمونه را به همراه برچسب (Label) مربوطه نمایش دهید.

## ۲. آماده‌سازی مدل

شبکه ResNet-50 را با وزن‌های آموزش‌دیده روی مجموعه‌داده ImageNet بارگذاری کرده و لایه نهایی آن را حذف نمایید. سپس دو لایه جدید شامل لایه Dense Softmax و لایه خروجی برای طبقه‌بندی شش نوع عیوب سطحی به شبکه اضافه کنید.

## ۳. مرحله‌ی آموزش

در ابتدا، بهمنظور بهره‌گیری از یادگیری انتقالی در مراحل ابتدایی، ضرایب وزنی همه لایه‌های شبکه پایه ثابت نگه داشته شده (از نظر یادگیری غیرفعال شده) و تنها بخش جدید (head) شبکه آموزش داده می‌شود. سپس به تدریج و از انتهای شبکه، برخی لایه‌ها از حالت ثابت نگه داشته شده (فریز) خارج شده و مدل بهصورت مرحله‌ای تحت فرآیند Fine-Tuning قرار می‌گیرد.

## ۴. ارزیابی نهایی

پس از تکمیل آموزش، عملکرد مدل را روی مجموعه آزمون ارزیابی کرده و موارد زیر را گزارش دهید:

- دقت نهایی (Accuracy)
- تابع خطا (Loss)
- ماتریس آشتفتگی (Confusion Matrix)

## ۵. مقایسه و تحلیل

مدل آموزش‌دیده با یادگیری انتقالی را با بهترین مدل CNN در بخش دوم مقایسه کنید. در تحلیل خود به موارد زیر اشاره نمایید:

- مقایسه دقت نهایی و زمان آموزش
- مقایسه اندازه و پیچیدگی مدل
- مزایا و معایب استفاده از یادگیری انتقالی در مقابل آموزش مدل از ابتدا

## ❖ حل مسئله

## ۱) بخش اول: دادگان UCI HAR

## ➤ معرفی کتابخانه‌های استفاده شده

```

import numpy as np
import pandas as pd
import os
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt
import seaborn as sns
import torch
import torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader

```

- کتابخانه NumPy برای محاسبات عددی استفاده می‌شود. np مخفف آن است و برای کار با آرایه‌ها، جبر خطی، آمار و توابع ریاضی بسیار کاربرد دارد.
- کتابخانه Pandas برای کار با داده‌های جدولی (DataFrame) به کار می‌رود. pd مخفف آن است و برای خواندن فایل‌های CSV، پاکسازی داده‌ها و تحلیل آماری استفاده می‌شود.
- کتابخانه os برای کار با سیستم‌عامل است، مانند: دسترسی به مسیر فایل‌ها، ساخت پوشه، خواندن محتویات دایرکتوری‌ها و مدیریت مسیرها.
- از scikit-learn برای نرمال‌سازی ویژگی‌ها استفاده می‌شود. StandardScaler داده‌ها را با میانگین صفر و انحراف معیار یک مقیاس‌بندی می‌کند.
- از train\_test\_split برای تقسیم داده‌ها به مجموعه آموزش و آزمون استفاده می‌شود. این تابع داده‌ها و برچسب‌ها را به طور تصادفی به نسبت دلخواه جدا می‌کند.
- از توابع ConfusionMatrixDisplay و confusion\_matrix و accuracy\_score برای محاسبه دقت مدل، ایجاد ماتریس آشفتگی و نمایش ماتریس آشفتگی استفاده می‌شود.
- Matplotlib برای ترسیم نمودارها به کار می‌رود. plt مخفف آن است که ابزارهای گرافیکی ساده را ارائه می‌دهد.

- Seaborn کتابخانه‌ای برای رسم نمودارهای آماری پیشرفته و زیباتر بر پایه Matplotlib heatmap است.
- Torch کتابخانه‌ای برای یادگیری ماشین و یادگیری عمیق است. torch پایه‌ی اصلی آن بوده و شامل عملیات روی Tensorها، تابع ریاضی و نرمال‌سازی گرادیان است.
- ماژول nn در PyTorch برای ساخت شبکه‌های عصبی استفاده می‌شود. شامل لایه‌ها، تابع فعال‌سازی و سایر اجزای مدل است.
- DataLoader و TensorDataset برای مدیریت داده‌ها در Torch کاربرد دارند. به این صورت که مجموعه‌ای ازTensorها با ساختار نمونه و برچسب و DataLoader امکان بارگذاری داده‌ها بهصورت تصادفی و تکرارشونده را ارائه می‌دهند.

## ۱-۱) آماده‌سازی داده‌ها

### ۱-۱-۱) معرفی روش‌های مختلف نرمال‌سازی و بیان مزایا و معایب هر کدام

» **Min-Max Scaling**: روابط را حفظ می‌کند و تمام داده‌ها را در یک محدوده خاص [۰، ۱] نگه می‌دارد و رابطه

آن بهصورت زیر است:

$$X' = \frac{X - X_{\min}}{X_{\max} - X_{\min}}$$

» **Z-score Standardization**: داده‌ها را طوری تبدیل می‌کند که میانگین آنها برابر با صفر و انحراف معیار آنها

یک باشد و برای اعمال آن از رابطه زیر استفاده می‌شود:

$$X' = \frac{X - \mu}{\sigma}$$

که  $\mu$  و  $\sigma$  بهترتیب میانگین و انحراف معیار داده‌ها هستند.

» **Robust Scaling**: از میانه (median) و IQR (فاصله بین چارکی) استفاده می‌کند و رابطه آن بهصورت زیر

است:

$$X' = \frac{X - \text{median}}{\text{IQR}}$$

که  $\text{IQR} = Q_3 - Q_1$  است.

به منظور مقایسه بهتر این سه روش، مزایا و معایب هر کدام در جدول زیر آورده شده است:

نقاط ضعف	مزیت	روش
به مقادیر پرت (Outliers) بسیار حساس است. هم‌چنین، اگر داده‌ی جدید خارج از بازه‌ی قبلی باشد، ممکن است نتیجه اشتباهی ایجاد شود.	برای الگوریتم‌هایی که فرض می‌کنند داده‌ها در بازه‌ای خاص قرار دارند (مثل شبکه‌های عصبی و KNN) بسیار مناسب است.	<b>Min-Max Scaling</b>
به مقادیر پرت تا حدودی حساس است. فرض می‌کند توزیع داده‌ها تقریباً نرمال است و توزیع گاووسی را در نظر می‌گیرد.	در بسیاری از الگوریتم‌ها (مانند رگرسیون خطی، SVM و PCA) عملکرد خوبی دارد. هم‌چنین، بهتر از Min-Max در مواجهه با مقادیر پرت عمل می‌کند. داده‌های پرت را بهتر مدیریت می‌کند و داده‌ها را در مرکز قرار می‌دهد.	<b>Z-score Standardization</b>
در برخی الگوریتم‌هایی که فرض توزیع نرمال دارند، ممکن است عملکرد ضعیفتری نسبت به Z-score داشته باشد.	مناسب برای داده‌هایی که دارای توزیع غیرنرمال یا دارای outlier هستند، زیرا تحت تأثیر داده‌های پرت قرار نمی‌گیرد.	<b>Robust Scaling</b>

### ۱-۲-۲) انتخاب یکی از روش‌های نرمال‌سازی و پیاده‌سازی آن

با توجه به اینکه امکان وجود داده‌های پرت (Outliers) در این تمرین زیاد است، از روش Z-score استفاده می‌شود. زیرا همانطور که در جدول بالا بیان شد، این روش، داده‌های پرت را بهتر مدیریت می‌کند و در مواجهه با مقادیر پرت، عملکرد خوبی دارد. قبل از پیاده‌سازی این روش، ابتدا باید داده‌های موجود در پوشه‌های train و test فراخوانی شود که برای انجام این کار از کد زیر استفاده شده است:

```

def load_dataset(folder):
    X = pd.read_csv(f'{folder}/X_{folder}.txt', sep=r'\s+', header=None)
    y = pd.read_csv(f'{folder}/y_{folder}.txt', sep=r'\s+', header=None)
    subject = pd.read_csv(f'{folder}/subject_{folder}.txt', sep=r'\s+', header=None)
    return X.values, y.values.ravel(), subject.values.ravel()

X_train, y_train, _ = load_dataset('train')
X_test, y_test, _ = load_dataset('test')

```

سپس، داده‌های آموزش و آزمون مربوط به  $x$  و  $y$  ادغام شده و سپس با میانگین=صفر و انحراف معیار=۱ نرمال

می‌شوند. در ادامه، کد پیاده‌سازی این روش آورده شده است:

```

X_all = np.vstack((X_train, X_test))
y_all = np.hstack((y_train, y_test))

scaler = StandardScaler()
X_all_norm = scaler.fit_transform(X_all)

```

### ۱-۱-۳) اختصاص ۸۵٪ و ۱۵٪ داده‌های نرمال شده به آموزش و آزمون و ذکر ابعاد هر بخش

برای انجام این بخش، ۸۵٪ داده‌های نرمال شده به آموزش و ۱۵٪ به آزمون با random-state=42 اختصاص

داده می‌شوند و در آخر ابعاد هر بخش چاپ می‌شود.

```

X_train_norm, X_test_norm, y_train_norm, y_test_norm = train_test_split(
    X_all_norm, y_all, test_size=0.15, random_state=42, stratify=y_all)

print("Train shape:", X_train_norm.shape)
print("Test shape:", X_test_norm.shape)

```

ابعاد بخش آموزش و آزمون به صورت زیر به دست می‌آید:

```

Train shape: (8754, 561)
Test shape: (1545, 561)

```

## ۲-۱) طراحی شبکه MLP

برای طراحی شبکه MLP از دو لایه‌ی پنهان با لایه Dropout با تعداد نورون ۶۴ تا ۱۲۸ و تابع فعال‌سازی

که مخفف Rectified Linear Unit ReLU است، استفاده می‌شود که دلایل استفاده از آنها در جدول زیر ارائه

شده است:

دليل	وبژگى
مدل به اندازه کافی پیچیده است که الگوها را یاد بگیرد، ولی نه آنقدر پیچیده که overfit شود. به همین دلیل استفاده از دو لایه پنهان مناسب است.	۲ لایه پنهان
بهمنظور کاهش تدریجی برای استخراج ویژگی‌ها و کاهش پارامترها، این تعداد نوروں مناسب است.	تعداد نوروں (۶۴ - ۱۲۸)
برای جلوگیری از overfitting با حفظ تعییم‌پذیری از لایه Dropout نیز استفاده می‌شود.	Dropout
بدلیل اینکه تابع فعال‌سازی ReLU ساده، مؤثر، سریع و مناسب برای لایه‌های پنهان است.	ReLU

بنابراین، برای طراحی شبکه MLP از کد زیر استفاده شده است:

```
class MLP(nn.Module):
    def __init__(self, input_dim=561, num_classes=6):
        super(MLP, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(input_dim, 128),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, num_classes)
        )

    def forward(self, x):
        return self.model(x)
```

همانطور که در کد فوق مشخص است، برای طراحی شبکه MLP از ۵۶۱ ورودی به همراه خروجی ۶ کلاسه، یک لایه خطی با ورودی به ابعاد ۱۲۸، تابع فعال‌سازی ReLU، لایه Dropout با نرخ ۰.۳ (برخی از نوروں را در حین آموزش غیرفعال می‌کند تا از overfitting جلوگیری کند)، لایه پنهان اول با ۶۴ تا ۱۲۸ نوروں، تابع فعال‌سازی ReLU برای لایه پنهان قبلی و لایه پنهان دوم که تعداد خروجی‌ها را برابر با تعداد کلاس‌ها (num\_classes) تنظیم می‌کند، استفاده شده است.

### ۱-۳) طراحی شبکه CNN

برای طراحی شبکه CNN از لایه‌های کانولوشن یک بعدی (Conv1D)، نرمال‌سازی دسته‌ای (BatchNorm)، Fully Connected، Dropout، Pooling استفاده می‌شود که دلایل استفاده از آنها در جدول زیر ارائه شده است:

دلیل	ویژگی
پردازش مؤثر سیگنال‌های سری‌زمانی	<b>Conv1D</b>
نرمال‌سازی و تسريع یادگیری	<b>BatchNorm</b>
کاهش حساسیت به نویز و افزایش بازده محاسباتی	<b>MaxPooling</b>
جلوگیری از overfitting	<b>Dropout</b>
استخراج ویژگی‌های ساده تا پیچیده	<b>۲ مرحله کانولوشن</b>
ترکیب ویژگی‌ها برای پیش‌بینی نهایی	<b>Fully Connected</b>

بنابراین، برای طراحی شبکه CNN از کد زیر استفاده شده است:

```
class CNN(nn.Module):
    def __init__(self, input_channels=1, num_classes=6):
        super(CNN, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv1d(input_channels, 32, kernel_size=3, padding=1),
            nn.BatchNorm1d(32),
            nn.ReLU(),
            nn.MaxPool1d(2),
            nn.Conv1d(32, 64, kernel_size=3, padding=1),
            nn.BatchNorm1d(64),
            nn.ReLU(),
            nn.MaxPool1d(2)
        )
        self.fc = nn.Sequential(
            nn.Dropout(0.3),
            nn.Linear(64 * 140, 100),
            nn.ReLU(),
            nn.Linear(100, num_classes)
        )

    def forward(self, x):
        x = self.conv(x)
        x = x.view(x.size(0), -1)
        return self.fc(x)
```

همانطور که در کد فوق مشخص است، برای طراحی شبکه CNN از یک کanal ورودی به همراه خروجی ۶ کلاسه، دو لایه کانولوشن یک بعدی با ۳۲ و ۶۴ فیلتر، نرمال‌سازی دسته‌ای (BatchNorm) به منظور بهبود سرعت آموخت و پایداری. تابع فعال‌سازی ReLU، لایه Pooling یک بعدی با اندازه پنجره ۲ که ابعاد خروجی را کاهش می‌دهد، Fully Connected برای ترکیب ویژگی‌ها، لایه Dropout با نرخ ۰.۳. (برخی از نورون‌ها را در حین آموخت غیرفعال می‌کند تا از overfitting جلوگیری کند) و دو لایه خطی (یکی به ابعاد ۶۴ در ۱۴۰ و خروجی ۱۰۰ و دیگری لایه نهایی که تعداد خروجی‌ها را برابر با تعداد کلاس‌ها (num\_classes) تنظیم می‌کند) استفاده شده است.

## ۴-۱ آموخت

در کد این بخش، ابتدا تابعی به نام prepare\_dataloader تعریف می‌شود که پارامترهایی مانند ورودی‌ها ( $x$  و  $y$ )، سایز بج و یک گزینه برای CNN را دریافت می‌کند. سپس داده‌های ورودی  $x$  به یک تنسور با نوع داده‌ای float32 تبدیل می‌شود. اگر گزینه CNN فعال باشد، شکل تنسور به گونه‌ای تغییر می‌کند که بعد اضافی برای کanal‌ها ایجاد کند. خروجی  $y$  نیز به تنسور Long تبدیل می‌شود که یک واحد از آن کاهش می‌یابد. سپس داده‌ها در یک dataset به نام TensorDataset دسته‌بندی می‌شوند و در نهایت یک DataLoader با سایز بج مشخص و ترتیب تصادفی برای دسترسی آسان به داده‌ها ایجاد می‌شود. در انتها، این تابع برای داده‌های آموختی و آزمایشی فراخوانی می‌شود. کد مربوط به آن در زیر آورده شده است:

```
def prepare_dataloader(x, y, batch_size=64, cnn=False):
    X_tensor = torch.tensor(x, dtype=torch.float32)
    if cnn:
        X_tensor = X_tensor.view(X_tensor.shape[0], 1, -1)
    y_tensor = torch.tensor(y - 1, dtype=torch.long)
    dataset = TensorDataset(X_tensor, y_tensor)
    return DataLoader(dataset, batch_size=batch_size, shuffle=True)

train_loader = prepare_dataloader(X_train_norm, y_train_norm, cnn=False)
test_loader = prepare_dataloader(X_test_norm, y_test_norm, cnn=False)
```

در ادامه، تابع train\_model برای آموزش یک مدل یادگیری عمیق تعریف شده است. ابتدا دستگاه (GPU) یا (Adam) مشخص می‌شود و مدل به آن منتقل می‌شود. سپس تابع هزینه (Cross Entropy) و بهینه‌ساز (CPU) با نرخ یادگیری ۰.۰۰۱ ایجاد می‌شوند.

هم‌چنان، حلقه‌ای برای تعداد مشخصی از دوره‌ها (epochs) ایجاد می‌شود که در هر دوره، مدل در حالت آموزش قرار می‌گیرد. مقادیر خطا (Loss) و دقت (Accuracy) برای داده‌های آموزشی محاسبه و ذخیره می‌شود. پس از آموزش، مدل به حالت ارزیابی منتقل شده و مقادیر دقت و خطا برای داده‌های آزمایشی نیز محاسبه می‌شود. در نهایت، دقت‌ها و خسارت‌ها برای هر دوره چاپ شده و مقادیر نهایی بازگردانده می‌شوند.

```
def train_model(model, train_loader, test_loader, epochs=20):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

    train_loss, test_loss, train_acc, test_acc = [], [], [], []

    for epoch in range(epochs):
        model.train()
        epoch_loss, correct = 0, 0
        for X_batch, y_batch in train_loader:
            X_batch, y_batch = X_batch.to(device), y_batch.to(device)
            optimizer.zero_grad()
            outputs = model(X_batch)
            loss = criterion(outputs, y_batch)
            loss.backward()
            optimizer.step()

            epoch_loss += loss.item()
            correct += (outputs.argmax(1) == y_batch).sum().item()

        train_loss.append(epoch_loss / len(train_loader))
        train_acc.append(correct / len(train_loader.dataset))
```

```

model.eval()
total, correct = 0, 0
loss_eval = 0
with torch.no_grad():
    for X_batch, y_batch in test_loader:
        X_batch, y_batch = X_batch.to(device), y_batch.to(device)
        outputs = model(X_batch)
        loss_eval += criterion(outputs, y_batch).item()
        correct += (outputs.argmax(1) == y_batch).sum().item()
test_loss.append(loss_eval / len(test_loader))
test_acc.append(correct / len(test_loader.dataset))

print(f"Epoch {epoch+1}: Train Acc: {train_acc[-1]:.4f}, Test Acc: {test_acc[-1]:.4f}")

return train_loss, test_loss, train_acc, test_acc, model

```

## ۱-۵) ارزیابی

### ۱-۵-۱) رسم نمودارهای خطا (Loss) و دقت (Accuracy) برای داده‌های آموزش و آزمون و تحلیل هر کدام

برای ارزیابی دو شبکه طراحی شده، نمودارهای دقت (Accuracy) و خطا (Loss) برای داده‌های آموزش و

آزمون هر یک از شبکه‌ها رسم می‌شود. کد مربوط به آن در زیر آورده شده است:

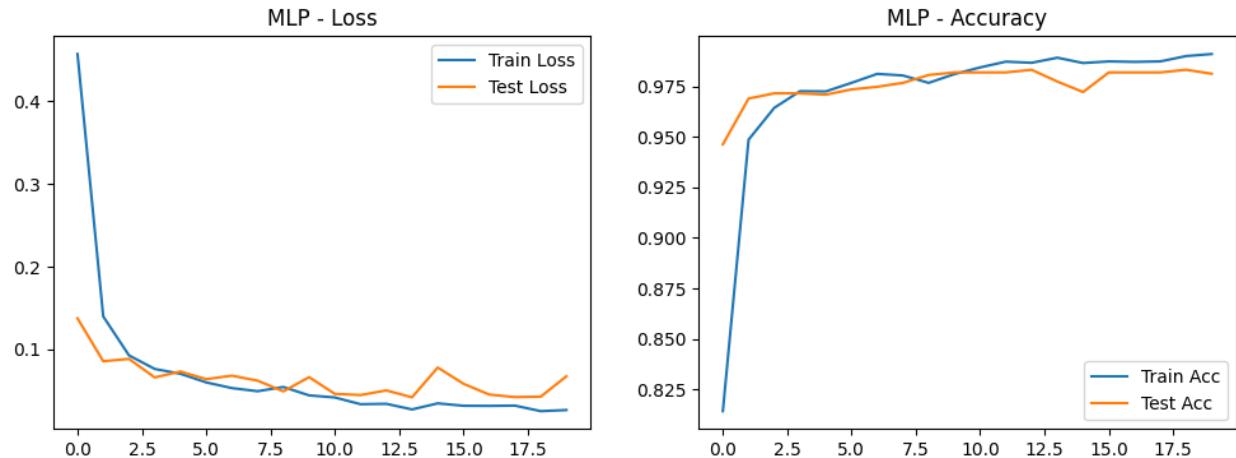
```

def plot_metrics(train_loss, test_loss, train_acc, test_acc, title='MLP'):
    plt.figure(figsize=(12, 4))
    plt.subplot(1, 2, 1)
    plt.plot(train_loss, label="Train Loss")
    plt.plot(test_loss, label="Test Loss")
    plt.title(f"{title} - Loss")
    plt.legend()

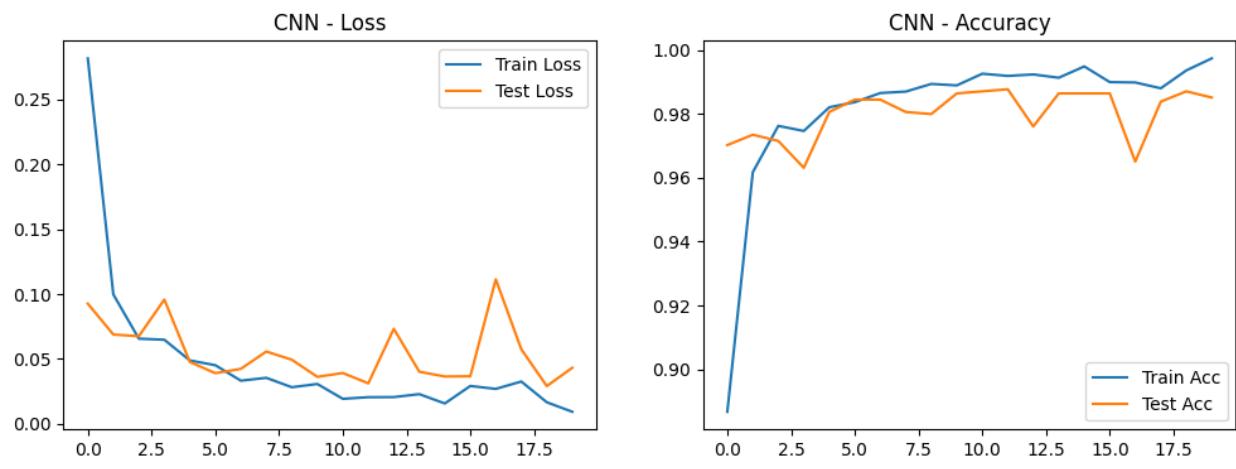
    plt.subplot(1, 2, 2)
    plt.plot(train_acc, label="Train Acc")
    plt.plot(test_acc, label="Test Acc")
    plt.title(f"{title} - Accuracy")
    plt.legend()
    plt.show()

