ARTIFICIAL NEURAL NETWORKS

Classification Using Artificial Neural Networks with Hyperparameter Tuning on Alphabets Data

Overview

In this assignment, you will be tasked with developing a classification model using Artificial Neural Networks (ANNs) to classify data points from the "Alphabets_data.csv" dataset into predefined categories of alphabets. This exercise aims to deepen your understanding of ANNs and the significant role hyperparameter tuning plays in enhancing model performance.

Dataset: "Alphabets_data.csv"

The dataset provided, "Alphabets_data.csv", consists of labeled data suitable for a classification task aimed at identifying different alphabets. Before using this data in your model, you'll need to preprocess it to ensure optimal performance.

Tasks

1. Data Exploration and Preprocessing

- **Begin by loading and exploring the "Alphabets_data.csv" dataset. Summarize its key features such as the number of samples, features, and classes.**
- **Execute necessary data preprocessing steps including data normalization, managing missing values.**

Answer :

Code used :

```
import pandas as pd
import pprint
import os

# --- File paths ---
input_path = r"D:\DATA SCIENCE\ASSIGNMENTS\18 neural networks\Neural networks\Alphabets_data.csv"
output_folder = r"d:\DATA SCIENCE\ASSIGNMENTS\18 neural networks\Neural networks"
output_file = os.path.join(output_folder, "dataset_overview.txt")

# --- Load dataset ---
alphabets_data_df = pd.read_csv(input_path)

# --- Dataset summary ---
dataset_overview = {
    "Shape": alphabets_data_df.shape,
    "Column_Names": alphabets_data_df.columns.tolist(),
    "Missing_Values": alphabets_data_df.isnull().sum().to_dict(),
    "Data_Types": alphabets_data_df.dtypes.astype(str).to_dict(),
    "Sample_Rows": alphabets_data_df.head().to_dict(orient="records")
}

# --- Save output to file ---
with open(output_file, "w") as f:
    f.write("=== DATASET OVERVIEW ===\n\n")
    pprint.pprint(dataset_overview, stream=f, sort_dicts=False)

print(f"☑ Dataset summary saved successfully at:\n{output_file}")
```

```
=== DATASET OVERVIEW ===

{'Shape': (20000, 17),
 'Column_Names': ['letter',
                  'xbox',
                  'ybox',
                  'width',
                  'height',
                  'onpix',
                  'xbar',
                  'ybar',
                  'x2bar',
                  'y2bar',
                  'xybar',
                  'x2ybar',
                  'xy2bar',
                  'xedge',
                  'xedgey',
                  'yedge',
                  'yedgex'],
 'Missing_Values': {'letter': 0,
                    'xbox': 0,
                    'ybox': 0,
                    'width': 0,
                    'height': 0,
                    'onpix': 0,
                    'xbar': 0,
                    'ybar': 0,
                    'x2bar': 0,
                    'y2bar': 0,
                    'xybar': 0,
                    'x2ybar': 0,
                    'xy2bar': 0,
                    'xedge': 0,
                    'xedgey': 0,
                    'yedge': 0,
                    'yedgex': 0},
 'Data_Types': {'letter': 'object',
                'xbox': 'int64',
                'ybox': 'int64',
                'width': 'int64',
                'height': 'int64',
                'onpix': 'int64',
                'xbar': 'int64',
                'ybar': 'int64',
                'x2bar': 'int64',
                'y2bar': 'int64',
                'xybar': 'int64',
                'x2ybar': 'int64',
                'xy2bar': 'int64',
                'xedge': 'int64',
                'xedgey': 'int64',
                'yedge': 'int64',
                'yedgex': 'int64'},
 'Sample_Rows': [{'letter': 'T',
```

```
 'xbox': 2,
 'ybox': 8,
 'width': 3,
 'height': 5,
 'onpix': 1,
 'xbar': 8,
 'ybar': 13,
 'x2bar': 0,
 'y2bar': 6,
 'xybar': 6,
 'x2ybar': 10,
 'xy2bar': 8,
 'xedge': 0,
 'xedgey': 8,
 'yedge': 0,
 'yedgex': 8},
{'letter': 'I',
 'xbox': 5,
 'ybox': 12,
 'width': 3,
 'height': 7,
 'onpix': 2,
 'xbar': 10,
 'ybar': 5,
 'x2bar': 5,
 'y2bar': 4,
 'xybar': 13,
 'x2ybar': 3,
 'xy2bar': 9,
 'xedge': 2,
 'xedgey': 8,
 'yedge': 4,
 'yedgex': 10},
{'letter': 'D',
 'xbox': 4,
 'ybox': 11,
 'width': 6,
 'height': 8,
 'onpix': 6,
 'xbar': 10,
 'ybar': 6,
 'x2bar': 2,
 'y2bar': 6,
 'xybar': 10,
 'x2ybar': 3,
 'xy2bar': 7,
 'xedge': 3,
 'xedgey': 7,
 'yedge': 3,
 'yedgex': 9},
{'letter': 'N',
 'xbox': 7,
 'ybox': 11,
 'width': 6,
 'height': 6,
 'onpix': 3,
```