```

که پاسخ آن به صورت زیر است:



شکل ۱: نمودارهای خطا و دقت برای داده‌های آموزش و آزمون شبکه MLP



شکل ۲: نمودارهای خطا و دقت برای داده‌های آموزش و آزمون شبکه CNN

### ► تحلیل نمودارهای MLP:

نمودار	آموزش و آزمون	تحلیل
نمودار خطا (Loss)	آموزش (train)	در ابتدا مقدار خطا نسبتاً بالا است (بیش از ۰.۴)، اما به سرعت کاهش یافته و پس از چند epoch به حدود ۰.۰۲ می‌رسد. روند نزولی یکنواخت است و نوسان خاصی دیده نمی‌شود که نشانه‌ی خوبی از یادگیری مناسب مدل روی داده‌های آموزش است.
	آزمون (test)	در ابتدا خطای آزمون نیز کاهش می‌یابد و تا حدود ۵ epoch در وضعیت خوبی قرار دارد. اما از آن به بعد کمی نوسان پیدا می‌کند. بهویشه در بازه‌ی

epoch های ۱۰ تا ۱۵ که افزایش نسبی دیده می شود، سپس دوباره کمی کاهش می یابد.		
افزایش سریع و قابل توجه در دقت در چند epoch اول. پس از آن به سطح بالای ۹۷.۵٪ می رسد و تقریباً ثابت می ماند، که نشان دهنده عملکرد بسیار خوب روی داده های آموزش است.	آموزش (train)	نمودار دقت (Accuracy)
مشابه روند آموزش در ابتدا افزایش سریعی دارد و به حدود ۹۷٪ می رسد. سپس در دوره های بعدی نوسانات جزئی دارد، اما به طور کلی در سطح بالایی باقی می ماند.	آزمون (test)	

## ۲-۵-۱ تحلیل نمودارهای CNN:

نمودار	آموزش و آزمون	تحلیل
	آموزش (train)	در ابتدا حدود ۰.۲۸ و با سرعت خوبی کاهش یافته. پس از حدود epoch ۵، مقدار loss در یک محدوده بسیار پایین (بین ۰.۰۱ تا ۰.۰۲) پایدار شده. این روند نشانه یادگیری موفق مدل روی داده های آموزش است.
نمودار خطا (Loss)	آزمون (test)	در ابتدا کاهش یافته ولی پس از آن نوساناتی در دوره های مختلف دیده می شود (مثلًا در epoch های ۴، ۱۰، ۱۵). در عین حال، میانگین مقدار loss روی داده های آزمون در محدوده ۰.۰۶ تا ۰.۰۳ باقی مانده که مقدار خوبی است. نوسانات زیادتر نسبت به Train Loss، ممکن است به دلیل پیچیدگی بیشتر داده های آزمون یا مقدار کم داده آزمون باشد.
نمودار دقت (Accuracy)	آموزش (train)	به سرعت به دقت بالای ۹۸٪ رسیده. بعد از ۵ epoch به بعد، تقریباً یکنواخت و با دقت بالای ۹۹٪ ادامه دارد.
نمودار دقت (Accuracy)	آزمون (test)	رونده صعودی اولیه دارد و به بالای ۹۸٪ می رسد. سپس کمی نوسان دارد (در epoch های ۶، ۱۲ و ۱۵ کمی افت کرده). با این حال، دقت آزمون همچنان بالا و قابل قبول است و تفاوت چندانی با دقت آموزش ندارد.

## ۲-۵-۲) دقت نهایی مدل روی داده های آزمون

در آخر دقت نهایی مدل روی داده های آزمون به دست آورده می شود:

```

mlp = MLP()
tr_loss, ts_loss, tr_acc, ts_acc, mlp_model = train_model(mlp, train_loader, test_loader)
plot_metrics(tr_loss, ts_loss, tr_acc, ts_acc, title='MLP')
plot_confusion_matrix(mlp_model, test_loader, title='MLP')
train_loader_cnn = prepare_dataloader(X_train_norm, y_train_norm, cnn=True)
test_loader_cnn = prepare_dataloader(X_test_norm, y_test_norm, cnn=True)
cnn = CNN()
train_loss_cnn, test_loss_cnn, train_acc_cnn, test_acc_cnn, cnn_model = train_model(
    cnn, train_loader_cnn, test_loader_cnn, epochs=20)
plot_metrics(train_loss_cnn, test_loss_cnn, train_acc_cnn, test_acc_cnn, title='CNN')
plot_confusion_matrix(cnn_model, test_loader_cnn, title='CNN')

```

که پاسخ آن برای شبکه MLP به صورت زیر است:

```

Epoch 1: Train Acc: 0.8143, Test Acc: 0.9463
Epoch 2: Train Acc: 0.9486, Test Acc: 0.9689
Epoch 3: Train Acc: 0.9644, Test Acc: 0.9715
Epoch 4: Train Acc: 0.9726, Test Acc: 0.9715
Epoch 5: Train Acc: 0.9725, Test Acc: 0.9709
Epoch 6: Train Acc: 0.9767, Test Acc: 0.9735
Epoch 7: Train Acc: 0.9812, Test Acc: 0.9748
Epoch 8: Train Acc: 0.9804, Test Acc: 0.9767
Epoch 9: Train Acc: 0.9767, Test Acc: 0.9806
Epoch 10: Train Acc: 0.9810, Test Acc: 0.9819
Epoch 11: Train Acc: 0.9844, Test Acc: 0.9819
Epoch 12: Train Acc: 0.9872, Test Acc: 0.9819
Epoch 13: Train Acc: 0.9866, Test Acc: 0.9832
Epoch 14: Train Acc: 0.9891, Test Acc: 0.9773
Epoch 15: Train Acc: 0.9865, Test Acc: 0.9722
Epoch 16: Train Acc: 0.9873, Test Acc: 0.9819
Epoch 17: Train Acc: 0.9871, Test Acc: 0.9819
Epoch 18: Train Acc: 0.9873, Test Acc: 0.9819
Epoch 19: Train Acc: 0.9899, Test Acc: 0.9832
Epoch 20: Train Acc: 0.9910, Test Acc: 0.9812

```

و برای شبکه CNN به صورت زیر است:

```

Epoch 1: Train Acc: 0.8867, Test Acc: 0.9702
Epoch 2: Train Acc: 0.9617, Test Acc: 0.9735
Epoch 3: Train Acc: 0.9762, Test Acc: 0.9715
Epoch 4: Train Acc: 0.9746, Test Acc: 0.9631
Epoch 5: Train Acc: 0.9821, Test Acc: 0.9806
Epoch 6: Train Acc: 0.9837, Test Acc: 0.9845
Epoch 7: Train Acc: 0.9865, Test Acc: 0.9845
Epoch 8: Train Acc: 0.9870, Test Acc: 0.9806
Epoch 9: Train Acc: 0.9894, Test Acc: 0.9799
Epoch 10: Train Acc: 0.9889, Test Acc: 0.9864
Epoch 11: Train Acc: 0.9926, Test Acc: 0.9871
Epoch 12: Train Acc: 0.9919, Test Acc: 0.9877
Epoch 13: Train Acc: 0.9923, Test Acc: 0.9761
Epoch 14: Train Acc: 0.9913, Test Acc: 0.9864
Epoch 15: Train Acc: 0.9949, Test Acc: 0.9864
Epoch 16: Train Acc: 0.9899, Test Acc: 0.9864
Epoch 17: Train Acc: 0.9898, Test Acc: 0.9650
Epoch 18: Train Acc: 0.9880, Test Acc: 0.9838
Epoch 19: Train Acc: 0.9936, Test Acc: 0.9871
Epoch 20: Train Acc: 0.9974, Test Acc: 0.9851

```

هر دو مدل MLP و CNN عملکرد خوبی روی مجموعه داده‌های UCI HAR دارند. دقت نهایی آنها روی داده‌های آموزش و آزمون در جدول زیر آورده شده است:

Final Test Accuracy	Final Train Accuracy	Model
98.12%	99.10%	MLP
98.51%	99.74%	CNN

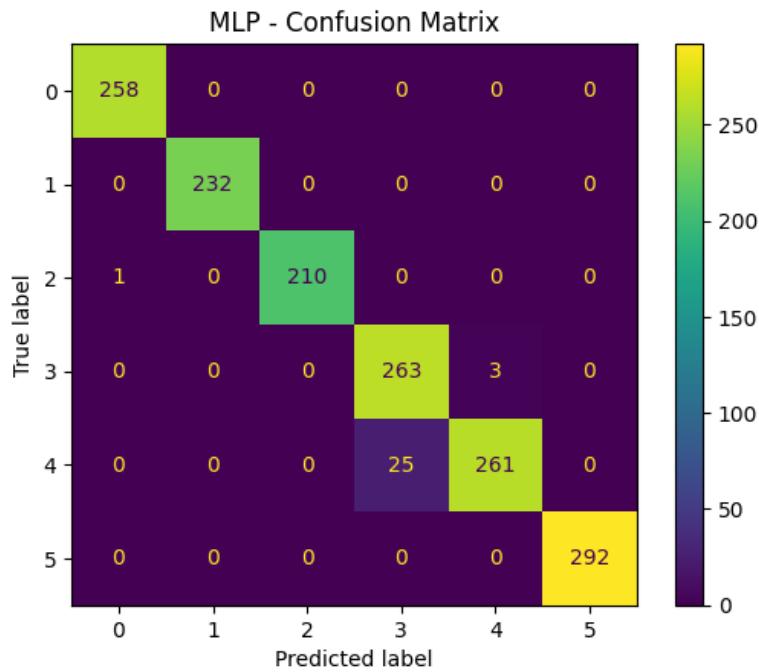
### ۳-۵-۱) رسم ماتریس آشتفتگی برای داده‌های آزمون و تفسیر آن

ماتریس‌های آشتفتگی برای داده‌های آزمون شبکه‌های MLP و CNN با استفاده از کد زیر رسم می‌شود:

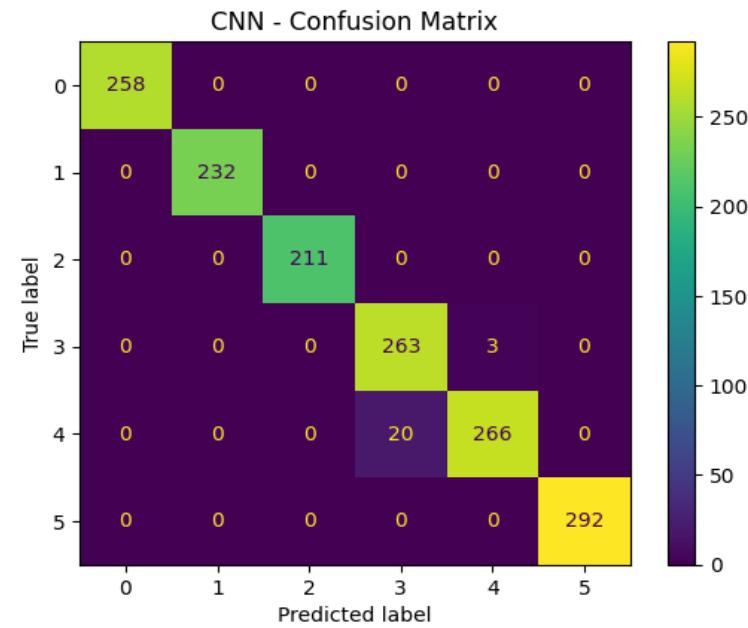
```
def plot_confusion_matrix(model, test_loader, title='MLP'):
    all_preds = []
    all_labels = []
    model.eval()
    with torch.no_grad():
        for X_batch, y_batch in test_loader:
            outputs = model(X_batch)
            preds = outputs.argmax(1).cpu().numpy()
            all_preds.extend(preds)
            all_labels.extend(y_batch.numpy())

    cm = confusion_matrix(all_labels, all_preds)
    disp = ConfusionMatrixDisplay(cm)
    disp.plot()
    plt.title(f'{title} - Confusion Matrix')
    plt.show()
```

که پاسخ آن به صورت زیر است:



شکل ۳: ماتریس آشتفتگی برای داده‌های آزمون شبکه MLP



شکل ۴: ماتریس آشتفتگی برای داده‌های آزمون شبکه CNN

» تفسیر نمودارهای ماتریس آشتفتگی:

هر ماتریس آشفتگی یک شبکه  $6 \times 6$  برای ۶ کلاس فعالیت (با برجسب‌های ۰ تا ۵) است که لیبل و درصد طبقه‌بندی درست هر یک از لیبل‌ها در شبکه‌های MLP و CNN در جدول زیر ارائه شده است:

Description	Label	Class
100% (258/258)	WALKING	0
100% (232/232)	WALKING_UPSTAIRS	1
MLP: 99.53% (210/211) CNN: 100% (211/211)	WALKING_DOWNSTAIRS	2
100% (263/263)	SITTING	3
MLP: 91.26% (261/286) CNN: 93% (266/286)	STANDING	4
100% (292/292)	LAYING	5

کلاس‌های ۰، ۱، ۳ و ۵ یعنی LAYING و SITTING، WALKING\_UPSTAIRS و WALKING پیش‌بینی‌های بی‌نقص در هر دو شبکه MLP و CNN هستند. فقط دو کلاس ۲ و ۴ دارای خطای جزئی هستند که در زیر بطور خلاصه توضیح داده شده است:

#### کلاس ۲: WALKING\_DOWNSTAIRS

- MLP: ۱ مورد به اشتباه در کلاس صفر طبقه‌بندی شده است.

- CNN: پیش‌بینی بی‌نقص

#### کلاس ۴: STANDING

- MLP: ۲۵ مورد به اشتباه در کلاس ۳ طبقه‌بندی شده‌اند.

- CNN: ۲۰ مورد به اشتباه در کلاس ۳ طبقه‌بندی شده‌اند.

در مجموع، CNN نسبت به MLP دقیق‌تر است، به خصوص در تشخیص WALKING\_DOWNSTAIRS و STANDING

## ۱-۶) تحلیل

به منظور مقایسه بهتر و دقیق‌تر بین دو شبکه CNN و MLP، خلاصه نتایج به دست آمده در جدول زیر آورده شده است:

مقایسه	CNN	MLP	معیار
CNN بهتر است	حدود٪۹۹	حدود٪۹۷.۵	<b>Train Accuracy</b>
CNN بهتر است	حدود٪۹۸.۵	حدود٪۹۷	<b>Test Accuracy</b>
CNN بهتر است	حدود ۰.۰۱ یا کمتر	حدود ۰.۰۲	<b>Train Loss</b>
MLP کمی بهتر است	حدود ۰.۰۴ با نوسان زیاد	حدود ۰.۰۵ با نوسان کم	<b>Test Loss</b>
هر دو برابر هستند	تا حدودی کنترل شده	کم و کنترل شده	<b>Overfitting</b>
CNN بهتر است	بیشتر	کمتر	<b>پیش‌بینی طبقه‌بندی</b>

به طور کلی شبکه CNN عملکرد بهتری از MLP دارد. این برتری در دقت نهایی، خطای آموزش کمتر و توان یادگیری بیشتر دیده می‌شود. علت‌های برتری مدل CNN نسبت به MLP در جدول زیر ارائه شده است:

توضیح	عامل
CNN از فیلترهای مکانی (convolutional filters) استفاده می‌کند که ویژگی‌های محلی (مانند لبه‌ها و بافت‌ها) را بهتر استخراج می‌کند، اما MLP تمام ویژگی‌ها را به صورت مسطح (flattened) و بدون لحاظ ساختار فضایی تصویر پردازش می‌کند.	ساختار تخصصی‌تر برای داده‌های تصویری
CNN با استفاده از اشتراک وزن (weight sharing) تعداد پارامترها را کاهش داده و احتمال overfitting را پایین می‌آورد.	تعداد پارامترهای کمتر (در عمل)
دقت بالاتر روی داده‌های آزمون نشان می‌دهد CNN بهتر قادر به تعمیم آموخته‌ها به داده‌های جدید است.	تعمیم‌پذیری بهتر به داده‌های آزمون
CNN قادر است ویژگی‌های مهم را خودش از تصویر یاد بگیرد، در حالی که MLP به ویژگی‌های از پیش استخراج شده وابسته‌تر است.	کاهش نیاز به پیش‌پردازش دستی

## (۲) بخش دوم: دادگان NEU Surface Defects

### » معرفی کتابخانه‌های استفاده شده

```

import os
import cv2
import numpy as np
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras.utils import to_categorical
from glob import glob
from tensorflow.keras.preprocessing.image import load_img, img_to_array
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense, Dropout
from tensorflow.keras.layers import Conv2D, MaxPooling2D, BatchNormalization, SpatialDropout2D
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

```

بعضی از کتابخانه‌های استفاده شده در این بخش، در بخش قبل (UCI HAR) نیز به کار رفته اند (صفحه ۵) که توضیحات آنها دیگر اینجا داده نمی‌شود و فقط کتابخانه‌های جدید معرفی خواهند شد:

- کتابخانه cv2 برای پردازش تصویر به کار می‌رود.
- کتابخانه tensorflow برای یادگیری عمیق و شبکه‌های عصبی استفاده می‌شود.
- tensorflow.keras.utils برای تبدیل برچسب‌ها به فرمت دسته‌بندی (One-hot encoding) به کار می‌رود.
- glob برای جستجو و لیست کردن فایل‌ها در دایرکتوری‌ها استفاده می‌شود.
- tensorflow.keras.preprocessing.image برای بارگذاری تصاویر و تبدیل آن‌ها به آرایه‌های عددی به کار می‌رود.
- tensorflow.keras.models برای تعریف مدل به صورت ساده و خطی (جایی که لایه‌ها پشت‌سرهم قرار می‌گیرند) استفاده می‌شود.
- tensorflow.keras.layers برای ایجاد لایه‌های مختلف شبکه عصبی مسطح‌سازی ویژگی‌ها برای اتصال به لایه به کار می‌رود.
- tensorflow.keras.layers استفاده برای ایجاد شبکه‌های کانولوشنی (Convolutional Neural Networks) می‌شود.

## ۲-۱) آماده‌سازی داده‌ها

### ۲-۱-۱) انتخاب یکی از روش‌های نرمال‌سازی و پیاده‌سازی روی داده‌ها

از روش Min-Max Scaling برای نرمال‌سازی تصاویر استفاده شده است. تصاویر RGB دارای مقادیری در بازه‌ی ۰ تا ۲۵۵ هستند. با تقسیم تمام مقادیر پیکسل‌ها بر ۲۵۵، تمامی مقادیر به بازه‌ی [۰, ۱] منتقل می‌شوند که در کد زیر آورده شده است:

```
def load_images_labels(data_dir, img_size=(64, 64)):
    images = []
    labels = []
    print(f"Loading from: {data_dir}")
    for class_name in os.listdir(data_dir):
        class_path = os.path.join(data_dir, class_name)
        print(f"Reading from: {class_path}")
        if not os.path.isdir(class_path):
            continue
        for img_file in os.listdir(class_path):
            if img_file.endswith(".jpg"):
                img_path = os.path.join(class_path, img_file)
                try:
                    img = load_img(img_path, target_size=img_size)
                    img = img_to_array(img) / 255.0
                    images.append(img)
                    labels.append(class_name)
                except Exception as e:
                    print(f"Failed to load {img_path}: {e}")
    return np.array(images), np.array(labels)
```

تابع load\_images\_labels به عنوان ورودی، مسیر دایرکتوری داده‌ها و اندازه تصویر می‌گیرد. ابتدا لیست‌های خالی برای تصاویر و برچسب‌ها ایجاد می‌کند و پیامی برای شروع بارگذاری نمایش می‌دهد. سپس برای هر کلاس در دایرکتوری، مسیر آن کلاس را می‌سازد و بررسی می‌کند که آیا مسیر مربوطه یک دایرکتوری است یا نه. اگر دایرکتوری باشد، به داخل آن رفته و برای هر فایل تصویری که پسوند jpg دارد، مسیر کامل تصویر را می‌سازد. سپس تصویر را با اندازه مشخص بارگذاری کرده و به آرایه تبدیل می‌کند و نرمالیزه می‌کند. تصویر و برچسب کلاس به لیست‌های مربوطه اضافه می‌شوند. اگر در بارگذاری تصویری خطایی پیش بیاید، پیام خطا نمایش داده می‌شود. در نهایت، آرایه‌های numpy از تصاویر و برچسب‌ها را بر می‌گرداند.

از دلایل استفاده از این روش در این بخش می‌توان به موارد زیر اشاره کرد:

- قرار دادن داده‌ها در بازه‌ی استاندارد (۰ تا ۱) باعث می‌شود داده‌ها مقیاس یکسانی داشته باشند.
- هنگام آموزش شبکه‌های عصبی با گرادیان نزولی، داشتن ورودی‌هایی در یک بازه‌ی کوچک‌تر (مثل ۰ تا ۱) باعث پایداری عددی بهتر، یادگیری سریع‌تر و کاهش خطر نوسانات زیاد در وزن‌ها می‌شود.
- در تصاویر خام، پیکسل‌هایی با مقادیر بزرگ‌تر ممکن است بر مدل غلبه کنند. نرمال‌سازی باعث می‌شود هیچ پیکسلی تأثیر غیرمعمولی نداشته باشد.
- عده‌های کوچک‌تر و در بازه‌ی ۰ تا ۱ معمولاً محاسبات را سریع‌تر و دقیق‌تر می‌کنند.

## ۲-۱-۲) اختصاص ۸۵٪ و ۱۵٪ داده‌های نرمال شده به آموزش و آزمون و ذکر ابعاد هر بخش

ابتدا تصاویر آموزشی و اعتبارسنجی بارگذاری و به اندازه ۱۲۸ در ۱۲۸ پیکسل تغییر اندازه داده می‌شوند. سپس، تعداد نمونه‌ها چاپ شده و تصاویر و برچسب‌ها ترکیب می‌شوند و برچسب‌ها با استفاده از LabelEncoder کدگذاری می‌شوند. در مرحله بعد، ۸۵٪ داده‌های نرمال شده به آموزش و ۱۵٪ به آزمون با random-state=42 اختصاص داده می‌شوند و در آخر ابعاد هر بخش چاپ می‌شود.

```
train_img_dir = r"G:\Artificial_Intelligence\Tamrin\Tamrin4\HW4\NEU-DET\train\images"
val_img_dir = r"G:\Artificial_Intelligence\Tamrin\Tamrin4\HW4\NEU-DET\validation\images"

train_images, train_labels = load_images_labels(train_img_dir, img_size=(128, 128))
val_images, val_labels = load_images_labels(val_img_dir, img_size=(128, 128))

print("Loaded train samples:", len(train_images))
print("Loaded val samples:", len(val_images))

X = np.concatenate((train_images, val_images))
y = np.concatenate((train_labels, val_labels))

le = LabelEncoder()
y_encoded = to_categorical(le.fit_transform(y))

X_train, X_test, y_train, y_test = train_test_split(
    X, y_encoded, test_size=0.15, random_state=42, stratify=y_encoded
)

print(f"Training shape: {X_train.shape}, {y_train.shape}")
print(f"Testing shape: {X_test.shape}, {y_test.shape}")
```

پاسخ کد فوق به صورت زیر است:

```

Loading from: G:\Artificial_Intelligence\Tamrin\Tamrin4\HW4\NEU-DET\train\images
Reading from: G:\Artificial_Intelligence\Tamrin\Tamrin4\HW4\NEU-DET\train\images\crazing
Reading from: G:\Artificial_Intelligence\Tamrin\Tamrin4\HW4\NEU-DET\train\images\inclusion
Reading from: G:\Artificial_Intelligence\Tamrin\Tamrin4\HW4\NEU-DET\train\images\patches
Reading from: G:\Artificial_Intelligence\Tamrin\Tamrin4\HW4\NEU-DET\train\images\pitted_surface
Reading from: G:\Artificial_Intelligence\Tamrin\Tamrin4\HW4\NEU-DET\train\images\rolled-in_scale
Reading from: G:\Artificial_Intelligence\Tamrin\Tamrin4\HW4\NEU-DET\train\images\scratches
Loading from: G:\Artificial_Intelligence\Tamrin\Tamrin4\HW4\NEU-DET\validation\images
Reading from: G:\Artificial_Intelligence\Tamrin\Tamrin4\HW4\NEU-DET\validation\images\crazing
Reading from: G:\Artificial_Intelligence\Tamrin\Tamrin4\HW4\NEU-DET\validation\images\inclusion
Reading from: G:\Artificial_Intelligence\Tamrin\Tamrin4\HW4\NEU-DET\validation\images\patches
Reading from: G:\Artificial_Intelligence\Tamrin\Tamrin4\HW4\NEU-DET\validation\images\pitted_surface
Reading from: G:\Artificial_Intelligence\Tamrin\Tamrin4\HW4\NEU-DET\validation\images\rolled-in_scale
Reading from: G:\Artificial_Intelligence\Tamrin\Tamrin4\HW4\NEU-DET\validation\images\scratches
Loaded train samples: 1440
Loaded val samples: 360
Training shape: (1530, 128, 128, 3), (1530, 6)
Testing shape: (270, 128, 128, 3), (270, 6)

```

همانطور که در پاسخ کد قابل مشاهده است، ابعاد هر بخش به صورت زیر به دست می‌آید:

ابعاد هر بخش:

Loaded train samples: 1440

Loaded val samples: 360

Training shape: (1530, 128, 128, 3), (1530, 6)

Testing shape: (270, 128, 128, 3), (270, 6)

## ۲-۲) طراحی شبکه MLP

برای طراحی شبکه MLP از لایه‌های Dense، Flatten، ReLU، Dropout و softmax تابع فعال‌سازی

استفاده می‌شود که دلایل استفاده از آنها در جدول زیر ارائه شده است:

لایه / مؤلفه	جزئیات	دلیل انتخاب
Flatten	Flatten(input_shape=(128, 128, 3))	تبديل ورودی سه‌بعدی تصویر به بردار یک‌بعدی برای تغذیه به لایه‌های Dense

استخراج ویژگی‌های غیرخطی و افزایش قدرت یادگیری مدل	Dense(256, activation='relu')	<b>Dense (256 units, ReLU)</b>
جلوگیری از بیشبرازش(Overfitting) با حذف تصادفی ۳۰٪ نورون‌ها در زمان آموزش	Dropout(0.3)	<b>Dropout (rate=0.3)</b>
کاهش ابعاد ویژگی‌ها و افزایش ویژگی‌ها	Dense(128, activation='relu')	<b>Dense (128 units, ReLU)</b>
لایه خروجی برای طبقه‌بندی چند کلاسه (۶ کلاس)، خروجی احتمال برای هر کلاس	Dense(6, activation='softmax')	<b>Dense (6 units, softmax)</b>
مناسب برای طبقه‌بندی چند کلاسه با One-hot برچسب‌های	categorical_crossentropy	<b>Loss Function</b>
بهینه‌ساز سریع و پایدار	Adam(0.001)	<b>Optimizer</b>
معیار استاندارد برای ارزیابی عملکرد طبقه‌بندی‌ها	accuracy	<b>Metrics</b>

بنابراین، برای طراحی شبکه MLP از کد زیر استفاده شده است:

```
mlp_model = Sequential([
    Flatten(input_shape=(128, 128, 3)),
    Dense(256, activation='relu'),
    Dropout(0.3),
    Dense(128, activation='relu'),
    Dense(6, activation='softmax')
])

mlp_model.compile(optimizer=tf.keras.optimizers.Adam(0.001),
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])

mlp_model.summary()
```

در این کد، یک مدل شبکه عصبی MLP تعریف می‌شود که شامل:

لایه Flatten: تبدیل ورودی ۱۲۸ در ۱۲۸ در ۳ به آرایه یک بعدی.

۲. لایه Dense (۲۵۶ نورون): با تابع فعال‌سازی ReLU برای یادگیری ویژگی‌های پیچیده.

۳. لایه Dropout (نرخ ۰.۳): جلوگیری از بیشبرازش.

۴. لایه Dense (۱۲۸ نورون): با تابع فعال‌سازی ReLU برای استخراج ویژگی‌های بیشتر.

۵. لایه Dense (۶ نورون): با تابع فعال‌سازی Softmax برای طبقه‌بندی به ۶ کلاس.

است. مدل با استفاده از Adam به عنوان بهینه‌ساز و تابع هزینه categorical\_crossentropy کامپایل می‌شود.

جزئیات مدل با تابع summary نمایش داده می‌شود که به صورت زیر است:

Model: "sequential"

Layer (type)	Output Shape	Param #
<hr/>		
flatten (Flatten)	(None, 49152)	0
dense (Dense)	(None, 256)	12583168
dropout (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 128)	32896
dense_2 (Dense)	(None, 6)	774
<hr/>		
Total params: 12616838 (48.13 MB)		
Trainable params: 12616838 (48.13 MB)		
Non-trainable params: 0 (0.00 Byte)		

## CNN طراحی شبکه (۳-۲)

برای طراحی شبکه CNN از سه لایه کانولوشن دو بعدی (Conv2D)، نرمال‌سازی (BatchNormalization)، Loss softmax، دو بعدی Loss، لایه‌های ReLU، Dropout، Flatten و MaxPooling است:metrics و Optimizer Function استفاده می‌شود که در جدول زیر آنها در جدول زیر ارائه شده است:

لایه / مؤلفه	جزئیات	دلیل انتخاب
Conv2D (32)	Conv2D(32, (3, 3), activation='relu')	استخراج ویژگی‌های محلی از تصویر با استفاده از فیلترهای ۳ در ۳
BatchNormalization	بعد از هر Conv2D	نرمال‌سازی ویژگی‌ها برای تسريع آموزش و کاهش حساسیت به مقدار اولیه‌سازی
MaxPooling2D (2x2)	MaxPooling2D(2, 2)	کاهش ابعاد فضایی و جلوگیری از overfitting با انتخاب ویژگی‌های مهم
Conv2D (64)	Conv2D(64, (3, 3), activation='relu')	افزایش عمق ویژگی‌ها و توانایی شناسایی الگوهای پیچیده‌تر

استخراج ویژگی‌های سطح بالا از تصویر	Conv2D(128, (3, 3), activation='relu')	<b>Conv2D (128)</b>
تبديل ویژگی‌های استخراج شده به بردار ۱ بعدی برای اتصال به لایه Dense	Flatten()	<b>Flatten</b>
کاهش خطر overfitting با حذف تصادفی نیمی از نورون‌ها در مرحله آموزش	Dropout(0.5)	<b>Dropout (0.5)</b>
ترکیب ویژگی‌های استخراج شده و یادگیری تصمیم نهایی	Dense(128, activation='relu')	<b>Dense (128 units, ReLU)</b>
لایه خروجی برای طبقه‌بندی ۶ کلاسه با احتمال‌های نرم	Dense(6, activation='softmax')	<b>Dense (6 units, softmax)</b>
مناسب برای مسائل طبقه‌بندی One-hot چند کلاسه با برچسب	categorical_crossentropy	<b>Loss Function</b>
الگوریتم قدرتمند و پرکاربرد با نرخ یادگیری ثابت برای آموزش پایدار	Adam(0.001)	<b>Optimizer</b>
معیار عمومی برای ارزیابی دقت طبقه‌بندی	accuracy	<b>Metrics</b>

بنابراین، برای طراحی شبکه CNN از کد زیر استفاده شده است:

```
cnn_model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(128, 128, 3)),
    BatchNormalization(),
    MaxPooling2D(2, 2),

    Conv2D(64, (3, 3), activation='relu'),
    BatchNormalization(),
    MaxPooling2D(2, 2),

    Conv2D(128, (3, 3), activation='relu'),
    BatchNormalization(),
    MaxPooling2D(2, 2),

    Flatten(),
    Dropout(0.5),
    Dense(128, activation='relu'),
    Dense(6, activation='softmax')
])

cnn_model.compile(optimizer=tf.keras.optimizers.Adam(0.001),
                   loss='categorical_crossentropy',
                   metrics=['accuracy'])

cnn_model.summary()
```

این کد یک مدل CNN با Keras تعریف می‌کند. شامل سه لایه کانولوشن، نرمال‌سازی و ماکس‌پولینگ برای استخراج ویژگی‌های تصویر است. لایه‌های نرمال‌سازی و ماکس‌پولینگ به بهود عملکرد و کاهش ابعاد کمک می‌کنند. سپس خروجی به لایه صاف (Flatten) و سپس به دو لایه Dense (یکی با ۱۲۸ نورون و ReLU، دیگری با ۶ نورون و Softmax) منتقل می‌شود. مدل با بهینه‌ساز Adam وتابع هزینه categorical\_crossentropy کامپایل شده و دقت به عنوان metric انتخاب شده است. جزئیات مدل با تابع summary نمایش داده می‌شود که

به صورت زیر است:

Model: "sequential\_1"

Layer (type)	Output Shape	Param #
<hr/>		
conv2d (Conv2D)	(None, 126, 126, 32)	896
batch_normalization (Batch Normalization)	(None, 126, 126, 32)	128
max_pooling2d (MaxPooling2D)	(None, 63, 63, 32)	0
conv2d_1 (Conv2D)	(None, 61, 61, 64)	18496
batch_normalization_1 (BatchNormalization)	(None, 61, 61, 64)	256
max_pooling2d_1 (MaxPooling2D)	(None, 30, 30, 64)	0
conv2d_2 (Conv2D)	(None, 28, 28, 128)	73856
batch_normalization_2 (BatchNormalization)	(None, 28, 28, 128)	512
max_pooling2d_2 (MaxPooling2D)	(None, 14, 14, 128)	0
flatten_1 (Flatten)	(None, 25088)	0
dropout_1 (Dropout)	(None, 25088)	0
dense_3 (Dense)	(None, 128)	3211392
dense_4 (Dense)	(None, 6)	774
<hr/>		
Total params: 3306310 (12.61 MB)		
Trainable params: 3305862 (12.61 MB)		
Non-trainable params: 448 (1.75 KB)		

## ۴-۲ آموزش

برای آموزش هر دو مدل MLP و CNN از روش fit استفاده می‌شود. داده‌های ورودی (x\_train) و برچسبها (y\_train) به مدل داده می‌شوند و در ۲۵ دوره (epochs) آموزش می‌بینند. همچنین، داده‌های تست (x\_test) و (y\_test) برای اعتبارسنجی مدل استفاده می‌شوند. سایز دسته‌ها نیز ۳۲ تعیین شده است.

```
history_cnn = cnn_model.fit(X_train, y_train, epochs=25, validation_data=(X_test, y_test), batch_size=32)
history_mlp = mlp_model.fit(X_train, y_train, epochs=25, validation_data=(X_test, y_test), batch_size=32)
```

که پاسخ مدل CNN آن به صورت زیر است:

```
Epoch 1/25
48/48 [=====] - 65s 1s/step - loss: 3.5026 - accuracy: 0.7046 - val_loss: 25.5143 - val_accuracy: 0.1889
Epoch 2/25
48/48 [=====] - 65s 1s/step - loss: 1.9875 - accuracy: 0.8111 - val_loss: 29.9988 - val_accuracy: 0.1704
Epoch 3/25
48/48 [=====] - 65s 1s/step - loss: 1.5657 - accuracy: 0.8425 - val_loss: 29.9949 - val_accuracy: 0.1630
Epoch 4/25
48/48 [=====] - 65s 1s/step - loss: 1.1393 - accuracy: 0.8706 - val_loss: 21.6733 - val_accuracy: 0.1667
Epoch 5/25
48/48 [=====] - 66s 1s/step - loss: 1.0960 - accuracy: 0.8752 - val_loss: 21.4506 - val_accuracy: 0.2926
Epoch 6/25
48/48 [=====] - 65s 1s/step - loss: 1.0390 - accuracy: 0.8882 - val_loss: 26.8674 - val_accuracy: 0.1852
Epoch 7/25
48/48 [=====] - 64s 1s/step - loss: 0.7885 - accuracy: 0.9000 - val_loss: 20.6184 - val_accuracy: 0.1815
Epoch 8/25
48/48 [=====] - 64s 1s/step - loss: 0.4264 - accuracy: 0.9366 - val_loss: 31.4153 - val_accuracy: 0.1704
Epoch 9/25
48/48 [=====] - 64s 1s/step - loss: 1.1498 - accuracy: 0.8948 - val_loss: 18.4412 - val_accuracy: 0.2074
Epoch 10/25
48/48 [=====] - 66s 1s/step - loss: 0.5928 - accuracy: 0.9242 - val_loss: 17.4119 - val_accuracy: 0.4000
Epoch 11/25
48/48 [=====] - 65s 1s/step - loss: 0.3198 - accuracy: 0.9431 - val_loss: 14.7271 - val_accuracy: 0.3593
Epoch 12/25
48/48 [=====] - 64s 1s/step - loss: 0.3814 - accuracy: 0.9471 - val_loss: 16.1932 - val_accuracy: 0.5148
Epoch 13/25
48/48 [=====] - 65s 1s/step - loss: 0.6592 - accuracy: 0.9261 - val_loss: 41.6186 - val_accuracy: 0.3037
Epoch 14/25
48/48 [=====] - 64s 1s/step - loss: 0.3298 - accuracy: 0.9529 - val_loss: 43.7240 - val_accuracy: 0.4037
Epoch 15/25
48/48 [=====] - 64s 1s/step - loss: 0.5169 - accuracy: 0.9399 - val_loss: 16.2610 - val_accuracy: 0.4111
Epoch 16/25
48/48 [=====] - 64s 1s/step - loss: 0.2194 - accuracy: 0.9569 - val_loss: 6.1653 - val_accuracy: 0.7370
Epoch 17/25
48/48 [=====] - 65s 1s/step - loss: 0.3247 - accuracy: 0.9523 - val_loss: 5.6654 - val_accuracy: 0.7519
Epoch 18/25
48/48 [=====] - 64s 1s/step - loss: 0.1565 - accuracy: 0.9686 - val_loss: 1.6690 - val_accuracy: 0.6667
Epoch 19/25
48/48 [=====] - 64s 1s/step - loss: 0.0934 - accuracy: 0.9752 - val_loss: 1.3648 - val_accuracy: 0.8889
Epoch 20/25
48/48 [=====] - 65s 1s/step - loss: 0.0833 - accuracy: 0.9797 - val_loss: 0.3889 - val_accuracy: 0.9593
Epoch 21/25
48/48 [=====] - 65s 1s/step - loss: 0.0910 - accuracy: 0.9804 - val_loss: 9.4811 - val_accuracy: 0.7074
Epoch 22/25
48/48 [=====] - 69s 1s/step - loss: 0.1223 - accuracy: 0.9778 - val_loss: 2.7283 - val_accuracy: 0.7704
Epoch 23/25
48/48 [=====] - 69s 1s/step - loss: 0.0566 - accuracy: 0.9856 - val_loss: 2.9173 - val_accuracy: 0.8370
Epoch 24/25
48/48 [=====] - 68s 1s/step - loss: 0.0788 - accuracy: 0.9843 - val_loss: 1.0645 - val_accuracy: 0.9037
Epoch 25/25
48/48 [=====] - 62s 1s/step - loss: 0.0584 - accuracy: 0.9863 - val_loss: 1.6465 - val_accuracy: 0.8704
```

و پاسخ مدل آن به صورت زیر است:

```

Epoch 1/25
48/48 [=====] - 12s 225ms/step - loss: 9.9305 - accuracy: 0.1915 - val_loss: 1.7505 - val_accuracy: 0.2519
Epoch 2/25
48/48 [=====] - 10s 219ms/step - loss: 1.7800 - accuracy: 0.2268 - val_loss: 1.7006 - val_accuracy: 0.2519
Epoch 3/25
48/48 [=====] - 11s 222ms/step - loss: 1.7162 - accuracy: 0.2281 - val_loss: 1.7785 - val_accuracy: 0.2074
Epoch 4/25
48/48 [=====] - 11s 220ms/step - loss: 1.7266 - accuracy: 0.2124 - val_loss: 1.6914 - val_accuracy: 0.2704
Epoch 5/25
48/48 [=====] - 11s 221ms/step - loss: 1.7006 - accuracy: 0.2320 - val_loss: 1.6754 - val_accuracy: 0.2630
Epoch 6/25
48/48 [=====] - 12s 240ms/step - loss: 1.7193 - accuracy: 0.2216 - val_loss: 1.6610 - val_accuracy: 0.2815
Epoch 7/25
48/48 [=====] - 12s 248ms/step - loss: 1.6977 - accuracy: 0.2340 - val_loss: 1.7362 - val_accuracy: 0.2185
Epoch 8/25
48/48 [=====] - 13s 268ms/step - loss: 1.7523 - accuracy: 0.1987 - val_loss: 1.7388 - val_accuracy: 0.2370
Epoch 9/25
48/48 [=====] - 12s 260ms/step - loss: 1.6673 - accuracy: 0.2582 - val_loss: 1.6228 - val_accuracy: 0.3185
Epoch 10/25
48/48 [=====] - 12s 258ms/step - loss: 1.6531 - accuracy: 0.2693 - val_loss: 1.7012 - val_accuracy: 0.2370
Epoch 11/25
48/48 [=====] - 12s 256ms/step - loss: 1.7185 - accuracy: 0.2144 - val_loss: 1.7350 - val_accuracy: 0.2148
Epoch 12/25
48/48 [=====] - 11s 234ms/step - loss: 1.7216 - accuracy: 0.2144 - val_loss: 1.6443 - val_accuracy: 0.2852
Epoch 13/25
48/48 [=====] - 11s 233ms/step - loss: 1.7170 - accuracy: 0.2190 - val_loss: 1.7369 - val_accuracy: 0.2111
Epoch 14/25
48/48 [=====] - 11s 234ms/step - loss: 1.6757 - accuracy: 0.2405 - val_loss: 1.6528 - val_accuracy: 0.2704
Epoch 15/25
48/48 [=====] - 11s 234ms/step - loss: 1.6941 - accuracy: 0.2275 - val_loss: 1.7259 - val_accuracy: 0.2148
Epoch 16/25
48/48 [=====] - 11s 233ms/step - loss: 1.6824 - accuracy: 0.2412 - val_loss: 1.6398 - val_accuracy: 0.2778
Epoch 17/25
48/48 [=====] - 11s 232ms/step - loss: 1.6593 - accuracy: 0.2542 - val_loss: 1.6196 - val_accuracy: 0.3296
Epoch 18/25
48/48 [=====] - 11s 234ms/step - loss: 1.6607 - accuracy: 0.2523 - val_loss: 1.6352 - val_accuracy: 0.2815
Epoch 19/25
48/48 [=====] - 11s 233ms/step - loss: 1.6493 - accuracy: 0.2601 - val_loss: 1.6820 - val_accuracy: 0.2296
Epoch 20/25
48/48 [=====] - 11s 235ms/step - loss: 1.6636 - accuracy: 0.2556 - val_loss: 1.6133 - val_accuracy: 0.2926
Epoch 21/25
48/48 [=====] - 11s 232ms/step - loss: 1.6629 - accuracy: 0.2529 - val_loss: 1.7419 - val_accuracy: 0.2000
Epoch 22/25
48/48 [=====] - 11s 233ms/step - loss: 1.6574 - accuracy: 0.2484 - val_loss: 1.7356 - val_accuracy: 0.2037
Epoch 23/25
48/48 [=====] - 11s 238ms/step - loss: 1.6923 - accuracy: 0.2366 - val_loss: 1.5944 - val_accuracy: 0.3037
Epoch 24/25
48/48 [=====] - 12s 244ms/step - loss: 1.6465 - accuracy: 0.2536 - val_loss: 1.6142 - val_accuracy: 0.2815
Epoch 25/25
48/48 [=====] - 12s 251ms/step - loss: 1.6356 - accuracy: 0.2614 - val_loss: 1.5791 - val_accuracy: 0.3296