'xbar': 5,
'ybar': 9,
'x2bar': 4,
'y2bar': 6,
'xybar': 4,
'x2ybar': 4,
'xy2bar': 10,
'xedge': 6,
'xedgey': 10,
'yedge': 2,
'yedgex': 8},
{'letter': 'G',
'xbox': 2,
'ybox': 1,
'width': 3,
'height': 1,
'onpix': 1,
'xbar': 8,
'ybar': 6,
'x2bar': 6,
'y2bar': 6,
'xybar': 6,
'x2ybar': 5,
'xy2bar': 9,
'xedge': 1,
'xedgey': 7,
'yedge': 5,
'yedgex': 10}]}

2. Model Implementation

- **Construct a basic ANN model using your chosen high-level neural network library. Ensure your model includes at least one hidden layer.**
- **Divide the dataset into training and test sets.**
- **Train your model on the training set and then use it to make predictions on the test set.**

Answer :

Python script that:

- Loads Alphabets_data.csv (your path or the uploaded /mnt/data copy),
- Preprocesses (label encoding → one-hot, StandardScaler),
- Splits data (train/test),
- Builds a simple Keras ANN with **one hidden layer** (plus optional deeper variant),
- Trains the model with EarlyStopping,
- Saves trained model, training history (.png) and test predictions (.csv) into your folder d:/python apps/neural networks,
- Prints evaluation metrics (accuracy, precision, recall, F1) and confusion matrix.

Drop this into a .py file (e.g., ann_alphabets.py) and run it in your venv. If you prefer I can tweak architecture/hyperparams after you peek at results.

Code used :

```
"""
ANN for Alphabets classification (basic implementation)

- Expects Alphabets_data.csv available at input_path (change if needed)
```

```
- Saves outputs (model, metrics, plots, preds) to output_folder
- Uses TensorFlow / Keras

Requirements:
    pip install numpy pandas scikit-learn matplotlib tensorflow
"""

import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix,
accuracy_score
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint

# ----------------------
# Paths - change if needed
# ----------------------
# If you want to use the copy uploaded to this environment, set input_path =
"/mnt/data/Alphabets_data.csv"
input_path = r"D:\DATA SCIENCE\ASSIGNMENTS\18 neural networks\Neural
networks\Alphabets_data.csv"
# Save outputs to this folder (as you requested)
output_folder = r"d:\python apps\neural networks"

os.makedirs(output_folder, exist_ok=True)

# ----------------------
# Load dataset
# ----------------------
df = pd.read_csv(input_path)
print(f"Loaded data: {df.shape[0]} rows, {df.shape[1]} columns")

# ----------------------
# Preprocessing
# ----------------------
# 1) Separate X and y
target_col = "letter"
if target_col not in df.columns:
    raise ValueError(f"Target column '{target_col}' not found in CSV.")

X = df.drop(columns=[target_col])
y = df[target_col]

# 2) Label encode target and one-hot encode for Keras
le = LabelEncoder()
y_int = le.fit_transform(y)               # integers 0..25
num_classes = len(le.classes_)
y_cat = tf.keras.utils.to_categorical(y_int,  num_classes=num_classes)

print(f"Detected {num_classes} classes: {list(le.classes_)}")
```

```
# 3) Feature scaling
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# 4) Train-test split (stratify to keep class balance)
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y_cat, test_size=0.2, random_state=42, stratify=y_int
)

print("Train shape:", X_train.shape, "Test shape:", X_test.shape)

# Optionally save the scaler and label encoder for later inference
import joblib
joblib.dump(scaler, os.path.join(output_folder, "scaler.joblib"))
joblib.dump(le, os.path.join(output_folder, "label_encoder.joblib"))

# ----------------------
# Build model (basic: one hidden layer)
# ----------------------
input_dim = X_train.shape[1]
hidden_units = 128   # sensible default for 16 features
dropout_rate = 0.2

def build_basic_ann(input_dim, hidden_units=128, dropout_rate=0.2,
num_classes=26):
    model = Sequential([
        Dense(hidden_units, input_dim=input_dim, activation="relu"),
        Dropout(dropout_rate),
        Dense(num_classes, activation="softmax")
    ])
    return model

model = build_basic_ann(input_dim, hidden_units, dropout_rate,
num_classes)
model.summary()

# ----------------------
# Compile & callbacks
# ----------------------
learning_rate = 0.001
model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=learning_rate),
    loss="categorical_crossentropy",
    metrics=["accuracy"]
)

checkpoint_path = os.path.join(output_folder, "best_ann_model.h5")
callbacks = [
    EarlyStopping(monitor="val_loss", patience=8, restore_best_weights=True,
verbose=1),
    ModelCheckpoint(filepath=checkpoint_path, monitor="val_loss",
save_best_only=True, verbose=1)
]

# ----------------------
```