```

## ۲-۵) ارزیابی

۱-۵-۲) رسم نمودارهای خطای آموزش و آزمون و تحلیل هر

کدام

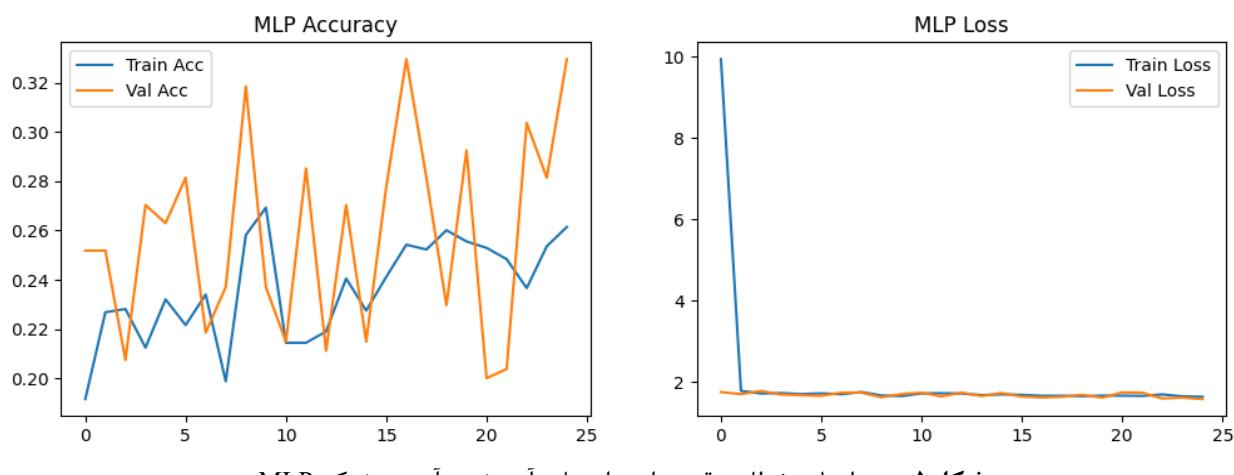
برای ارزیابی دو شبکه طراحی شده، نمودارهای دقت (Accuracy) و خطا (Loss) برای داده‌های آموزش و آزمون هر یک از شبکه‌ها رسم می‌شود. لازم به ذکر است پوشه Validation همان داده‌های آزمون در نظر گرفته شده است. کد مربوط به آن در زیر آورده شده است:

```
def plot_history(history, title):
    plt.figure(figsize=(12,4))
    plt.subplot(1,2,1)
    plt.plot(history.history['accuracy'], label='Train Acc')
    plt.plot(history.history['val_accuracy'], label='Val Acc')
    plt.title(f'{title} Accuracy')
    plt.legend()

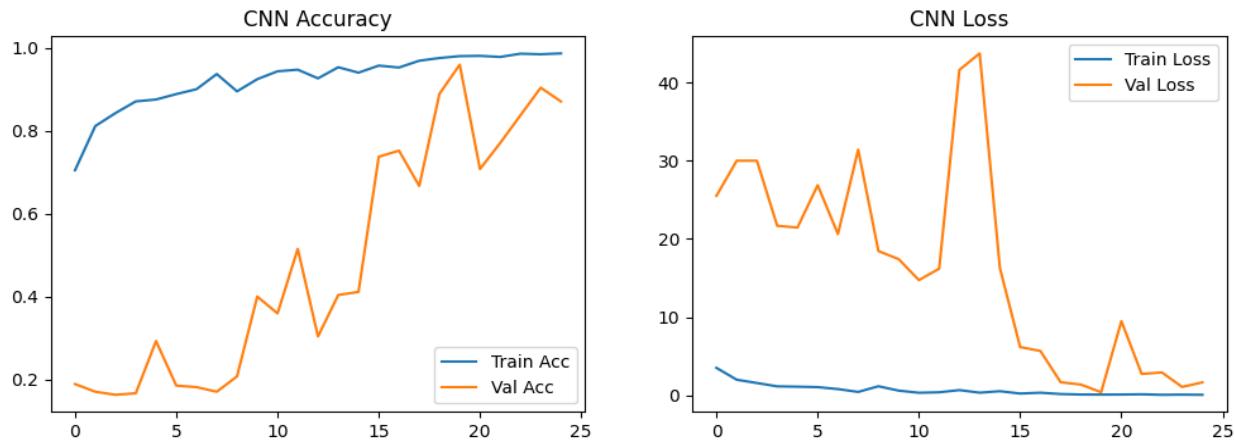
    plt.subplot(1,2,2)
    plt.plot(history.history['loss'], label='Train Loss')
    plt.plot(history.history['val_loss'], label='Val Loss')
    plt.title(f'{title} Loss')
    plt.legend()
    plt.show()

plot_history(history_cnn, "CNN")
plot_history(history_mlp, "MLP")
```

که پاسخ آن به صورت زیر است:



شکل ۵: نمودارهای خطا و دقت برای داده‌های آموزش و آزمون شبکه MLP



شکل ۶: نمودارهای خطا و دقت برای داده‌های آموزش و آزمون شبکه CNN

## ► تحلیل نمودارهای MLP

نمودار	آموزش و آزمون	تحلیل
نمودار خطا (Loss)	آموزش (train)	در ابتدای آموزش مقدار خطا بسیار بالا است (حدود ۱۰)، اما بلافاصله در همان چند دوره ابتدایی به حدود ۱.۸ کاهش می‌یابد. پس از کاهش ناگهانی، مقدار خطا تقریباً ثابت باقی مانده و تغییرات کمی دارد. این نشان می‌دهد که مدل به سرعت به یک حد پایین خطا رسیده و سپس دیگر بهبود قابل توجهی نداشته است.
نمودار دقت (Accuracy)	آزمون (test)	مقدار خطا برای داده‌های آزمون در بازه‌ای حدود ۱.۹ تا ۱.۷ قرار دارد و نسبتاً ثابت است. تغییرات خطا در آزمون نسبت به دقت آن کمتر است، که نشان‌دهنده‌ی این است که مدل از نظر تابع هزینه دچار تغییرات بزرگی نشده اما دقت ناپایدار است.
نمودار دقت (Accuracy)	آموزش (train)	دقت مدل در داده‌های آموزش بین حدود ۰.۲۰ تا ۰.۲۶ در نوسان است. با افزایش تعداد دوره‌ها، نوسانات کمی وجود دارد، ولی روند کلی تقریباً ثابت یا کمی افزایشی است. دقت آموزش پایین است که ممکن است به دلیل پیچیدگی کم مدل یا دشواری داده‌ها باشد.
نمودار دقت (Accuracy)	آزمون (test)	دقت آزمون نوسانات زیادی دارد و در برخی دوره‌ها تا ۰.۳۲ بالا می‌رود و در برخی دیگر تا زیر ۰.۲۰ کاهش می‌یابد. این نوسانات زیاد ممکن است نشان‌دهنده‌ی عدم پایداری مدل روی داده‌های آزمون باشد که ممکن است به دلیل بیش‌برازش (Overfitting) باشد.

## » تحلیل نمودارهای CNN

نمودار	آموزش و آزمون	تحلیل
	آموزش (train)	خطای آموزش از حدود ۵ شروع شده و به طور یکنواخت و تدریجی کاهش یافته تا به کمتر از ۰.۵ برسد. این کاهش تدریجی و یکنواخت، نشان‌دهنده‌ی یادگیری مؤثر و پایدار مدل در دوره‌های آموزش است.
نمودار خطا (Loss)	آزمون (test)	خطای آزمون در دوره‌های ابتدایی بین ۲۰ تا ۳۰ است و سپس در برخی نقاط (مثلًاً حوالی دوره ۱۳) تا بیش از ۴۰ افزایش یافته است. پس از آن، خطابه شدت کاهش یافته و به کمتر از ۵ رسیده است. این رفتار نوسانی ولی در نهایت نزولی نشان می‌دهد که مدل به مرور زمان توانسته تعمیم بهتری روی داده‌های آزمون پیدا کند، اما در برخی نقاط ممکن است دچار ناپایداری یا لحظه‌ای شده باشد.
نمودار دقت (Accuracy)	آموزش (train)	دقت آموزش از حدود ۰.۷ شروع شده و به سرعت تا نزدیک ۱.۰ افزایش یافته است. این روند نشان می‌دهد که مدل توانسته داده‌های آموزش را به خوبی یاد بگیرد. دقت بسیار بالا (نزدیک به ۱) نشانه‌ی یادگیری قوی مدل روی داده‌های آموزش است، اما همچنان، می‌تواند نشانه‌ای از بیش‌برازش (Overfitting) باشد.
	آزمون (test)	دقت آزمون در ابتدا بسیار پایین است (تقریباً ۰.۲)، اما به تدریج افزایش یافته و در نهایت تا حدود ۰.۹ رسیده است. با وجود نوساناتی در میانه مسیر، روند کلی صعودی و امیدوارکننده است. با این حال، نوسانات زیاد در بعضی نقاط (خصوصاً دوره ۲۰ ام) ممکن است به دلیل حساسیت مدل یا تنوع زیاد داده‌های آزمون باشد.

## ۲-۵-۲) دقت نهایی مدل روی داده‌های آزمون

برای تعیین دقت نهایی مدل روی داده‌های آزمون از کد زیر می‌شود:

```
loss_cnn, acc_cnn = cnn_model.evaluate(X_test, y_test, verbose=0)
print(f"CNN Test Accuracy: {acc_cnn * 100:.2f}%")

loss_mlp, acc_mlp = mlp_model.evaluate(X_test, y_test, verbose=0)
print(f"MLP Test Accuracy: {acc_mlp * 100:.2f}%")
```

که پاسخ آن به صورت زیر است:

CNN Test Accuracy: 87.04%

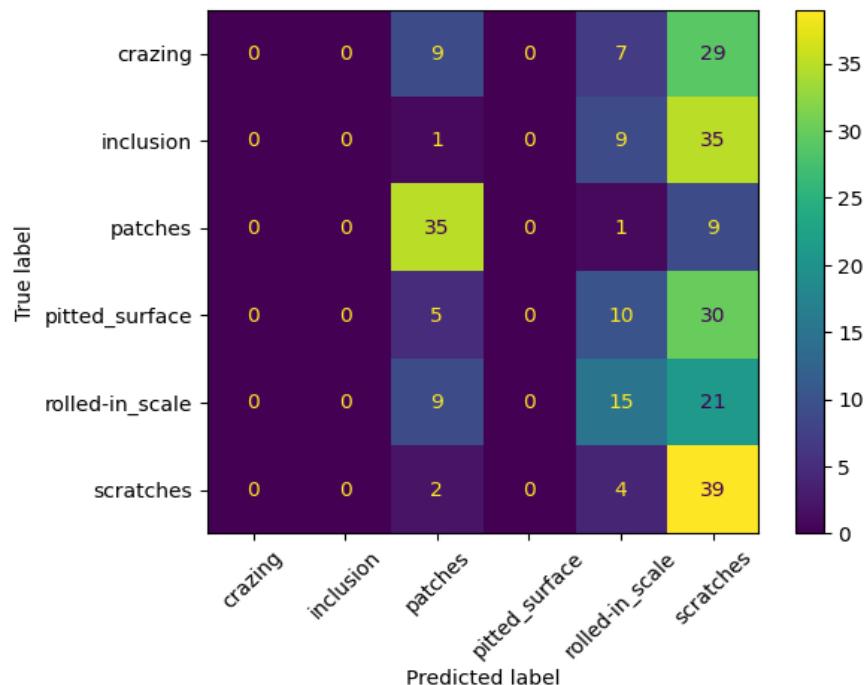
MLP Test Accuracy: 32.96%

### ۳-۵-۲) رسم ماتریس آشفتگی برای داده‌های آزمون و تفسیر آن

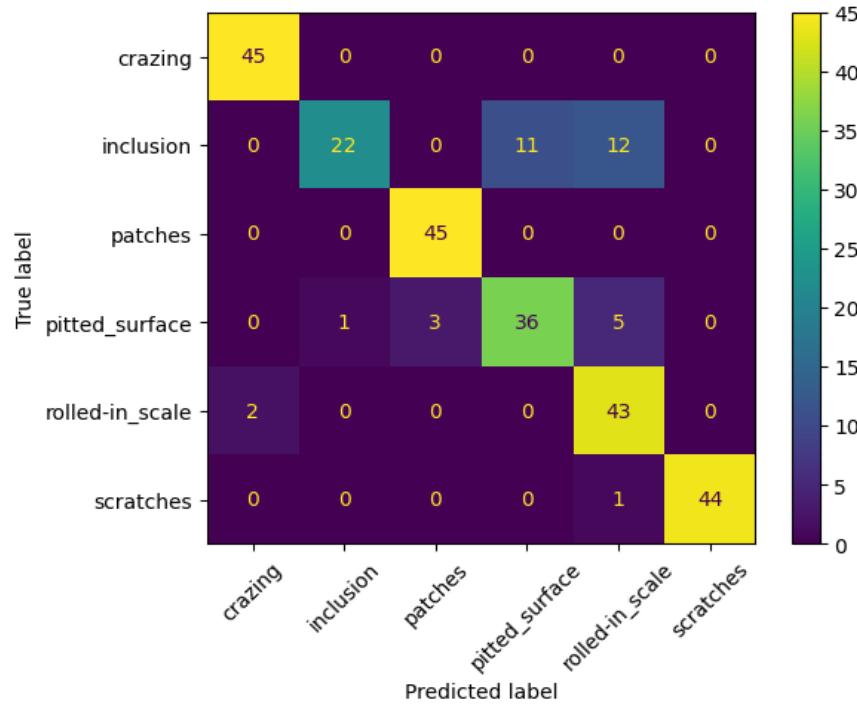
ماتریس‌های آشفتگی برای داده‌های آزمون شبکه‌های CNN و MLP با استفاده از کد زیر رسم می‌شود:

```
y_pred = np.argmax(mlp_model.predict(X_test), axis=1)
y_true = np.argmax(y_test, axis=1)
cm = confusion_matrix(y_true, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=le.classes_)
disp.plot(xticks_rotation=45)
plt.show()
```

```
y_pred = np.argmax(cnn_model.predict(X_test), axis=1)
y_true = np.argmax(y_test, axis=1)
cm = confusion_matrix(y_true, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=le.classes_)
disp.plot(xticks_rotation=45)
plt.show()
```



شکل ۷: ماتریس آشفتگی برای داده‌های آزمون شبکه MLP



شکل ۸: ماتریس آشفتگی برای داده‌های آزمون شبکه CNN

#### ➤ تفسیر نمودارهای ماتریس آشفتگی:

هر ماتریس آشفتگی یک شبکه  $6 \times 6$  برای ۶ کلاس فعالیت (با برجسب‌های ۰ تا ۵) است که لیبل و درصد طبقه‌بندی درست هر یک از کلاس‌ها در شبکه‌های MLP و CNN در جدول زیر ارائه شده است:

Description	Label	Class
MLP: 0% (0/45) CNN: 100% (45/45)	Crazing	0
MLP: 0% (0/45) CNN: 48.89% (22/45)	Inclusion	1
MLP: 77.78% (35/45) CNN: 100% (45/45)	Patches	2
MLP: 0% (0/45) CNN: 80% (36/45)	Pitted_Surface	3
MLP: 33.33% (15/45)	Rolled-in_Scale	4

CNN: 95.56% (43/45)		
MLP: 86.67% (39/45) CNN: 97.78% (44/45)	Scratches	5

**✓ تحلیل مدل MLP:**

مدل MLP در تشخیص بسیاری از کلاس‌ها عملکرد ضعیفی دارد، مخصوصاً کلاس‌های صفر، ۱ و ۳ (Crazing) که هیچ نمونه‌ای را درست تشخیص نداده است. جزئیات خطای مدل MLP در کلاس‌های دیگر نیز در ادامه به‌طور خلاصه توضیح داده شده است:

- » کلاس ۲ (Patches): ۱ مورد به اشتباه در کلاس ۴ (Rolled-in\_Scale) و ۹ مورد به اشتباه در کلاس ۵ (Scratches) طبقه‌بندی شده‌اند.
- » کلاس ۴ (Rolled-in\_Scale): ۹ مورد به اشتباه در کلاس ۲ (Patches) و ۲۱ مورد به اشتباه در کلاس ۵ (Scratches) طبقه‌بندی شده‌اند.

» کلاس ۵ (Scratches): ۲ مورد به اشتباه در کلاس ۲ (Patches) و ۴ مورد به اشتباه در کلاس ۴ (Rolled-in\_Scale) طبقه‌بندی شده‌اند.

بنابراین، در مدل MLP، کلاس‌ها اکثراً با Scratches یا کلاس‌های شبیه به آن اشتباه گرفته می‌شوند. دلیل اصلی این ضعف، ناتوانی MLP در درک ساختار مکانی تصاویر است.

**✓ تحلیل مدل CNN:**

مدل CNN به خوبی کلاس‌های صفر و ۲ (Crazing و Patches) را شناسایی کرده است. بیشترین اشتباه مدل CNN در تشخیص کلاس ۱ یعنی Inclusion مشاهده می‌شود که با کلاس‌های Pitted\_surface و Rolled-in\_scale اشتباه گرفته شده است. جزئیات خطای مدل CNN در کلاس‌های دیگر نیز در ادامه به‌طور خلاصه توضیح داده شده است:

- » کلاس ۱ (Inclusion): ۱۱ مورد به اشتباه در کلاس ۳ (Pitted\_Surface) و ۱۲ مورد به اشتباه در کلاس ۴ (Rolled-in\_Scale) طبقه‌بندی شده‌اند.

- » کلاس ۳ (Pitted\_Surface): ۱ مورد به اشتباه در کلاس ۱ (Inclusion)، ۳ مورد به اشتباه در کلاس ۲ و ۵ مورد به اشتباه در کلاس ۴ (Rolled-in\_Scale) طبقه‌بندی شده‌اند.
  - » کلاس ۴ (Rolled-in\_Scale): فقط ۲ مورد به اشتباه در کلاس صفر (Crazing) طبقه‌بندی شده است.
  - » کلاس ۵ (Scratches): فقط ۱ مورد به اشتباه در کلاس ۴ (Rolled-in\_Scale) طبقه‌بندی شده است.
- در مجموع، CNN نسبت به MLP پیش‌بینی خیلی بهتری دارد.

## ۶-۲) تغییر های پارامترها

### ۱-۶-۲) استفاده از لایه Dropout به جای CNN در شبکه

#### ✓ طراحی شبکه CNN-Block

جایگزینی لایه Dropout با SpatialDropout2D که می‌توان آن را به عنوان نوعی Block Dropout که کانولوشنی در نظر گرفت، در معماری شبکه‌های کانولوشنی (CNN) یک تصمیم برای بهبود توان تعمیم‌دهی کانولوشنی در این امکان را می‌دهد که مجموعه‌های از نودها را بتوان به صورت گروهی غیرفعال کرد، در حالی که در Dropout معمولی، نودها به صورت تصادفی غیرفعال می‌شوند. این کنترل می‌تواند باعث بهبود عملکرد مدل شود. کد این بخش مانند بخش ۳-۲ (طراحی شبکه CNN) است و فقط لایه SpatialDropout2D جایگزین لایه Dropout شده است. بنابراین، توضیحات مربوط به لایه‌های انتخاب شده و توابع فعال‌سازی در بخش ۳-۲ آورده است.

```

cnn_model_block = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(128, 128, 3)),
    BatchNormalization(),
    SpatialDropout2D(0.2),
    MaxPooling2D(),

    Conv2D(64, (3, 3), activation='relu'),
    BatchNormalization(),
    SpatialDropout2D(0.2),
    MaxPooling2D(),

    Flatten(),
    Dense(128, activation='relu'),
    Dense(6, activation='softmax')
])

cnn_model_block.compile(optimizer=tf.keras.optimizers.Adam(0.001),
                        loss='categorical_crossentropy',
                        metrics=['accuracy'])

cnn_model_block.summary()

```

که پاسخ کد فوق به صورت زیر است:

Model: "sequential\_3"

Layer (type)	Output Shape	Param #
<hr/>		
conv2d_5 (Conv2D)	(None, 126, 126, 32)	896
batch_normalization_5 (BatchNormalization)	(None, 126, 126, 32)	128
spatial_dropout2d_2 (SpatialDropout2D)	(None, 126, 126, 32)	0
max_pooling2d_5 (MaxPooling2D)	(None, 63, 63, 32)	0
conv2d_6 (Conv2D)	(None, 61, 61, 64)	18496
batch_normalization_6 (BatchNormalization)	(None, 61, 61, 64)	256
spatial_dropout2d_3 (SpatialDropout2D)	(None, 61, 61, 64)	0
max_pooling2d_6 (MaxPooling2D)	(None, 30, 30, 64)	0
flatten_3 (Flatten)	(None, 57600)	0
dense_7 (Dense)	(None, 128)	7372928

dense_8 (Dense)	(None, 6)	774
<hr/>		
=====		
Total params: 7393478 (28.20 MB)		
Trainable params: 7393286 (28.20 MB)		
Non-trainable params: 192 (768.00 Byte)		

---

## ✓ آموزش مدل CNN-Block

در ادامه، مانند بخش ۲-۴، برای آموزش مدل CNN-Block از روش fit استفاده می‌شود. داده‌های ورودی (x\_train) و برچسب‌ها (y\_train) به مدل داده می‌شوند و در ۲۵ دوره (epochs) آموزش می‌بیند. هم‌چنین، داده‌های تست (x\_test) و (y\_test) برای اعتبارسنجی مدل استفاده می‌شوند. سایز دسته‌ها نیز ۳۲×۳۲ تعیین شده است.

```
history_cnn_block = cnn_model_block.fit(X_train, y_train, epochs=25, validation_data=(X_test, y_test), batch_size=32)
```

که پاسخ آن به صورت زیر است:

```
Epoch 1/25
48/48 [=====] - 49s 988ms/step - loss: 14.2619 - accuracy: 0.5980 - val_loss: 13.8735 - val_accuracy: 0.259
3
Epoch 2/25
48/48 [=====] - 51s 1s/step - loss: 8.8100 - accuracy: 0.7425 - val_loss: 31.0508 - val_accuracy: 0.3037
Epoch 3/25
48/48 [=====] - 53s 1s/step - loss: 7.0073 - accuracy: 0.7608 - val_loss: 65.2325 - val_accuracy: 0.1667
Epoch 4/25
48/48 [=====] - 55s 1s/step - loss: 3.2370 - accuracy: 0.8346 - val_loss: 78.2128 - val_accuracy: 0.1667
Epoch 5/25
48/48 [=====] - 56s 1s/step - loss: 2.6915 - accuracy: 0.8601 - val_loss: 83.3628 - val_accuracy: 0.1667
Epoch 6/25
48/48 [=====] - 56s 1s/step - loss: 1.9673 - accuracy: 0.8693 - val_loss: 86.0365 - val_accuracy: 0.1667
Epoch 7/25
48/48 [=====] - 56s 1s/step - loss: 1.4849 - accuracy: 0.8706 - val_loss: 57.8384 - val_accuracy: 0.1963
Epoch 8/25
48/48 [=====] - 56s 1s/step - loss: 1.0379 - accuracy: 0.8954 - val_loss: 45.3706 - val_accuracy: 0.2444
Epoch 9/25
48/48 [=====] - 56s 1s/step - loss: 0.6206 - accuracy: 0.9020 - val_loss: 39.8339 - val_accuracy: 0.1963
Epoch 10/25
48/48 [=====] - 57s 1s/step - loss: 0.8259 - accuracy: 0.8993 - val_loss: 17.8146 - val_accuracy: 0.2852
Epoch 11/25
48/48 [=====] - 60s 1s/step - loss: 0.5413 - accuracy: 0.9176 - val_loss: 10.8158 - val_accuracy: 0.4704
Epoch 12/25
48/48 [=====] - 58s 1s/step - loss: 0.5013 - accuracy: 0.9216 - val_loss: 3.2734 - val_accuracy: 0.5815
Epoch 13/25
48/48 [=====] - 61s 1s/step - loss: 0.3920 - accuracy: 0.9386 - val_loss: 4.6258 - val_accuracy: 0.5815
Epoch 14/25
48/48 [=====] - 64s 1s/step - loss: 0.3741 - accuracy: 0.9366 - val_loss: 2.5183 - val_accuracy: 0.7333
```

```

Epoch 15/25
48/48 [=====] - 58s 1s/step - loss: 0.3632 - accuracy: 0.9359 - val_loss: 1.7963 - val_accuracy: 0.8407
Epoch 16/25
48/48 [=====] - 58s 1s/step - loss: 0.2816 - accuracy: 0.9484 - val_loss: 0.7020 - val_accuracy: 0.8815
Epoch 17/25
48/48 [=====] - 60s 1s/step - loss: 0.2707 - accuracy: 0.9471 - val_loss: 1.0803 - val_accuracy: 0.7778
Epoch 18/25
48/48 [=====] - 64s 1s/step - loss: 0.4137 - accuracy: 0.9366 - val_loss: 2.0305 - val_accuracy: 0.8148
Epoch 19/25
48/48 [=====] - 65s 1s/step - loss: 0.2037 - accuracy: 0.9529 - val_loss: 1.5130 - val_accuracy: 0.7185
Epoch 20/25
48/48 [=====] - 59s 1s/step - loss: 0.1513 - accuracy: 0.9660 - val_loss: 0.6662 - val_accuracy: 0.9037
Epoch 21/25
48/48 [=====] - 62s 1s/step - loss: 0.1233 - accuracy: 0.9712 - val_loss: 0.6683 - val_accuracy: 0.9111
Epoch 22/25
48/48 [=====] - 64s 1s/step - loss: 0.2261 - accuracy: 0.9575 - val_loss: 1.2103 - val_accuracy: 0.8519
Epoch 23/25
48/48 [=====] - 63s 1s/step - loss: 0.1389 - accuracy: 0.9725 - val_loss: 3.6855 - val_accuracy: 0.8259
Epoch 24/25
48/48 [=====] - 62s 1s/step - loss: 0.1770 - accuracy: 0.9712 - val_loss: 1.0799 - val_accuracy: 0.8667
Epoch 25/25
48/48 [=====] - 59s 1s/step - loss: 0.1399 - accuracy: 0.9706 - val_loss: 0.9773 - val_accuracy: 0.8630

```

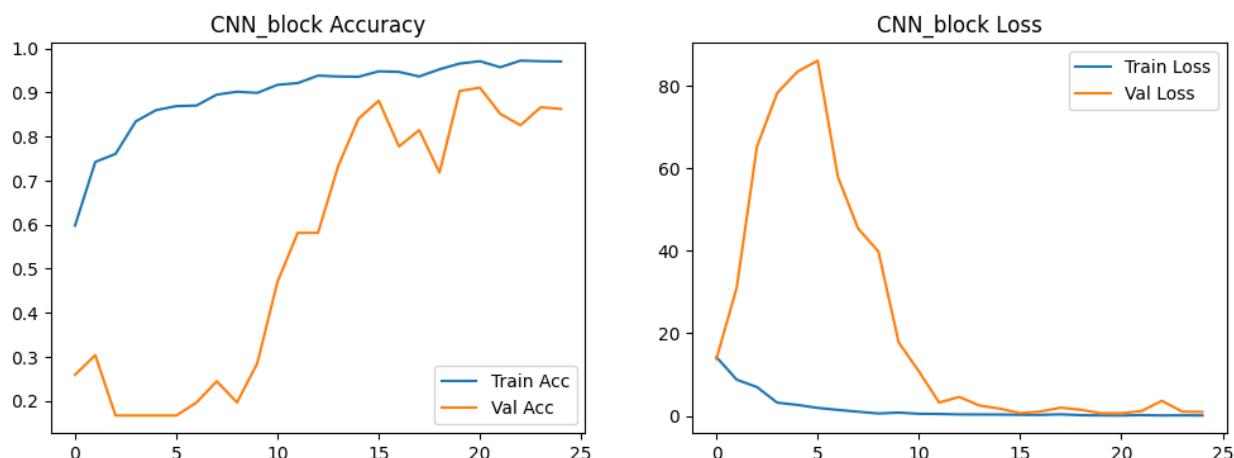
### ✓ ارزیابی شبکه CNN-Block

سپس، نمودارهای دقت (Accuracy) و خطا (Loss) برای داده‌های آموزش و آزمون مدل CNN-Block رسم

می‌شود. کد مربوط به آن در زیر آورده شده است:

```
plot_history(history_cnn_block, "CNN_block")
```

که پاسخ آن به صورت زیر است:



شکل ۹: نمودارهای خطا و دقت برای داده‌های آموزش و آزمون شبکه CNN-Block

## تحلیل نمودارهای CNN-Block

نمودار	آموزش و آزمون	تحلیل
	آموزش (train)	خطای آموزش از مقدار حدود ۱۵ شروع شده و به سرعت کاهش یافته، در میانه دوره‌ها به زیر ۱ رسیده و در ادامه بسیار کوچک شده است (نزدیک صفر). این کاهش سریع و پایدار نشان‌دهنده‌ی یادگیری مؤثر است.
نمودار خطا (Loss)	آزمون (test)	خطا در ابتدا روند افزایشی دارد (از ۱۵ تا بالای ۸۵) که غیرعادی است. سپس، از حدود دوره‌ی ششم به بعد، کاهش شدید آغاز می‌شود و به زیر ۵ و سپس نزدیک صفر می‌رسد. این رفتار می‌تواند به دلیل یادگیری تدریجی مدل در مراحل ابتدایی و نیاز به گرم شدن اولیه باشد. در انتهای، خطای بسیار پایین و پایدار می‌شود.
نمودار دقت (Accuracy)	آموزش (train)	دقت آموزش از حدود ۰.۶۰ شروع می‌شود و در طول ۲۵ دوره تا نزدیک ۰.۹۹ افزایش می‌یابد. روند افزایش یکنواخت و پیوسته است، بدون نوسانات شدید. این نشان می‌دهد که مدل به خوبی در حال یادگیری الگوهای داده‌های آموزش است.
	آزمون (test)	دقت آزمون در ابتدا پایین (حدود ۰.۲۵) است، اما از حدود دوره‌ی دهم به بعد به سرعت افزایش می‌یابد و به بیش از ۰.۹۰ می‌رسد. اگرچه نوساناتی کوچک در بین دوره‌ها دیده می‌شود (مثلاً افت در دوره ۱۸)، اما روند کلی بسیار خوب و صعودی است. مدل در آزمون نیز عملکرد بسیار خوبی دارد.

در مرحله بعد، دقت نهایی مدل روی داده‌های آزمون با استفاده از کد زیر تعیین می‌شود:

```
loss_cnn, acc_cnn = cnn_model_block.evaluate(X_test, y_test, verbose=0)
print(f"CNN_block Test Accuracy: {acc_cnn * 100:.2f}%")
```

که پاسخ آن به صورت زیر است:

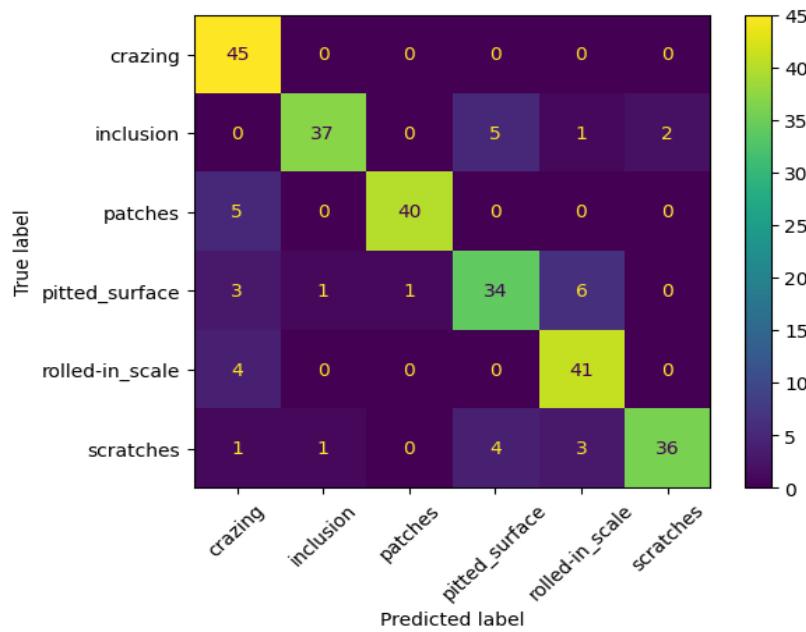
CNN\_block Test Accuracy: 86.30%

همانطور که در بخش ۲-۵ مشخص شد، دقت نهایی مدل CNN روی داده‌های آزمون ۴۷.۰٪ به دست آمده بود که از مدل CNN-Block حدود ۷۴٪ بیشتر است. بنابراین، دقت مدل CNN-Block نسبت به مدل CNN مقدار بسیار کوچکی کمتر شده است. دلایل کمتر شدن دقت در مدل CNN-Block نسبت به CNN را می‌توان در جدول زیر خلاصه کرد:

توضیح	عامل
یادگیری مدل تحت تأثیر تصادفی بودن اولیه‌سازی وزن‌ها، ترتیب داده‌ها و حذف‌های تصادفی در Dropout است. این تفاوت ممکن است صرفاً ناشی از این نوسانات باشد.	نوسانات تصادفی در آموزش
اگر تعداد داده‌ها نسبتاً کم باشد، تغییرات جزئی در تکنیک‌های regularization (مانند SpatialDropout یا Dropout) می‌توانند تأثیر متفاوتی بگذارند.	اندازه‌ی Dataset
در Dropout(0.۵) از مقدار ۰.۲. استفاده شده که ممکن است نسبت به Dropout2D در مدل قبلی، کمتر اثرگذار باشد.	Dropout مقدار
در مدل دوم، یک لایه Conv کمتر نسبت به مدل قبلی وجود دارد. مدل اول دارای سه لایه Conv بود ولی در این مدل فقط دو لایه Conv استفاده شده، که این می‌تواند به کاهش دقت منجر شود.	تعداد لایه‌های CNN

در نهایت، ماتریس‌های آشفتگی برای داده‌های آزمون مدل CNN-Block با استفاده از کد زیر رسم می‌شود:

```
y_pred = np.argmax(cnn_model_block.predict(X_test), axis=1)
y_true = np.argmax(y_test, axis=1)
cm = confusion_matrix(y_true, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=le.classes_)
disp.plot(xticks_rotation=45)
plt.show()
```



شکل ۱۰: ماتریس آشفتگی برای داده‌های آزمون مدل CNN-Block

**:CNN-Block تفسیر نمودار ماتریس آشفتگی مدل**

هر ماتریس آشفتگی یک شبکه  $6 \times 6$  برای ۶ کلاس فعالیت (با برجسبهای ۰ تا ۵) است که لیبل و درصد

طبقه‌بندی درست هر یک از کلاس‌ها در مدل CNN-Block در جدول زیر ارائه شده است:

Description	Label	Class
CNN-Block: 100% (45/45)	Crazing	۰
CNN-Block: 82.22% (37/45)	Inclusion	۱
CNN-Block: 88.89% (40/45)	Patches	۲
CNN-Block: 75.56% (34/45)	Pitted_Surface	۳
CNN-Block: 91.11% (41/45)	Rolled-in_Scale	۴
CNN-Block: 80% (36/45)	Scratches	۵

**:CNN-Block مدل تحلیل**

مدل CNN-Block در تشخیص بسیاری از کلاس‌ها عملکرد خوبی دارد، مخصوصاً کلاس صفر (Crazing) که همه‌ی نمونه‌ها را درست تشخیص داده است. جزئیات خطای مدل CNN-Block در کلاس‌های دیگر نیز در ادامه به طور خلاصه توضیح داده شده است:

- کلاس ۱ (Inclusion): ۵ مورد به اشتباه در کلاس ۳ (Pitted\_Surface)، ۱ مورد به اشتباه در کلاس ۴ (Scratches) و ۲ مورد به اشتباه در کلاس ۵ (Rolled-in\_Scale) طبقه‌بندی شده‌اند.
- کلاس ۲ (Patches): فقط ۵ مورد به اشتباه در کلاس صفر (Crazing) طبقه‌بندی شده است.
- کلاس ۳ (Pitted\_Surface): ۳ مورد به اشتباه در کلاس صفر (Crazing)، ۱ مورد به اشتباه در کلاس‌های ۱ (Inclusion) و ۲ (Rolled-in\_Scale) طبقه‌بندی شده‌اند.
- کلاس ۴ (Rolled-in\_Scale): فقط ۴ مورد به اشتباه در کلاس صفر (Crazing) طبقه‌بندی شده است.
- کلاس ۵ (Scratches): ۱ مورد به اشتباه در کلاس‌های صفر و ۱ (Inclusion) و ۴ مورد به اشتباه در کلاس ۳ (Pitted\_Surface) و ۳ مورد به اشتباه در کلاس ۴ (Rolled-in\_Scale) طبقه‌بندی شده‌اند.

بنابراین، مدل CNN-Block در تشخیص کلاس ۳ (Pitted\_Surface) کمترین دقت و در تشخیص کلاس صفر (Crazing) بیشترین دقت را داشته است.

## ۲-۶-۲ توضیح و پیاده‌سازی تجزیه فیلترها (Kernel Factorization) در شبکه CNN

### ✓ طراحی شبکه CNN-Factorized

تجزیه فیلترها (Factorization Kernel) یک تکنیک در یادگیری ماشین و به ویژه در زمینه یادگیری عمیق و شبکه‌های عصبی است که به منظور کاهش پیچیدگی محاسباتی و بهبود کارایی مدل‌ها مورد استفاده قرار می‌گیرد. این مفهوم به ویژه در زمینه‌هایی مانند پردازش تصویر و تحلیل داده‌های چندبعدی کاربرد دارد. در واقع، تجزیه فیلترها به معنای تقسیم یک فیلتر یا کرنل بزرگ به چند فیلتر یا کرنل کوچکتر است. این کار می‌تواند به کاهش تعداد پارامترها و همچنین کاهش محاسبات لازم برای آموزش و پیش‌بینی کمک کند. در نتیجه، هدف اصلی این است که یک فیلتر پیچیده را به چند فیلتر ساده‌تر تقسیم کنیم که به صورت مستقل عمل کنند. کد این بخش مانند بخش ۳-۲ (طراحی شبکه CNN) است، فقط با فاکتور کردن کانولوشن‌های دو بعدی  $3 \times 3$  به چند کانولوشن دو بعدی ساده‌تر (مثل  $1 \times 1$  و  $3 \times 3$ )، هم تعداد پارامتر کاهش می‌یابد و هم از overfitting جلوگیری می‌شود. بنابراین، توضیحات مربوط به لایه‌های انتخاب شده و توابع فعال‌سازی در بخش ۳-۲ آورده است.

```

cnn_factorized = Sequential([
    Conv2D(32, (1, 3), activation='relu', input_shape=(128, 128, 3)),
    Conv2D(32, (3, 1), activation='relu'),
    BatchNormalization(),
    MaxPooling2D(2, 2),

    Conv2D(64, (1, 3), activation='relu'),
    Conv2D(64, (3, 1), activation='relu'),
    BatchNormalization(),
    MaxPooling2D(2, 2),

    Conv2D(128, (1, 3), activation='relu'),
    Conv2D(128, (3, 1), activation='relu'),
    BatchNormalization(),
    MaxPooling2D(2, 2),

    Flatten(),
    Dropout(0.5),
    Dense(128, activation='relu'),
    Dense(6, activation='softmax')
])

cnn_factorized.compile(optimizer=tf.keras.optimizers.Adam(0.001),
                       loss='categorical_crossentropy',
                       metrics=['accuracy'])

cnn_factorized.summary()

```

که پاسخ کد فوق به صورت زیر است:

Model: "sequential\_4"

Layer (type)	Output Shape	Param #
conv2d_7 (Conv2D)	(None, 128, 126, 32)	320
conv2d_8 (Conv2D)	(None, 126, 126, 32)	3104
batch_normalization_7 (BatchNormalization)	(None, 126, 126, 32)	128
max_pooling2d_7 (MaxPooling2D)	(None, 63, 63, 32)	0
conv2d_9 (Conv2D)	(None, 63, 61, 64)	6208
conv2d_10 (Conv2D)	(None, 61, 61, 64)	12352
batch_normalization_8 (BatchNormalization)	(None, 61, 61, 64)	256
max_pooling2d_8 (MaxPooling2D)	(None, 30, 30, 64)	0
conv2d_11 (Conv2D)	(None, 30, 28, 128)	24704
conv2d_12 (Conv2D)	(None, 28, 28, 128)	49280

```

batch_normalization_9 (BatchNormalization)      512
max_pooling2d_9 (MaxPooling2D)                0
flatten_4 (Flatten)                          0
dropout_2 (Dropout)                         0
dense_9 (Dense)                            3211392
dense_10 (Dense)                           774
=====
Total params: 3309030 (12.62 MB)
Trainable params: 3308582 (12.62 MB)
Non-trainable params: 448 (1.75 KB)

```

### ✓ آموزش مدل CNN-Factorized

در ادامه، مانند بخش ۲-۴، برای آموزش مدل CNN-Factorized از روش fit استفاده می‌شود. داده‌های ورودی (x\_train) و برچسب‌ها (y\_train) به مدل داده می‌شوند و در ۲۵ دوره (epochs) آموزش می‌بینند. همچنین، داده‌های تست (x\_test) و (y\_test) برای اعتبارسنجی مدل استفاده می‌شوند. سایز دسته‌ها نیز ۳۲ تعیین شده است.

```
history_cnn_factorized = cnn_factorized.fit(X_train, y_train, epochs=25, validation_data=(X_test, y_test), batch_size=32)
```

که پاسخ آن به صورت زیر است:

```

Epoch 1/25
48/48 [=====] - 80s 2s/step - loss: 2.9517 - accuracy: 0.7235 - val_loss: 16.3325 - val_accuracy: 0.1667
Epoch 2/25
48/48 [=====] - 83s 2s/step - loss: 0.8851 - accuracy: 0.8699 - val_loss: 16.1606 - val_accuracy: 0.1704
Epoch 3/25
48/48 [=====] - 85s 2s/step - loss: 0.5979 - accuracy: 0.9150 - val_loss: 27.7588 - val_accuracy: 0.1667
Epoch 4/25
48/48 [=====] - 85s 2s/step - loss: 0.7967 - accuracy: 0.8850 - val_loss: 18.2684 - val_accuracy: 0.1519
Epoch 5/25
48/48 [=====] - 86s 2s/step - loss: 0.4582 - accuracy: 0.9078 - val_loss: 21.5110 - val_accuracy: 0.2741
Epoch 6/25
48/48 [=====] - 85s 2s/step - loss: 0.4368 - accuracy: 0.9242 - val_loss: 21.2934 - val_accuracy: 0.2259
Epoch 7/25
48/48 [=====] - 88s 2s/step - loss: 0.5044 - accuracy: 0.9144 - val_loss: 11.2251 - val_accuracy: 0.2148
Epoch 8/25
48/48 [=====] - 85s 2s/step - loss: 0.3430 - accuracy: 0.9392 - val_loss: 14.3955 - val_accuracy: 0.2148
Epoch 9/25
48/48 [=====] - 85s 2s/step - loss: 0.4989 - accuracy: 0.9209 - val_loss: 17.5689 - val_accuracy: 0.2222
Epoch 10/25
48/48 [=====] - 85s 2s/step - loss: 0.3050 - accuracy: 0.9320 - val_loss: 17.0085 - val_accuracy: 0.2630
Epoch 11/25
48/48 [=====] - 87s 2s/step - loss: 0.3896 - accuracy: 0.9275 - val_loss: 4.7170 - val_accuracy: 0.5963

```

```

Epoch 11/25
48/48 [=====] - 87s 2s/step - loss: 0.3896 - accuracy: 0.9275 - val_loss: 4.7170 - val_accuracy: 0.5963
Epoch 12/25
48/48 [=====] - 97s 2s/step - loss: 0.1986 - accuracy: 0.9438 - val_loss: 3.6496 - val_accuracy: 0.6593
Epoch 13/25
48/48 [=====] - 87s 2s/step - loss: 0.1114 - accuracy: 0.9693 - val_loss: 2.9057 - val_accuracy: 0.6556
Epoch 14/25
48/48 [=====] - 97s 2s/step - loss: 0.1473 - accuracy: 0.9654 - val_loss: 1.0137 - val_accuracy: 0.7704
Epoch 15/25
48/48 [=====] - 88s 2s/step - loss: 0.2353 - accuracy: 0.9471 - val_loss: 6.1576 - val_accuracy: 0.7889
Epoch 16/25
48/48 [=====] - 85s 2s/step - loss: 0.1303 - accuracy: 0.9582 - val_loss: 4.4636 - val_accuracy: 0.6037
Epoch 17/25
48/48 [=====] - 85s 2s/step - loss: 0.2362 - accuracy: 0.9523 - val_loss: 32.3357 - val_accuracy: 0.2148
Epoch 18/25
48/48 [=====] - 85s 2s/step - loss: 0.1344 - accuracy: 0.9595 - val_loss: 1.1819 - val_accuracy: 0.7741
Epoch 19/25
48/48 [=====] - 85s 2s/step - loss: 0.0811 - accuracy: 0.9654 - val_loss: 4.7287 - val_accuracy: 0.5370
Epoch 20/25
48/48 [=====] - 94s 2s/step - loss: 0.0731 - accuracy: 0.9752 - val_loss: 3.2334 - val_accuracy: 0.6407
Epoch 21/25
48/48 [=====] - 90s 2s/step - loss: 0.0671 - accuracy: 0.9765 - val_loss: 0.3938 - val_accuracy: 0.9259
Epoch 22/25
48/48 [=====] - 83s 2s/step - loss: 0.0980 - accuracy: 0.9804 - val_loss: 0.5315 - val_accuracy: 0.9111
Epoch 23/25
48/48 [=====] - 82s 2s/step - loss: 0.0310 - accuracy: 0.9882 - val_loss: 0.2696 - val_accuracy: 0.9593
Epoch 24/25
48/48 [=====] - 83s 2s/step - loss: 0.0435 - accuracy: 0.9869 - val_loss: 0.1185 - val_accuracy: 0.9704
Epoch 25/25
48/48 [=====] - 85s 2s/step - loss: 0.0563 - accuracy: 0.9843 - val_loss: 0.4350 - val_accuracy: 0.8926

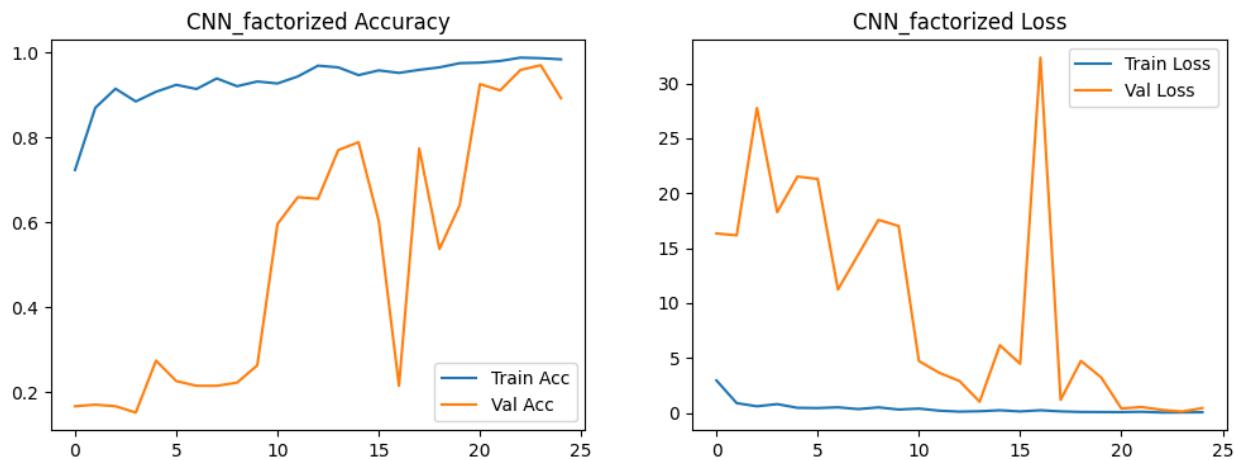
```

## ✓ ارزیابی شبکه CNN-Factorized

سپس، نمودارهای دقت (Accuracy) و خطا (Loss) برای داده‌های آموزش و آزمون مدل CNN-Factorized رسم می‌شود. کد مربوط به آن در زیر آورده شده است:

```
plot_history(history_cnn_factorized, "CNN_factorized")
```

که پاسخ آن به صورت زیر است:



شکل ۱۱: نمودارهای خطا و دقت برای داده‌های آموزش و آزمون شبکه CNN-Factorized

## تحلیل نمودارهای CNN-Factorized

نمودار	آموزش و آزمون	تحلیل
	آموزش (train)	خطای آموزش از حدود ۳ شروع شده و به سرعت به مقادیر نزدیک صفر کاهش یافته است. کاهش خطابسیار پایدار است و هیچ افزایش یا نوسان خاصی ندارد.
نمودار خطاب (Loss)	آزمون (test)	خطا از مقدار ۱۶ شروع شده و با نوسانات شدید به بیش از ۳۰ نیز می‌رسد. با وجود کاهش کلی تا حدود ۰.۵، چند پیک بزرگ وجود دارد (مثلاً دوره ۱۶). این رفتار ناپایدار نشان می‌دهد که مدل با وجود یادگیری خوب، در برخی نقاط در تعیین‌دهی به داده‌های آزمون مشکل دارد.
	آموزش (train)	دقت از حدود ۰.۷ آغاز شده و به مرور به حدود ۰.۹۸ رسیده است. روند صعودی یکنواخت و بدون نوسان زیاد است. این نشان‌دهنده‌ی یادگیری خوب مدل بر روی داده‌های آموزش است.
نمودار دقیقت (Accuracy)	آزمون (test)	دقت در ابتدا بسیار پایین (قریباً ۰.۱۵)، سپس با نوسانات شدید مواجه می‌شود. از دوره دهم به بعد به حدود ۰.۹۵ می‌رسد، اما در برخی دوره‌ها افت ناگهانی دیده می‌شود (مثلاً دوره ۱۶ تا زیر ۰.۳). این نوسانات می‌توانند نشان‌دهنده‌ی حساسیت مدل به داده‌های خاص یا batch‌های نامتوافق باشد.