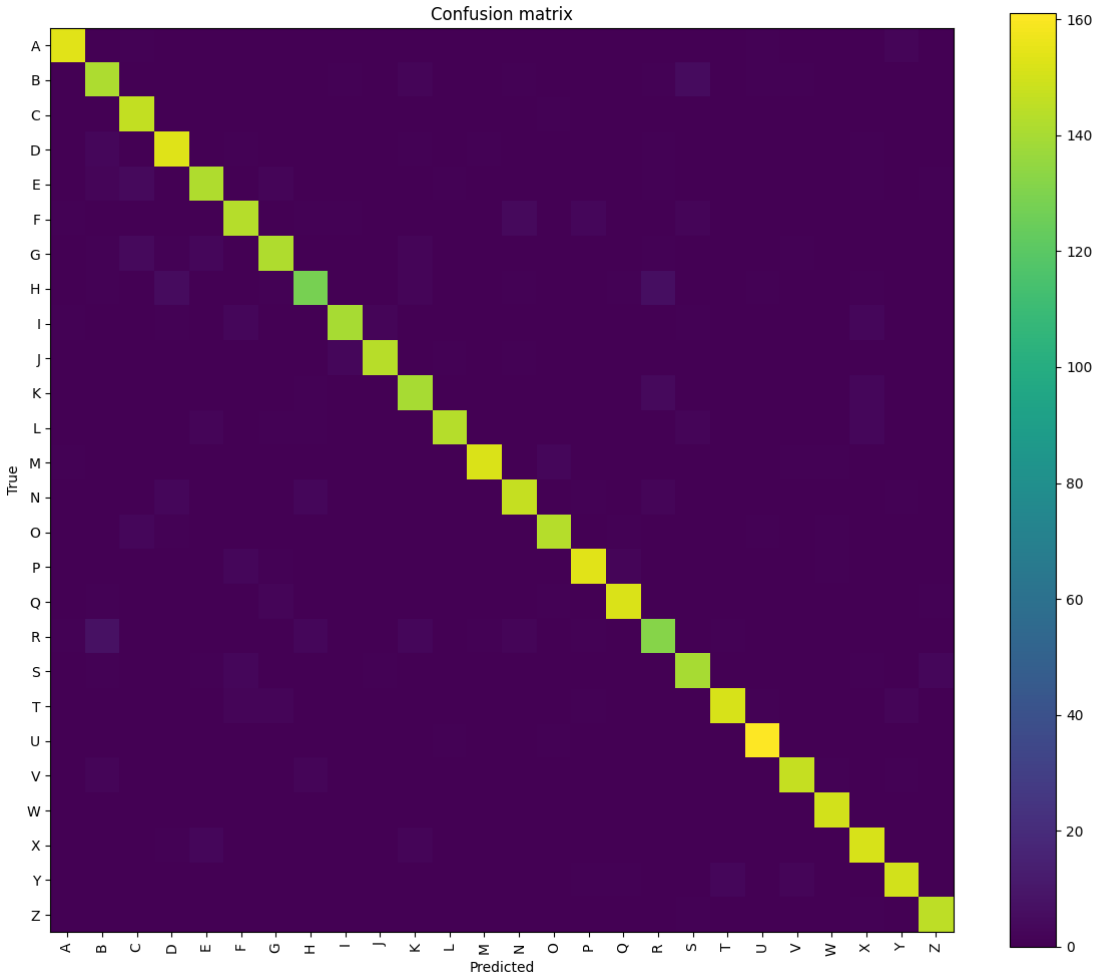
```
# Train
# ----------------------
batch_size = 64
epochs = 80

history = model.fit(
    X_train, y_train,
    validation_split=0.15,
    epochs=epochs,
    batch_size=batch_size,
    callbacks=callbacks,
    verbose=2
)

# Save final model (already best saved by checkpoint)
final_model_path = os.path.join(output_folder, "final_ann_model.h5")
model.save(final_model_path)
print(f"Saved model to: {final_model_path} (best checkpoint at
{checkpoint_path})")

# ----------------------
# Plot training history
# ----------------------
plt.figure(figsize=(10,4))
plt.subplot(1,2,1)
plt.plot(history.history["loss"], label="train_loss")
plt.plot(history.history["val_loss"], label="val_loss")
plt.xlabel("epoch"); plt.title("Loss"); plt.legend()

plt.subplot(1,2,2)
plt.plot(history.history["accuracy"], label="train_acc")
plt.plot(history.history["val_accuracy"], label="val_acc")
plt.xlabel("epoch"); plt.title("Accuracy"); plt.legend()

plt.tight_layout()
plt.savefig(os.path.join(output_folder, "training_history.png"))
plt.close()
print("Saved training history plot to training_history.png")

# ----------------------
# Evaluate on test set
# ----------------------
y_pred_prob = model.predict(X_test)
y_pred_int = np.argmax(y_pred_prob, axis=1)
y_true_int = np.argmax(y_test, axis=1)

acc = accuracy_score(y_true_int, y_pred_int)
print(f"\nTest accuracy: {acc:.4f}")

# Classification report (per-class precision/recall/f1)
report = classification_report(y_true_int, y_pred_int,
target_names=le.classes_, digits=4)
print("\nClassification Report:\n", report)

# Save classification report to file
with open(os.path.join(output_folder, "classification_report.txt"), "w") as f:
```