در مرحله بعد، دقیقت نهایی مدل بر روی آزمون با استفاده از کد زیر تعیین می‌شود:

```
loss_cnn, acc_cnn = cnn_factorized.evaluate(X_test, y_test, verbose=0)
print(f"CNN_factorized Test Accuracy: {acc_cnn * 100:.2f}%")
```

که پاسخ آن به صورت زیر است:

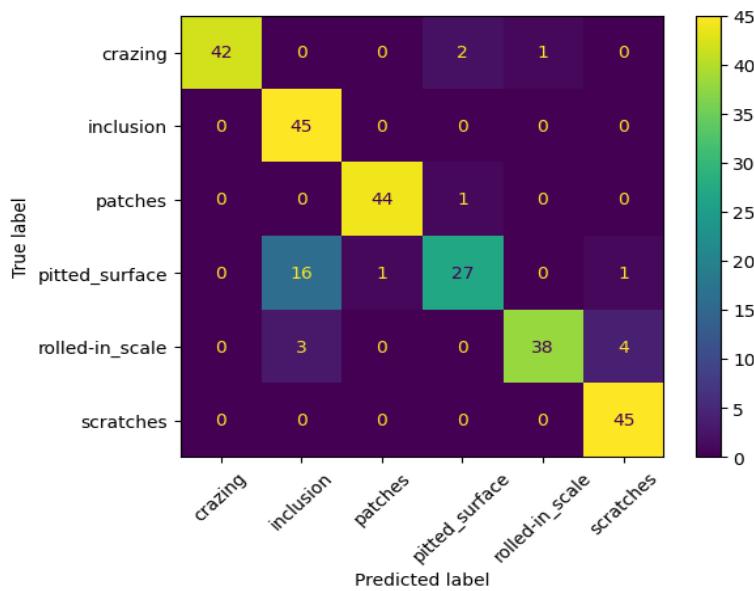
CNN\_factorized Test Accuracy: 89.26%

همانطور که در بخش‌های ۲-۵-۲ و ۱-۶-۲ مشخص شد، دقیقت نهایی مدل‌های CNN و CNN-Block روی داده‌های آزمون به ترتیب، ۸۷.۰۴٪ و ۸۶.۳۰٪ بدست آمده بود که از مدل CNN-Factorized به ترتیب حدود ۲.۲۲٪ و ۲.۹۶٪ کمتر است. بنابراین، دقیقت مدل CNN-Factorized نسبت به مدل‌های CNN و CNN-Block مقدار قابل توجهی بیشتر شده است دلایل بیشتر شدن دقیقت در مدل CNN-Factorized نسبت به مدل‌های CNN-Block را می‌توان در جدول زیر خلاصه کرد:

توضیح	عامل
با فاکتور کردن کانولوشن های استاندارد (مثلًا $3 \times 3$ ) به چند کانولوشن ساده‌تر (مثل $1 \times 1$ و $3 \times 1$ ، هم تعداد پارامتر کاهش می‌باید، هم از overfitting جلوگیری می‌شود.	کاهش پیچیدگی مدل
این نوع کانولوشن به مدل اجازه می‌دهد ویژگی‌های پیچیده‌تر با پارامترهای کمتر را بیاموزد، که باعث بهبود تعیین‌دهی روی داده‌های جدید می‌شود.	افزایش کارایی یادگیری ویژگی‌ها
فاکتورایز کردن باعث می‌شود مدل ساختار فضایی تصویر را با دقت بیشتری حفظ کند، در حالی که Dropout به صورت تصادفی بعضی از ویژگی‌ها را حذف می‌کند.	حفظ بهتر ساختار مکانی تصویر

در نهایت، ماتریس‌های آشفتگی برای داده‌های آزمون مدل CNN-Factorized با استفاده از کد زیر رسم می‌شود:

```
y_pred = np.argmax(cnn_factorized.predict(X_test), axis=1)
y_true = np.argmax(y_test, axis=1)
cm = confusion_matrix(y_true, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=le.classes_)
disp.plot(xticks_rotation=45)
plt.show()
```



شکل ۱۲: ماتریس آشفتگی برای داده‌های آزمون مدل CNN-Factorized

تفسیر نمودار ماتریس آشفتگی مدل :CNN-Factorized

هر ماتریس آشفتگی یک شبکه  $6 \times 6$  برای ۶ کلاس فعالیت (با برجسب‌های ۰ تا ۵) است که لیبل و درصد طبقه‌بندی درست هر یک از کلاس‌ها در مدل CNN-Factorized در جدول زیر ارائه شده است:

Description	Label	Class
CNN-Block: 93.33% (42/45)	Crazing	۰
CNN-Block: 100% (45/45)	Inclusion	۱
CNN-Block: 97.78% (44/45)	Patches	۲
CNN-Block: 60% (27/45)	Pitted_Surface	۳
CNN-Block: 84.44% (38/45)	Rolled-in_Scale	۴
CNN-Block: 100% (45/45)	Scratches	۵

### تحلیل مدل CNN-Factorized

مدل CNN-Factorized هم مانند مدل CNN-Block در تشخیص بسیاری از کلاس‌ها عملکرد خوبی دارد، مخصوصاً کلاس ۱ و ۵ (Inclusion و Scratches) که همهی نمونه‌ها را درست تشخیص داده است. جزئیات خطای مدل CNN-Factorized در کلاس‌های دیگر نیز در ادامه به‌طور خلاصه توضیح داده شده است:

► کلاس صفر (Crazing): ۲ مورد به اشتباه در کلاس ۳ (Pitted\_Surface) و ۱ مورد به اشتباه در کلاس ۴ (Rolled-in\_Scale) طبقه‌بندی شده‌اند.

► کلاس ۲ (Patches): فقط ۱ مورد به اشتباه در کلاس ۳ (Pitted\_Surface) طبقه‌بندی شده است.

► کلاس ۳ (Pitted\_Surface): ۱۶ مورد به اشتباه در کلاس ۱ (Crazing)، ۱ مورد به اشتباه در کلاس ۲ (Patches) و ۱ مورد به اشتباه در کلاس ۵ (Scratches) طبقه‌بندی شده‌اند.

► کلاس ۴ (Rolled-in\_Scale): ۳ مورد به اشتباه در کلاس ۱ (Crazing) و ۴ مورد به اشتباه در کلاس ۵ (Scratches) طبقه‌بندی شده‌اند.

بنابراین، مدل CNN-Factorized مانند مدل CNN-Block فقط در تشخیص کلاس ۳ (Pitted\_Surface) دقیقی داشته و در تشخیص بقیه کلاس‌ها دقیقی بالایی داشته است.

## ۷-۲) تحلیل

به منظور مقایسه بهتر و دقیق‌تر بین سه شبکه CNN-Factorized، CNN-Block و CNN-Block، خلاصه نتایج به دست آمده در جدول زیر آورده شده است:

CNN-Factorized	CNN-Block	CNN	معیار
۰.۹۹	۰.۹۹	۰.۹۹	Train Accuracy
۰.۹۵	۰.۹۲	۰.۹	Test Accuracy
تقریباً صفر	تقریباً صفر	۰.۰۵	Train Loss
تقریباً صفر	تقریباً صفر	۰.۱	Test Loss
پایدار	بسیار پایدار	پایدار	Train Stability
ناپایدار	نسبتاً پایدار	ناپایدار	Test Stability
% ۸۹.۲۶	% ۸۶.۳۰	% ۸۷.۰۴	Final Test Accuracy
متوسط	کم	متوسط به بالا	Overfitting Risk

:CNN ✓

- گرچه دقت آموزش بالاست، ولی در آزمون نوسانات زیادی دارد.
- ممکن است به دلیل سادگی مدل در برابر داده‌های جدید، تعمیم‌پذیری کمی داشته باشد.
- احتمال overfitting یا حساسیت به batch ها وجود دارد.

:CNN-Block ✓

- بهترین تعادل بین دقت و پایداری را دارد.
- خطا و دقت در هر دو بخش آموزش و آزمون، پایدار و نزدیک به هم هستند.
- مناسب‌ترین گزینه برای مدل نهایی است.

**:CNN-Factorized ✓**

- دقت نهایی آزمون بالا ولی بسیار نایاب است.
- وجود پیکهای بزرگ در خطای آزمون نشان‌دهندهٔ ضعف در generalization یا وجود نویز/عدم تعادل در داده‌های آزمون است.

مدل CNN با کانولوشن‌های فاکتورایز شده (CNN\_Factorized) با دقت تست ۸۹.۲۶٪ صرفاً عملکرد دقت بهتری نسبت به دو مدل CNN و CNN-Block داشته است. علت‌های برتری دقت مدل CNN\_Factorized نسبت به دو مدل دیگر در جدول زیر ارائه شده است:

توضیح	عامل
برخلاف MLP که تصاویر را به بردار تخت تبدیل می‌کند، CNN ویژگی‌های مکانی (مانند لبه‌ها، بافت‌ها و اشکال) را حفظ می‌کند.	حفظ ساختار فضایی تصویر
با فاکتور کردن کانولوشن‌ها (مثالاً $3 \times 3$ به $3 \times 1$ و $1 \times 3$ )، مدل قادر به استخراج ویژگی‌های پیچیده با پارامتر کمتر است، که باعث تعمیم بهتر روی داده‌های تست می‌شود.	یادگیری ویژگی‌های دقیق‌تر
فاکتورایز کردن منجر به کاهش پارامترها و افزایش سرعت آموزش و کاهش overfitting می‌شود، بدون کاهش دقت.	کاهش پارامتر و افزایش کارایی
برخلاف SpatialDropout2D که ویژگی‌های تصادفی را حذف می‌کند، فاکتورایز کردن بدون حذف اطلاعات، ویژگی‌ها را فشرده‌تر و دقیق‌تر یاد می‌گیرد.	پایداری بیشتر نسبت به Dropout

در نتیجه، مدل CNN-Block، پایدارترین و قابل اعتمادترین عملکرد را بین سه مدل دارد. بنابراین، اگر هدف اصلی، پایداری و تعمیم‌پذیری باشد، مدل CNN-Block بهترین گزینه است. اگر فقط دقت نهایی مهم باشد، CNN-Factorized برتری دارد.

### (۳) بخش سوم: یادگیری انتقالی (Transfer Learning)

#### ➤ معرفی کتابخانه‌های استفاده شده

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import matplotlib.pyplot as plt
import numpy as np
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.layers import GlobalAveragePooling2D, Dense
from tensorflow.keras.models import Model
import tensorflow as tf
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
```

بعضی از کتابخانه‌های استفاده شده در این بخش، در بخش‌های قبل (NEU Surface Defects و UCI HAR)

نیز به کار رفته‌اند که توضیحات آنها دیگر اینجا داده نمی‌شود و فقط کتابخانه‌های جدید معرفی خواهند شد:

- ImageDataGenerator برای افزایش تنوع و بهبود کیفیت داده‌های تصویری مانند چرخش، تغییر مقیاس و برش. این کار موجب افزایش دقت مدل می‌شود.
- ResNet50 یک مدل پیش‌آموزش داده شده با ۵۰ لایه برای شناسایی تصاویر طراحی شده است و از باقی‌مانده‌ها استفاده می‌کند. این مدل به عنوان پایه برای شناسایی ویژگی‌های تصاویر کاربرد دارد.
- Fully Connected کاهش ابعاد و جمع‌بندی ویژگی‌ها قبل از لایه‌های GlobalAveragePooling2D

### (۱) آماده‌سازی داده‌ها

به منظور آماده‌سازی داده‌ها، ابتدا یک شیء از کلاس ImageDataGenerator برای پیش‌پردازش تصاویر ایجاد می‌شود. برای اعمال داده‌افرازی (Data Augmentation)، این شیء به پارامترهایی مانند مقیاس‌بندی پیکسل‌ها، تقسیم داده‌ها به دو بخش آموزشی و آزمون، چرخش، بزرگنمایی، معکوس کردن افقی و تغییرات موقعیت افقی و عمودی تصاویر تنظیم شده است. سپس با استفاده از روش flow\_from\_directory، داده‌های آموزش و آزمون از دایرکتوری‌های مشخص بارگذاری می‌شوند. نام کلاس‌ها از train\_generator استخراج شده و یک دسته از داده‌های آموزشی به همراه برچسب‌هایشان گرفته می‌شود. در نهایت، با استفاده از matplotlib، تصاویری از این دسته نمایش داده می‌شود. کد مربوط به این بخش در زیر آورده شده است:

```

datagen = ImageDataGenerator(
    rescale=1./255,
    validation_split=0.15,
    rotation_range=20,
    zoom_range=0.2,
    horizontal_flip=True,
    width_shift_range=0.1,
    height_shift_range=0.1
)

train_generator = datagen.flow_from_directory(
    'G:\Artificial_Intelligence\Tamrin\Tamrin4\HW4\NEU-DET\train\images',
    target_size=(224, 224),
    batch_size=32,
    class_mode='categorical',
    subset='training',
    shuffle=True
)

val_generator = datagen.flow_from_directory(
    'G:\Artificial_Intelligence\Tamrin\Tamrin4\HW4\NEU-DET\validation\images',
    target_size=(224, 224),
    batch_size=32,
    class_mode='categorical',
    subset='validation',
    shuffle=True
)

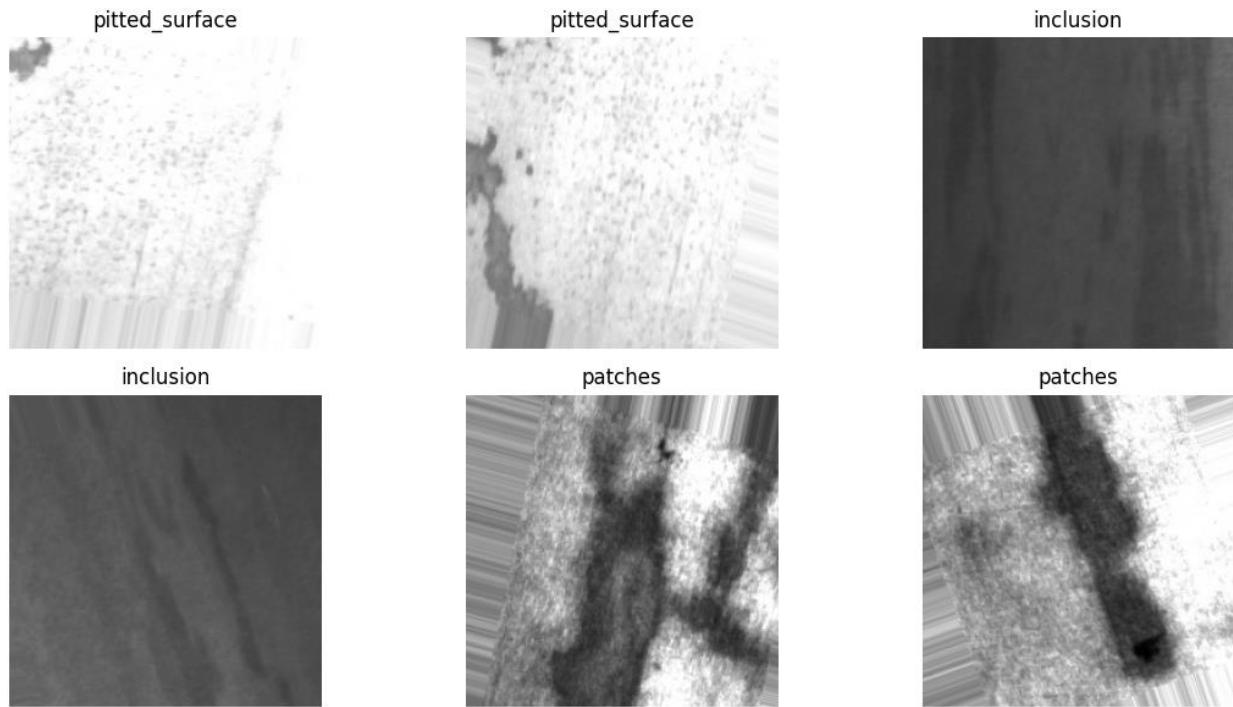
class_names = list(train_generator.class_indices.keys())
x_batch, y_batch = next(train_generator)

plt.figure(figsize=(12,6))
for i in range(6):
    plt.subplot(2,3,i+1)
    plt.imshow(x_batch[i])
    plt.title(class_names[np.argmax(y_batch[i])])
    plt.axis('off')
plt.tight_layout()
plt.show()

```

پاسخ کد فوق به صورت زیر است:

Found 1224 images belonging to 6 classes.  
 Found 216 images belonging to 6 classes.



### ۲-۳ آماده‌سازی مدل

ابتدا، یک مدل پایه به نام ResNet50 با وزن‌های از پیش آموزش دیده روی دیتاست ImageNet بارگذاری می‌شود که در آن بخش‌های انتهایی مدل (top) را شامل نمی‌شود. سپس، همه لایه‌های این مدل به حالت غیرقابل آموزش درآمده تا خصوصیات از پیش یادگرفته شده حفظ شود.

در مرحله بعد، خروجی مدل پایه به لایه‌ای میانگین‌گیری کلی (Global Average Pooling) متصل می‌شود که ابعادش را کاهش می‌دهد. سپس، لایه Dense با ۱۲۸ نورون و تابع فعال‌سازی ReLU به این خروجی اضافه می‌شود. در نهایت، خروجی به لایه دیگری با ۶ نورون و تابع فعال‌سازی Softmax متصل می‌شود که پیش‌بینی نهایی دسته‌بندی را انجام می‌دهد.

مدل نهایی با ورودی مدل پایه و پیش‌بینی‌ها تعریف و کامپایل می‌شود. در اینجا، از بهینه‌ساز Adam و تابع categorical\_crossentropy استفاده می‌شود. در نهایت، خلاصه‌ای از ساختار مدل نمایش داده می‌شود. کد مربوط به این بخش در ادامه ارائه شده است:

```

base_model = ResNet50(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
for layer in base_model.layers:
    layer.trainable = False

x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(128, activation='relu')(x)
predictions = Dense(6, activation='softmax')(x)

model_resnet = Model(inputs=base_model.input, outputs=predictions)
model_resnet.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
model_resnet.summary()

```

پاسخ کد فوق به صورت زیر است:

Model: "model"

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, 224, 224, 3)]	0	[]
conv1_pad (ZeroPadding2D)	(None, 230, 230, 3)	0	['input_1[0][0]']
conv1_conv (Conv2D)	(None, 112, 112, 64)	9472	['conv1_pad[0][0]']
conv1_bn (BatchNormalizati on)	(None, 112, 112, 64)	256	['conv1_conv[0][0]']
conv1_relu (Activation)	(None, 112, 112, 64)	0	['conv1_bn[0][0]']
pool1_pad (ZeroPadding2D)	(None, 114, 114, 64)	0	['conv1_relu[0][0]']
pool1_pool (MaxPooling2D)	(None, 56, 56, 64)	0	['pool1_pad[0][0]']
conv2_block1_1_conv (Conv2 D)	(None, 56, 56, 64)	4160	['pool1_pool[0][0]']
conv2_block1_1_bn (BatchNo rmalization)	(None, 56, 56, 64)	256	['conv2_block1_1_conv[0][0]']
conv2_block1_1_relu (Activ ation)	(None, 56, 56, 64)	0	['conv2_block1_1_bn[0][0]']
conv2_block1_2_conv (Conv2 D)	(None, 56, 56, 64)	36928	['conv2_block1_1_relu[0][0]']
conv2_block1_2_bn (BatchNo rmalization)	(None, 56, 56, 64)	256	['conv2_block1_2_conv[0][0]']
conv2_block1_2_relu (Activ ation)	(None, 56, 56, 64)	0	['conv2_block1_2_bn[0][0]']
conv2_block1_0_conv (Conv2 D)	(None, 56, 56, 256)	16640	['pool1_pool[0][0]']

conv2_block1_3_conv (Conv2D)	(None, 56, 56, 256)	16640	[ 'conv2_block1_2_relu[0][0]' ]
conv2_block1_0_bn (BatchNormalization)	(None, 56, 56, 256)	1024	[ 'conv2_block1_0_conv[0][0]' ]
conv2_block1_3_bn (BatchNormalization)	(None, 56, 56, 256)	1024	[ 'conv2_block1_3_conv[0][0]' ]
conv2_block1_add (Add)	(None, 56, 56, 256)	0	[ 'conv2_block1_0_bn[0][0]', 'conv2_block1_3_bn[0][0]' ]
conv2_block1_out (Activation)	(None, 56, 56, 256)	0	[ 'conv2_block1_add[0][0]' ]
conv2_block2_1_conv (Conv2D)	(None, 56, 56, 64)	16448	[ 'conv2_block1_out[0][0]' ]
conv2_block2_1_bn (BatchNormalization)	(None, 56, 56, 64)	256	[ 'conv2_block2_1_conv[0][0]' ]
conv2_block2_1_relu (Activation)	(None, 56, 56, 64)	0	[ 'conv2_block2_1_bn[0][0]' ]
conv2_block2_2_conv (Conv2D)	(None, 56, 56, 64)	36928	[ 'conv2_block2_1_relu[0][0]' ]
conv2_block3_2_bn (BatchNormalization)	(None, 56, 56, 64)	256	[ 'conv2_block3_2_conv[0][0]' ]
conv2_block3_2_relu (Activation)	(None, 56, 56, 64)	0	[ 'conv2_block3_2_bn[0][0]' ]
conv2_block3_3_conv (Conv2D)	(None, 56, 56, 256)	16640	[ 'conv2_block3_2_relu[0][0]' ]
conv2_block3_3_bn (BatchNormalization)	(None, 56, 56, 256)	1024	[ 'conv2_block3_3_conv[0][0]' ]
conv2_block3_add (Add)	(None, 56, 56, 256)	0	[ 'conv2_block2_out[0][0]', 'conv2_block3_3_bn[0][0]' ]
conv2_block3_out (Activation)	(None, 56, 56, 256)	0	[ 'conv2_block3_add[0][0]' ]
conv3_block1_1_conv (Conv2D)	(None, 28, 28, 128)	32896	[ 'conv2_block3_out[0][0]' ]
conv3_block1_1_bn (BatchNormalization)	(None, 28, 28, 128)	512	[ 'conv3_block1_1_conv[0][0]' ]
conv3_block1_1_relu (Activation)	(None, 28, 28, 128)	0	[ 'conv3_block1_1_bn[0][0]' ]
conv3_block1_2_conv (Conv2D)	(None, 28, 28, 128)	147584	[ 'conv3_block1_1_relu[0][0]' ]

conv3_block1_2_bn (BatchNormalization)	(None, 28, 28, 128)	512	['conv3_block1_2_conv[0][0]']
conv3_block1_2_relu (Activation)	(None, 28, 28, 128)	0	['conv3_block1_2_bn[0][0]']
conv3_block1_0_conv (Conv2D)	(None, 28, 28, 512)	131584	['conv2_block3_out[0][0]']
conv3_block1_3_conv (Conv2D)	(None, 28, 28, 512)	66048	['conv3_block1_2_relu[0][0]']
conv3_block1_0_bn (BatchNormalization)	(None, 28, 28, 512)	2048	['conv3_block1_0_conv[0][0]']
conv3_block1_3_bn (BatchNormalization)	(None, 28, 28, 512)	2048	['conv3_block1_3_conv[0][0]']
conv3_block1_add (Add)	(None, 28, 28, 512)	0	['conv3_block1_0_bn[0][0]', 'conv3_block1_3_bn[0][0]']
conv3_block1_out (Activation)	(None, 28, 28, 512)	0	['conv3_block1_add[0][0]']
conv3_block2_1_conv (Conv2D)	(None, 28, 28, 128)	65664	['conv3_block1_out[0][0]']
conv3_block2_1_bn (BatchNormalization)	(None, 28, 28, 128)	512	['conv3_block2_1_conv[0][0]']
conv3_block2_1_relu (Activation)	(None, 28, 28, 128)	0	['conv3_block2_1_bn[0][0]']
conv3_block2_2_conv (Conv2D)	(None, 28, 28, 128)	147584	['conv3_block2_1_relu[0][0]']
conv3_block2_2_bn (BatchNormalization)	(None, 28, 28, 128)	512	['conv3_block2_2_conv[0][0]']
conv3_block2_2_relu (Activation)	(None, 28, 28, 128)	0	['conv3_block2_2_bn[0][0]']
conv3_block2_3_conv (Conv2D)	(None, 28, 28, 512)	66048	['conv3_block2_2_relu[0][0]']
conv3_block2_3_bn (BatchNormalization)	(None, 28, 28, 512)	2048	['conv3_block2_3_conv[0][0]']
conv3_block2_add (Add)	(None, 28, 28, 512)	0	['conv3_block1_out[0][0]', 'conv3_block2_3_bn[0][0]']
conv3_block2_out (Activation)	(None, 28, 28, 512)	0	['conv3_block2_add[0][0]']
conv3_block3_1_conv (Conv2D)	(None, 28, 28, 128)	65664	['conv3_block2_out[0][0]']
conv3_block3_1_bn (BatchNormalization)	(None, 28, 28, 128)	512	['conv3_block3_1_conv[0][0]']

conv3_block3_1_relu (Activation)	0	[ 'conv3_block3_1_bn[0][0]' ]
conv3_block3_2_conv (Conv2D)	147584	[ 'conv3_block3_1_relu[0][0]' ]
conv3_block3_2_bn (Batch Normalization)	512	[ 'conv3_block3_2_conv[0][0]' ]
conv3_block3_2_relu (Activation)	0	[ 'conv3_block3_2_bn[0][0]' ]
conv3_block3_3_conv (Conv2D)	66048	[ 'conv3_block3_2_relu[0][0]' ]
conv3_block3_3_bn (Batch Normalization)	2048	[ 'conv3_block3_3_conv[0][0]' ]
conv3_block3_add (Add)	0	[ 'conv3_block2_out[0][0]', 'conv3_block3_3_bn[0][0]' ]
conv3_block3_out (Activation)	0	[ 'conv3_block3_add[0][0]' ]
conv3_block4_1_conv (Conv2D)	65664	[ 'conv3_block3_out[0][0]' ]
conv3_block4_1_bn (Batch Normalization)	512	[ 'conv3_block4_1_conv[0][0]' ]
conv3_block4_1_relu (Activation)	0	[ 'conv3_block4_1_bn[0][0]' ]
conv3_block4_2_conv (Conv2D)	147584	[ 'conv3_block4_1_relu[0][0]' ]
conv3_block4_2_bn (Batch Normalization)	512	[ 'conv3_block4_2_conv[0][0]' ]
conv3_block4_2_relu (Activation)	0	[ 'conv3_block4_2_bn[0][0]' ]
conv3_block4_3_conv (Conv2D)	66048	[ 'conv3_block4_2_relu[0][0]' ]
conv3_block4_3_bn (Batch Normalization)	2048	[ 'conv3_block4_3_conv[0][0]' ]
conv3_block4_add (Add)	0	[ 'conv3_block3_out[0][0]', 'conv3_block4_3_bn[0][0]' ]
conv3_block4_out (Activation)	0	[ 'conv3_block4_add[0][0]' ]
conv4_block1_1_conv (Conv2D)	131328	[ 'conv3_block4_out[0][0]' ]
conv4_block1_1_bn (Batch Normalization)	1024	[ 'conv4_block1_1_conv[0][0]' ]

conv4_block1_1_relu (Activation)	0	['conv4_block1_1_bn[0][0]']
conv4_block1_2_conv (Conv2D)	590080	['conv4_block1_1_relu[0][0]']
conv4_block1_2_bn (BatchNormalization)	1024	['conv4_block1_2_conv[0][0]']
conv4_block1_2_relu (Activation)	0	['conv4_block1_2_bn[0][0]']
conv4_block1_0_conv (Conv2D)	525312	['conv3_block4_out[0][0]']
conv4_block1_3_conv (Conv2D)	263168	['conv4_block1_2_relu[0][0]']
conv4_block1_0_bn (BatchNormalization)	4096	['conv4_block1_0_conv[0][0]']
conv4_block1_3_bn (BatchNormalization)	4096	['conv4_block1_3_conv[0][0]']
conv4_block1_add (Add)	0	['conv4_block1_0_bn[0][0]', 'conv4_block1_3_bn[0][0]']
conv4_block1_out (Activation)	0	['conv4_block1_add[0][0]']
conv4_block2_1_conv (Conv2D)	262400	['conv4_block1_out[0][0]']
conv4_block2_1_bn (BatchNormalization)	1024	['conv4_block2_1_conv[0][0]']
conv4_block2_1_relu (Activation)	0	['conv4_block2_1_bn[0][0]']
conv4_block2_2_conv (Conv2D)	590080	['conv4_block2_1_relu[0][0]']
conv4_block2_2_bn (BatchNormalization)	1024	['conv4_block2_2_conv[0][0]']
conv4_block2_2_relu (Activation)	0	['conv4_block2_2_bn[0][0]']
conv4_block2_3_conv (Conv2D)	263168	['conv4_block2_2_relu[0][0]']
conv4_block2_3_bn (BatchNormalization)	4096	['conv4_block2_3_conv[0][0]']
conv4_block2_add (Add)	0	['conv4_block1_out[0][0]', 'conv4_block2_3_bn[0][0]']
conv4_block2_out (Activation)	0	['conv4_block2_add[0][0]']

conv4_block3_1_conv (Conv2D)	262400	['conv4_block2_out[0][0]']
conv4_block3_1_bn (BatchNormalization)	1024	['conv4_block3_1_conv[0][0]']
conv4_block3_1_relu (Activation)	0	['conv4_block3_1_bn[0][0]']
conv4_block3_2_conv (Conv2D)	590080	['conv4_block3_1_relu[0][0]']
conv4_block3_2_bn (BatchNormalization)	1024	['conv4_block3_2_conv[0][0]']
conv4_block3_2_relu (Activation)	0	['conv4_block3_2_bn[0][0]']
conv4_block3_3_conv (Conv2D)	263168	['conv4_block3_2_relu[0][0]']
conv4_block3_3_bn (BatchNormalization)	4096	['conv4_block3_3_conv[0][0]']
conv4_block3_add (Add)	0	['conv4_block2_out[0][0]', 'conv4_block3_3_bn[0][0]']
conv4_block3_out (Activation)	0	['conv4_block3_add[0][0]']
conv4_block4_1_conv (Conv2D)	262400	['conv4_block3_out[0][0]']
conv4_block4_1_bn (BatchNormalization)	1024	['conv4_block4_1_conv[0][0]']
conv4_block4_1_relu (Activation)	0	['conv4_block4_1_bn[0][0]']
conv4_block4_2_conv (Conv2D)	590080	['conv4_block4_1_relu[0][0]']
conv4_block4_2_bn (BatchNormalization)	1024	['conv4_block4_2_conv[0][0]']
conv4_block4_2_relu (Activation)	0	['conv4_block4_2_bn[0][0]']
conv4_block4_3_conv (Conv2D)	263168	['conv4_block4_2_relu[0][0]']
conv4_block4_3_bn (BatchNormalization)	4096	['conv4_block4_3_conv[0][0]']
conv4_block4_add (Add)	0	['conv4_block3_out[0][0]', 'conv4_block4_3_bn[0][0]']
conv4_block4_out (Activation)	0	['conv4_block4_add[0][0]']

conv4_block5_1_conv (Conv2D)	(None, 14, 14, 256)	262400	[ 'conv4_block4_out[0][0]' ]
conv4_block5_1_bn (BatchNormalization)	(None, 14, 14, 256)	1024	[ 'conv4_block5_1_conv[0][0]' ]
conv4_block5_1_relu (Activation)	(None, 14, 14, 256)	0	[ 'conv4_block5_1_bn[0][0]' ]
conv4_block5_2_conv (Conv2D)	(None, 14, 14, 256)	590080	[ 'conv4_block5_1_relu[0][0]' ]
conv4_block5_2_bn (BatchNormalization)	(None, 14, 14, 256)	1024	[ 'conv4_block5_2_conv[0][0]' ]
conv4_block5_2_relu (Activation)	(None, 14, 14, 256)	0	[ 'conv4_block5_2_bn[0][0]' ]
conv4_block5_3_conv (Conv2D)	(None, 14, 14, 1024)	263168	[ 'conv4_block5_2_relu[0][0]' ]
conv4_block5_3_bn (BatchNormalization)	(None, 14, 14, 1024)	4096	[ 'conv4_block5_3_conv[0][0]' ]
conv4_block5_add (Add)	(None, 14, 14, 1024)	0	[ 'conv4_block4_out[0][0]', 'conv4_block5_3_bn[0][0]' ]
conv4_block5_out (Activation)	(None, 14, 14, 1024)	0	[ 'conv4_block5_add[0][0]' ]
conv4_block6_1_conv (Conv2D)	(None, 14, 14, 256)	262400	[ 'conv4_block5_out[0][0]' ]
conv4_block6_1_bn (BatchNormalization)	(None, 14, 14, 256)	1024	[ 'conv4_block6_1_conv[0][0]' ]
conv4_block6_1_relu (Activation)	(None, 14, 14, 256)	0	[ 'conv4_block6_1_bn[0][0]' ]
conv4_block6_2_conv (Conv2D)	(None, 14, 14, 256)	590080	[ 'conv4_block6_1_relu[0][0]' ]
conv4_block6_2_bn (BatchNormalization)	(None, 14, 14, 256)	1024	[ 'conv4_block6_2_conv[0][0]' ]
conv4_block6_2_relu (Activation)	(None, 14, 14, 256)	0	[ 'conv4_block6_2_bn[0][0]' ]
conv4_block6_3_conv (Conv2D)	(None, 14, 14, 1024)	263168	[ 'conv4_block6_2_relu[0][0]' ]
conv4_block6_3_bn (BatchNormalization)	(None, 14, 14, 1024)	4096	[ 'conv4_block6_3_conv[0][0]' ]
conv4_block6_add (Add)	(None, 14, 14, 1024)	0	[ 'conv4_block5_out[0][0]', 'conv4_block6_3_bn[0][0]' ]
conv4_block6_out (Activation)	(None, 14, 14, 1024)	0	[ 'conv4_block6_add[0][0]' ]

conv5_block1_1_conv (Conv2D)	524800	[ 'conv4_block6_out[0][0]' ]
conv5_block1_1_bn (BatchNormalization)	2048	[ 'conv5_block1_1_conv[0][0]' ]
conv5_block1_1_relu (Activation)	0	[ 'conv5_block1_1_bn[0][0]' ]
conv5_block1_2_conv (Conv2D)	2359808	[ 'conv5_block1_1_relu[0][0]' ]
conv5_block1_2_bn (BatchNormalization)	2048	[ 'conv5_block1_2_conv[0][0]' ]
conv5_block1_2_relu (Activation)	0	[ 'conv5_block1_2_bn[0][0]' ]
conv5_block1_0_conv (Conv2D)	2099200	[ 'conv4_block6_out[0][0]' ]
conv5_block1_3_conv (Conv2D)	1050624	[ 'conv5_block1_2_relu[0][0]' ]
conv5_block1_0_bn (BatchNormalization)	8192	[ 'conv5_block1_0_conv[0][0]' ]
conv5_block1_3_bn (BatchNormalization)	8192	[ 'conv5_block1_3_conv[0][0]' ]
conv5_block1_add (Add)	0	[ 'conv5_block1_0_bn[0][0]', 'conv5_block1_3_bn[0][0]' ]
conv5_block1_out (Activation)	0	[ 'conv5_block1_add[0][0]' ]
conv5_block2_1_conv (Conv2D)	1049088	[ 'conv5_block1_out[0][0]' ]
conv5_block2_1_bn (BatchNormalization)	2048	[ 'conv5_block2_1_conv[0][0]' ]
conv5_block2_1_relu (Activation)	0	[ 'conv5_block2_1_bn[0][0]' ]
conv5_block2_2_conv (Conv2D)	2359808	[ 'conv5_block2_1_relu[0][0]' ]
conv5_block2_2_bn (BatchNormalization)	2048	[ 'conv5_block2_2_conv[0][0]' ]
conv5_block2_2_relu (Activation)	0	[ 'conv5_block2_2_bn[0][0]' ]
conv5_block2_3_conv (Conv2D)	1050624	[ 'conv5_block2_2_relu[0][0]' ]
conv5_block2_3_bn (BatchNormalization)	8192	[ 'conv5_block2_3_conv[0][0]' ]

conv5_block2_add (Add)	(None, 7, 7, 2048)	0	['conv5_block1_out[0][0]', 'conv5_block2_3_bn[0][0]']
conv5_block2_out (Activation)	(None, 7, 7, 2048)	0	['conv5_block2_add[0][0]']
conv5_block3_1_conv (Conv2D)	(None, 7, 7, 512)	1049088	['conv5_block2_out[0][0]']
conv5_block3_1_bn (Batch Normalization)	(None, 7, 7, 512)	2048	['conv5_block3_1_conv[0][0]']
conv5_block3_1_relu (Activation)	(None, 7, 7, 512)	0	['conv5_block3_1_bn[0][0]']
conv5_block3_2_conv (Conv2D)	(None, 7, 7, 512)	2359808	['conv5_block3_1_relu[0][0]']
conv5_block3_2_bn (Batch Normalization)	(None, 7, 7, 512)	2048	['conv5_block3_2_conv[0][0]']
conv5_block3_2_relu (Activation)	(None, 7, 7, 512)	0	['conv5_block3_2_bn[0][0]']
conv5_block3_3_conv (Conv2D)	(None, 7, 7, 2048)	1050624	['conv5_block3_2_relu[0][0]']
conv5_block3_3_bn (Batch Normalization)	(None, 7, 7, 2048)	8192	['conv5_block3_3_conv[0][0]']
conv5_block3_add (Add)	(None, 7, 7, 2048)	0	['conv5_block2_out[0][0]', 'conv5_block3_3_bn[0][0]']
conv5_block3_out (Activation)	(None, 7, 7, 2048)	0	['conv5_block3_add[0][0]']
global_average_pooling2d (GlobalAveragePooling2D)	(None, 2048)	0	['conv5_block3_out[0][0]']
dense (Dense)	(None, 128)	262272	['global_average_pooling2d[0][0]']
dense_1 (Dense)	(None, 6)	774	['dense[0][0]']

---

Total params: 23850758 (90.98 MB)  
Trainable params: 263046 (1.00 MB)  
Non-trainable params: 23587712 (89.98 MB)

### ۳-۳) مرحله‌ی آموزش

همانطور که در صورت سوال گفته شده است، ابتدا به منظور بهره‌گیری از یادگیری انتقالی در مراحل ابتدایی، ضرایب وزنی همه‌ی لایه‌های شبکه، ثابت نگه داشته شده (از نظر یادگیری غیرفعال شده) و تنها بخش جدید

(head) شبکه آموزش داده می‌شود. بنابراین، در مرحله اول، داده‌های آموزشی از train\_generator گرفته می‌شود. سپس، تعداد دوره‌های آموزشی (ایپاک‌ها) به ۲۵ تنظیم می‌شود. هم‌چنین، داده‌های آزمون از val\_generator استفاده می‌شود تا عملکرد مدل در طول آموزش بررسی شود. در نهایت، نتیجه فرآیند آموزش در متغیر history\_resnet ذخیره می‌شود. کد مربوط به آن در زیر آورده شده است:

```
history_resnet = model_resnet.fit(
    train_generator,
    epochs=25,
    validation_data=val_generator
)
```

و پاسخ آن به صورت زیر است:

```
Epoch 1/25
39/39 [=====] - 314s 8s/step - loss: 1.6686 - accuracy: 0.2876 - val_loss: 1.6508 - val_accuracy: 0.2315
Epoch 2/25
39/39 [=====] - 318s 8s/step - loss: 1.6588 - accuracy: 0.3007 - val_loss: 1.6125 - val_accuracy: 0.2639
Epoch 3/25
39/39 [=====] - 321s 8s/step - loss: 1.6025 - accuracy: 0.3309 - val_loss: 1.5847 - val_accuracy: 0.3194
Epoch 4/25
39/39 [=====] - 318s 8s/step - loss: 1.5842 - accuracy: 0.3652 - val_loss: 1.5488 - val_accuracy: 0.3148
Epoch 5/25
39/39 [=====] - 316s 8s/step - loss: 1.5476 - accuracy: 0.3521 - val_loss: 1.4958 - val_accuracy: 0.4306
Epoch 6/25
39/39 [=====] - 327s 8s/step - loss: 1.5110 - accuracy: 0.4003 - val_loss: 1.5171 - val_accuracy: 0.3380
Epoch 7/25
39/39 [=====] - 324s 8s/step - loss: 1.4981 - accuracy: 0.3938 - val_loss: 1.4486 - val_accuracy: 0.4028
Epoch 8/25
39/39 [=====] - 331s 8s/step - loss: 1.4506 - accuracy: 0.4363 - val_loss: 1.4468 - val_accuracy: 0.3657
Epoch 9/25
39/39 [=====] - 320s 8s/step - loss: 1.4239 - accuracy: 0.4722 - val_loss: 1.3962 - val_accuracy: 0.4861
Epoch 10/25
39/39 [=====] - 318s 8s/step - loss: 1.3996 - accuracy: 0.4297 - val_loss: 1.3978 - val_accuracy: 0.3287
Epoch 11/25
39/39 [=====] - 312s 8s/step - loss: 1.3703 - accuracy: 0.4444 - val_loss: 1.3447 - val_accuracy: 0.5324
Epoch 12/25
39/39 [=====] - 318s 8s/step - loss: 1.3594 - accuracy: 0.4926 - val_loss: 1.2912 - val_accuracy: 0.4769
Epoch 13/25
39/39 [=====] - 316s 8s/step - loss: 1.3410 - accuracy: 0.4387 - val_loss: 1.3023 - val_accuracy: 0.5046
Epoch 14/25
39/39 [=====] - 315s 8s/step - loss: 1.2951 - accuracy: 0.5376 - val_loss: 1.2517 - val_accuracy: 0.4815
Epoch 15/25
39/39 [=====] - 334s 9s/step - loss: 1.2888 - accuracy: 0.4910 - val_loss: 1.2996 - val_accuracy: 0.3657
Epoch 16/25
39/39 [=====] - 320s 8s/step - loss: 1.2624 - accuracy: 0.5327 - val_loss: 1.2043 - val_accuracy: 0.5741
Epoch 17/25
39/39 [=====] - 317s 8s/step - loss: 1.2388 - accuracy: 0.5368 - val_loss: 1.2326 - val_accuracy: 0.5231
Epoch 18/25
39/39 [=====] - 314s 8s/step - loss: 1.2133 - accuracy: 0.5621 - val_loss: 1.2454 - val_accuracy: 0.4769
Epoch 19/25
39/39 [=====] - 320s 8s/step - loss: 1.1899 - accuracy: 0.5662 - val_loss: 1.1361 - val_accuracy: 0.5694
Epoch 20/25
39/39 [=====] - 314s 8s/step - loss: 1.1631 - accuracy: 0.5972 - val_loss: 1.1445 - val_accuracy: 0.5787
Epoch 21/25
39/39 [=====] - 314s 8s/step - loss: 1.1415 - accuracy: 0.5980 - val_loss: 1.1498 - val_accuracy: 0.4722
Epoch 22/25
39/39 [=====] - 317s 8s/step - loss: 1.1338 - accuracy: 0.6103 - val_loss: 1.0926 - val_accuracy: 0.6481
Epoch 23/25
39/39 [=====] - 316s 8s/step - loss: 1.1201 - accuracy: 0.5858 - val_loss: 1.0990 - val_accuracy: 0.6065
Epoch 24/25
39/39 [=====] - 320s 8s/step - loss: 1.1274 - accuracy: 0.5980 - val_loss: 1.0370 - val_accuracy: 0.6019
Epoch 25/25
39/39 [=====] - 325s 8s/step - loss: 1.1107 - accuracy: 0.5923 - val_loss: 1.0846 - val_accuracy: 0.5370
```

در ادامه، طبق صورت سوال، به تدریج و از انتهای شبکه، برخی لایه‌ها از حالت ثابت نگه داشته شده (فریز) خارج شده و مدل به صورت مرحله‌ای تحت فرآیند Fine-Tuning قرار می‌گیرد. به همین منظور، ابتدا حلقه‌ای برای تنظیم قابلیت آموزش لایه‌های پایانی مدل پایه (base\_model) ایجاد می‌شود. در این حلقه، لایه‌های آخر ۳۰ مدل به گونه‌ای تنظیم می‌شوند که قابلیت آموزش داشته باشند. این به این معنی است که وزن‌های این لایه‌ها در طی فرآیند آموزش به روز خواهند شد. سپس، یک بهینه‌ساز از نوع Adam با نرخ یادگیری بسیار کم ( $1e^{-5}$ ) انتخاب می‌شود. هم‌چنین،تابع هزینه categorical\_crossentropy برای ارزیابی عملکرد مدل در حین آموزش و معیار دقت (accuracy) برای اندازه‌گیری عملکرد مدل تعیین می‌شود.

در نهایت، مدل با استفاده از روش fit آموزش داده می‌شود. در این مرحله، برای داده‌های آموزشی از استفاده از val\_generator و برای داده‌های آزمون از train\_generator و برای تغییر ذخیره می‌شود تا بتوان نتایج را بعداً (epoch) ادامه خواهد داشت و تاریخچه آموزش در history\_finetune ذخیره می‌شود. فرآیند آموزش برای ۲۵ دوره بررسی کرد. کد مربوط به آن در ادامه ارائه شده است:

```
for layer in base_model.layers[-30:]:
    layer.trainable = True

model_resnet.compile(optimizer=tf.keras.optimizers.Adam(1e-5),
                      loss='categorical_crossentropy',
                      metrics=['accuracy'])

history_finetune = model_resnet.fit(
    train_generator,
    epochs=25,
    validation_data=val_generator
)
```

پاسخ کد فوق به صورت زیر است:

```
Epoch 1/25
39/39 [=====] - 437s 11s/step - loss: 9.4732 - accuracy: 0.4812 - val_loss: 7.3057 - val_accuracy: 0.2824
Epoch 2/25
39/39 [=====] - 422s 11s/step - loss: 0.9431 - accuracy: 0.6814 - val_loss: 26.8096 - val_accuracy: 0.1667
Epoch 3/25
39/39 [=====] - 422s 11s/step - loss: 0.7139 - accuracy: 0.7647 - val_loss: 34.3263 - val_accuracy: 0.1667
Epoch 4/25
39/39 [=====] - 429s 11s/step - loss: 0.6601 - accuracy: 0.7672 - val_loss: 33.1444 - val_accuracy: 0.1667
Epoch 5/25
39/39 [=====] - 423s 11s/step - loss: 0.6329 - accuracy: 0.7966 - val_loss: 27.1167 - val_accuracy: 0.1667
Epoch 6/25
39/39 [=====] - 440s 11s/step - loss: 0.5564 - accuracy: 0.8309 - val_loss: 18.7796 - val_accuracy: 0.1667
Epoch 7/25
39/39 [=====] - 427s 11s/step - loss: 0.5742 - accuracy: 0.8162 - val_loss: 12.1562 - val_accuracy: 0.2222
```

```

Epoch 8/25
39/39 [=====] - 452s 12s/step - loss: 0.6099 - accuracy: 0.7900 - val_loss: 8.6376 - val_accuracy: 0.2870
Epoch 9/25
39/39 [=====] - 437s 11s/step - loss: 0.5256 - accuracy: 0.8325 - val_loss: 6.5637 - val_accuracy: 0.3333
Epoch 10/25
39/39 [=====] - 427s 11s/step - loss: 0.5201 - accuracy: 0.8374 - val_loss: 1.3297 - val_accuracy: 0.4537
Epoch 11/25
39/39 [=====] - 472s 12s/step - loss: 0.4603 - accuracy: 0.8505 - val_loss: 1.4523 - val_accuracy: 0.4722
Epoch 12/25
39/39 [=====] - 430s 11s/step - loss: 0.4582 - accuracy: 0.8676 - val_loss: 1.5403 - val_accuracy: 0.4537
Epoch 13/25
39/39 [=====] - 423s 11s/step - loss: 0.3898 - accuracy: 0.9011 - val_loss: 0.4243 - val_accuracy: 0.8796
Epoch 14/25
39/39 [=====] - 434s 11s/step - loss: 0.4221 - accuracy: 0.8652 - val_loss: 0.4800 - val_accuracy: 0.8380
Epoch 15/25
39/39 [=====] - 441s 11s/step - loss: 0.4376 - accuracy: 0.8636 - val_loss: 0.3458 - val_accuracy: 0.9213
Epoch 16/25
39/39 [=====] - 438s 11s/step - loss: 0.4416 - accuracy: 0.8513 - val_loss: 0.4986 - val_accuracy: 0.8102
Epoch 17/25
39/39 [=====] - 421s 11s/step - loss: 0.4049 - accuracy: 0.8668 - val_loss: 0.5291 - val_accuracy: 0.8241
Epoch 18/25
39/39 [=====] - 420s 11s/step - loss: 0.3774 - accuracy: 0.8791 - val_loss: 0.3229 - val_accuracy: 0.9074
Epoch 19/25
39/39 [=====] - 425s 11s/step - loss: 0.3639 - accuracy: 0.8832 - val_loss: 0.5599 - val_accuracy: 0.7963
Epoch 20/25
39/39 [=====] - 422s 11s/step - loss: 0.3859 - accuracy: 0.8725 - val_loss: 0.5031 - val_accuracy: 0.8565
Epoch 21/25
39/39 [=====] - 423s 11s/step - loss: 0.3763 - accuracy: 0.8701 - val_loss: 0.3992 - val_accuracy: 0.8704
Epoch 22/25
39/39 [=====] - 421s 11s/step - loss: 0.3239 - accuracy: 0.8987 - val_loss: 0.5557 - val_accuracy: 0.7222
Epoch 23/25
39/39 [=====] - 420s 11s/step - loss: 0.3231 - accuracy: 0.8946 - val_loss: 0.7266 - val_accuracy: 0.7546
Epoch 24/25
39/39 [=====] - 421s 11s/step - loss: 0.4437 - accuracy: 0.8554 - val_loss: 0.2901 - val_accuracy: 0.9213
Epoch 25/25
39/39 [=====] - 421s 11s/step - loss: 0.3725 - accuracy: 0.8742 - val_loss: 0.7112 - val_accuracy: 0.6389

```

### ۴-۳) ارزیابی نهایی

۱-۴-۳) رسم نمودارهای آموزش و آزمون و تحلیل هر

کدام

در این بخش، نمودارهای دقت (Accuracy) و خطا (Loss) برای داده‌های آموزش و آزمون مدل‌های ResNet50 Fine-Tuning و ResNet50 Head Training رسم می‌شود. کد مربوط به آن در زیر آورده شده است:

```

def plot_history(history, title=""):
    plt.figure(figsize=(12, 4))

    plt.subplot(1, 2, 1)
    plt.plot(history.history['accuracy'], label='Train Accuracy')
    plt.plot(history.history['val_accuracy'], label='Val Accuracy')
    plt.title(f'{title} Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()

```

```

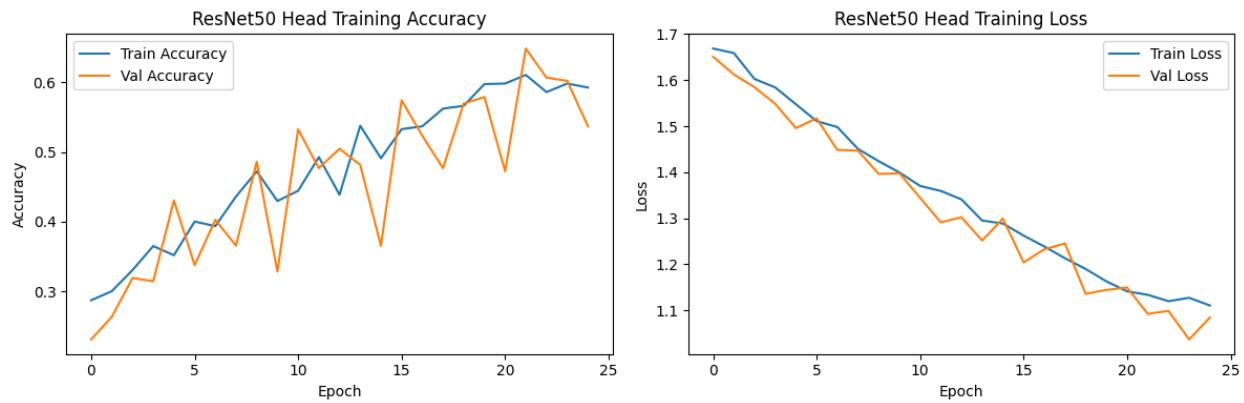
plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Val Loss')
plt.title(f'{title} Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.tight_layout()
plt.show()

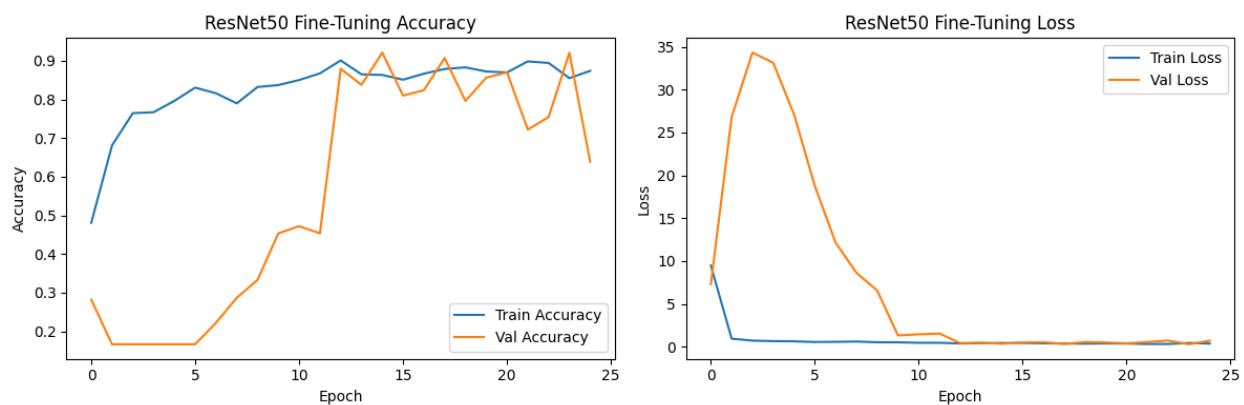
plot_history(history_resnet, title="ResNet50 Head Training")
plot_history(history_finetune, title="ResNet50 Fine-Tuning")

```

که پاسخ آن به صورت زیر است:



شکل ۱۳: نمودارهای خطا و دقت برای داده‌های آموزش و آزمون شبکه ResNet50 Head Training



شکل ۱۴: نمودارهای خطا و دقت برای داده‌های آموزش و آزمون شبکه ResNet50 Fine-Tuning

**تحلیل نمودارهای ResNet50 Head Training**

نمودار	آموزش و آزمون	تحلیل
نمودار خطا (Loss)	آموزش (train)	خطای آموزش از حدود ۱.۶۷ به نزدیک ۱.۰۵ کاهش یافته است. کاهش پیوسته خطا نشان می‌دهد مدل به خوبی در حال بهینه‌سازی روی داده‌های آموزشی است.
	آزمون (test)	خطای آزمون با روند مشابه آموزش کاهش یافته و به حدود ۱.۰۵ رسیده است. تطابق نسبتاً خوب بین کاهش خطای آموزش و آزمون دیده می‌شود که نشانه‌ای مثبت برای عدم overfitting است.
	آموزش (train)	دقت آموزش از حدود ۰.۲۸ در ابتدای آموزش به حدود ۰.۶۰ در انتهای آفزایش یافته است. روند دقت نسبتاً صعودی و پایدار است و نشان می‌دهد مدل در حال یادگیری از داده‌های آموزشی است.
نمودار دقت (Accuracy)	آزمون (test)	دقت آزمون با نوساناتی زیاد از حدود ۰.۲۴ شروع و تا ۰.۶۵ آفزایش یافته است. نوسانات زیاد (برخلاف آموزش) نشان‌دهنده ناپایداری یادگیری روی داده‌های دیده نشده است. مدل در برخی ایپاک‌ها بهتر از آموزش عمل کرده است که ممکن است ناشی از تصادفی بودن داده‌ها یا کوچک بودن مجموعه آزمون باشد.

**تحلیل نمودارهای ResNet50 Fine-Tuning**

نمودار	آموزش و آزمون	تحلیل
نمودار خطا (Loss)	آموزش (train)	خطای آموزش از مقدار اولیه حدود ۹ به زیر ۱ کاهش یافته و تقریباً به صفر نزدیک شده است. این نشان‌دهنده یادگیری عمیق مدل از داده‌های آموزشی و همگرایی مناسب شبکه است.
	آزمون (test)	خطای آزمون در ابتدا بسیار بالا بوده (۳۵)، اما با سرعت زیاد کاهش یافته و به نزدیک صفر رسیده است. کاهش سریع و همگرایی خوب خطای آزمون به معنای عملکرد مؤثر مدل روی داده‌های دیده نشده است.
نمودار دقت (Accuracy)	آموزش (train)	دقت آموزش از حدود ۰.۴۸ شروع و تا حدود ۰.۸۹ آفزایش یافته است. روند صعودی نسبتاً یکنواختی دارد که نشانه‌ی یادگیری پایدار و مؤثر است.

دقت آزمون در ابتدا پایین (۰.۲۸) بوده، اما به سرعت افزایش یافته و تا حدود ۰.۹۰ بالا رفته است. دقต در ایپاک‌های میانی تا انتهای نوساناتی دارد. اما در سطح بالایی باقی می‌ماند.

آزمون (test)

### ۲-۴-۳) دقت نهایی مدل روی داده‌های آزمون

برای تعیین دقت نهایی مدل روی داده‌های آزمون از کد زیر می‌شود:

```
loss_resnet, acc_resnet = model_resnet.evaluate(val_generator)
print(f"Final Test Accuracy: {acc_resnet * 100:.2f}%")
print(f"Final Test Loss: {loss_resnet:.4f}")
```

که پاسخ آن به صورت زیر است:

```
7/7 [=====] - 38s 5s/step - loss: 0.6776 - accuracy: 0.6667
Final Test Accuracy: 66.67%
Final Test Loss: 0.6776
```

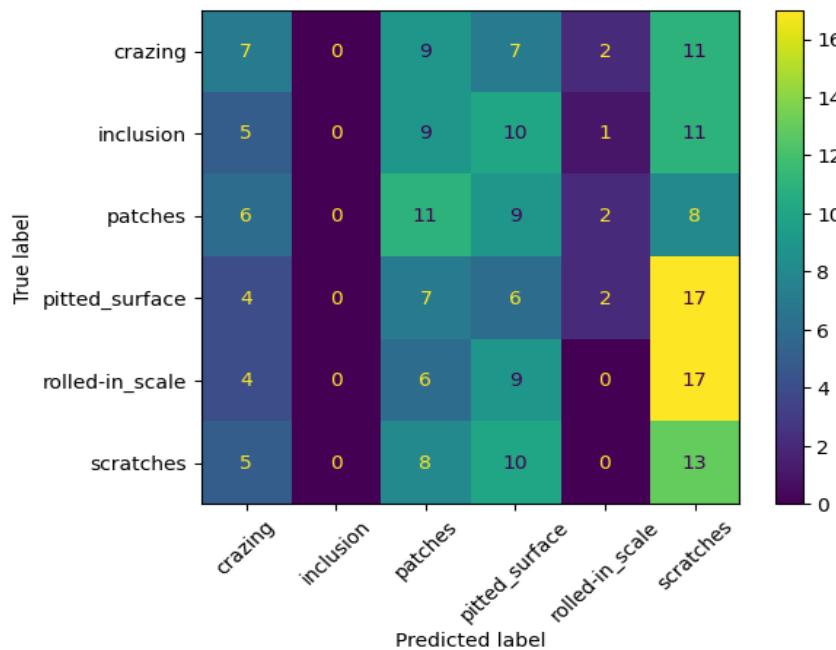
### ۳-۴-۳) رسم ماتریس آشتفتگی برای داده‌های آزمون و تفسیر آن

ماتریس‌های آشتفتگی برای داده‌های آزمون مدل ResNet-50 با استفاده از کد زیر رسم می‌شود:

```
y_true = val_generator.classes
y_pred = np.argmax(model_resnet.predict(val_generator), axis=1)

cm = confusion_matrix(y_true, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=class_names)
disp.plot(xticks_rotation=45)
plt.show()
```

پاسخ کد فوق به صورت زیر است:



شکل ۱۵: ماتریس آشفتگی برای داده‌های آزمون مدل ResNet-50

### تفسیر نمودار ماتریس آشفتگی مدل ResNet-50

هر ماتریس آشفتگی یک شبکه  $6 \times 6$  برای ۶ کلاس فعالیت (با برجسب‌های ۰ تا ۵) است که لیبل و درصد طبقه‌بندی درست هر یک از کلاس‌ها در مدل ResNet-50 در جدول زیر ارائه شده است:

Description	Label	Class
CNN-Block: 19.44% (7/36)	Crazing	0
CNN-Block: 0% (0/36)	Inclusion	1
CNN-Block: 30.56% (11/36)	Patches	2
CNN-Block: 16.67% (6/36)	Pitted_Surface	3
CNN-Block: 0% (0/36)	Rolled-in_Scale	4
CNN-Block: 36.11% (13/36)	Scratches	5

✓ تحلیل مدل ResNet-50

مدل ResNet-50 در تشخیص بسیاری از کلاس‌ها عملکرد خوبی ندارد، مخصوصاً کلاس ۱ و ۴ (Inclusion) و کلاس‌های دیگر نیز در ادامه به‌طور خلاصه توضیح داده شده است:

► کلاس صفر (Crazing): ۹ مورد به اشتباه در کلاس ۲ (Patches)، ۷ مورد به اشتباه در کلاس ۳ (Scratches) طبقه‌بندی شده‌اند.

► کلاس ۲ (Patches): ۶ مورد به اشتباه در کلاس صفر (Crazing)، ۹ مورد به اشتباه در کلاس ۳ (Scratches) طبقه‌بندی شده‌اند.

► کلاس ۳ (Scratches): ۴ مورد به اشتباه در کلاس صفر (Crazing)، ۷ مورد به اشتباه در کلاس ۲ (Pitted\_Surface)، ۲ مورد به اشتباه در کلاس ۴ (Rolled-in\_Scale) و ۸ مورد به اشتباه در کلاس ۵ (Pitted\_Surface) طبقه‌بندی شده‌اند.

► کلاس ۵ (Scratches): ۵ مورد به اشتباه در کلاس صفر (Crazing)، ۸ مورد به اشتباه در کلاس ۲ (Patches) و ۱۷ مورد به اشتباه در کلاس ۴ (Rolled-in\_Scale) طبقه‌بندی شده‌اند.

بنابراین، مدل ResNet-50 در تشخیص کلاس‌های ۱ و ۴ (Rolled-in\_Scale و Inclusion) کمترین دقیق و در تشخیص کلاس ۵ (Scratches) بیشترین دقیق را داشته است.

### ۵-۳ مقایسه و تحلیل

با توجه به اینکه مدل CNN-Factorized در بخش قبل دارای بیشترین دقیق نهایی روی داده‌های آزمون بود، این مدل برای مقایسه با مدل ResNet-50 در نظر گذرفته شده است.

✓ مقایسه دقیق نهایی:

مدل	دقت نهایی (Test Accuracy)
ResNet-50	66.67%
CNN_factorized	89.26%

مدل CNN به طور قابل توجهی بهتر از ResNet-50 عمل کرده است. این نشان می‌دهد که طراحی اختصاصی و آموزش از ابتدا روی این مجموعه داده، منجر به یادگیری ویژگی‌های مناسب‌تری شده است.

#### ✓ مقایسه زمان آموزش:

مدل	زمان آموزش (تقریبی)
ResNet-50	بیشتر (به خصوص در مرحله fine-tuning به دلیل عمق زیاد شبکه)
CNN_factorized	کمتر (ساختار سبک‌تر و کمتر پیچیده)

مدل ResNet-50 به دلیل تعداد زیاد لایه‌ها و پارامترها، نیاز به زمان آموزش بیشتر دارد، مخصوصاً در مرحله Fine-Tuning

#### ✓ مقایسه اندازه و پیچیدگی مدل:

مدل	تعداد پارامترها	پیچیدگی
ResNet-50	حدود 23 میلیون	بسیار پیچیده
CNN_factorized	چند صد هزار	ساده‌تر و سریع‌تر

مدل ResNet-50 سنگین‌تر و نیازمند منابع پردازشی بیشتری است. در حالی که CNN\_factorized ساده‌تر و مناسب‌تر.

**✓ مزایا و معایب یادگیری انتقالی (Transfer Learning)****➤ مزایا:**

- استفاده از ویژگی‌های از پیش یادگرفته شده روی مجموعه‌های بزرگ.
- مفید در مواردی که داده‌های آموزش کم هستند.
- صرفه‌جویی در زمان آموزش اولیه.

**➤ معایب:**

- مدل ممکن است به خوبی با داده‌های خاص تطبیق پیدا نکند.
- گاهی ویژگی‌های انتقالی ناکارآمد است.
- مدل بزرگ‌تر و کندتر در اجرا است.

**❖ گیت هاب**

گزارش و کد مربوط به این تمرین در گیت هاب به آدرس

[https://github.com/MM-Touiserkani/AI\\_HW4\\_Touiserkani](https://github.com/MM-Touiserkani/AI_HW4_Touiserkani)

قرار داده شده است.