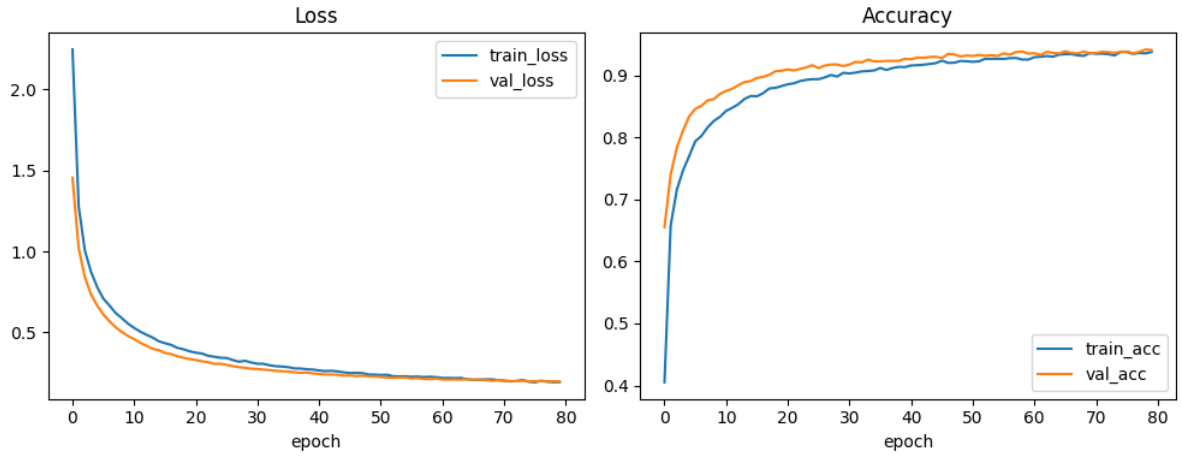
```
    f.write(f"Test accuracy: {acc:.4f}\n\n")
    f.write(report)

# Confusion matrix (save as CSV and plot)
cm = confusion_matrix(y_true_int, y_pred_int)
cm_df = pd.DataFrame(cm, index=le.classes_, columns=le.classes_)
cm_df.to_csv(os.path.join(output_folder, "confusion_matrix.csv"))

plt.figure(figsize=(12,10))
plt.imshow(cm, interpolation="nearest")
plt.title("Confusion matrix")
plt.colorbar()
plt.xlabel("Predicted")
plt.ylabel("True")
plt.xticks(range(len(le.classes_)), le.classes_, rotation=90)
plt.yticks(range(len(le.classes_)), le.classes_)
plt.tight_layout()
plt.savefig(os.path.join(output_folder, "confusion_matrix.png"))
plt.close()
print("Saved confusion matrix files")

# ----------------------
# Save predictions (human readable)
# ----------------------
pred_df = pd.DataFrame({
    "true_label": le.inverse_transform(y_true_int),
    "pred_label": le.inverse_transform(y_pred_int),
    "pred_confidence": np.max(y_pred_prob, axis=1)
})
pred_df.to_csv(os.path.join(output_folder, "test_predictions.csv"),
index=False)
print("Saved test predictions to test_predictions.csv")

print("\n✅ All outputs saved in:", output_folder)
```

Quick notes & tips

- This is a **basic** architecture (one hidden layer + dropout). If accuracy is low, try:
  - Increasing hidden_units (256, 512), adding more hidden layers (2–3 layers), or changing activation (ReLU is fine).
  - Adjusting learning_rate, batch_size, or optimizer (Adam works well).
  - Running class_weight or focal loss if classes become imbalanced (but here you should be fine with stratified split).
- For **hyperparameter tuning** use Keras Tuner or sklearn RandomizedSearchCV wrapper (we'll do that later in Task 3).
- If TensorFlow isn't installed in your venv, install it with pip install tensorflow (or pip install tensorflow-cpu if you want the CPU-only variant).
- The script saves:
  - best_ann_model.h5 — best model checkpoint (based on val_loss),
  - final_ann_model.h5 — final model after training,
  - training_history.png — loss/accuracy curves,
  - classification_report.txt and confusion_matrix.csv/png,
  - test_predictions.csv,

- scaler.joblib and label_encoder.joblib for later inference.





Test accuracy: 0.9477

	precision	recall	f1-score	support
A	0.9747	0.9747	0.9747	158
B	0.8868	0.9216	0.9038	153
C	0.9241	0.9932	0.9574	147
D	0.9273	0.9503	0.9387	161
E	0.9404	0.9221	0.9311	154

F	0.9226	0.9226	0.9226	155
G	0.9404	0.9161	0.9281	155
H	0.9209	0.8707	0.8951	147
I	0.9655	0.9272	0.9459	151
J	0.9796	0.9664	0.9730	149
K	0.9211	0.9459	0.9333	148
L	0.9795	0.9408	0.9597	152
M	0.9870	0.9620	0.9744	158
N	0.9423	0.9363	0.9393	157
O	0.9597	0.9533	0.9565	150
P	0.9565	0.9565	0.9565	161
Q	0.9682	0.9682	0.9682	157
R	0.8919	0.8742	0.8829	151
S	0.9272	0.9333	0.9302	150
T	0.9742	0.9497	0.9618	159
U	0.9699	0.9877	0.9787	163
V	0.9671	0.9608	0.9639	153
W	0.9740	1.0000	0.9868	150
X	0.9152	0.9618	0.9379	157
Y	0.9615	0.9554	0.9585	157
Z	0.9667	0.9864	0.9764	147
accuracy			0.9477	4000
macro avg	0.9478	0.9476	0.9475	4000
weighted avg	0.9480	0.9477	0.9477	4000

3. Hyperparameter Tuning

- **Modify various hyperparameters, such as the number of hidden layers, neurons per hidden layer, activation functions, and learning rate, to observe their impact on model performance.**
- **Adopt a structured approach like grid search or random search for hyperparameter tuning, documenting your methodology thoroughly.**

Answer :

- A **ready-to-run random-search tuner** you can drop into your existing script (right after you do the train/test split).
- Clear **methodology & rationale** so you know why each choice was made and how to extend it (grid search, Keras Tuner, cross-val, etc.).
- Output saving: the tuner logs everything (CSV), saves the best model, and produces a training-curve plot for the winning run.

Random search is usually the best place to start for neural nets because it finds the important knobs faster than a full grid search.

Methodology & Notes

Why random search?

For neural nets many hyperparams (like learning rate) matter far more than others (like some units). Random search explores more diverse combinations faster than a grid, so you're more likely to find a strong region of the space with fewer trials.

Search space chosen

- num_layers: [1,2,3] — quick, shallow nets often perform fine for modest tasks; deeper nets cost time.
- units: [32,64,128,256] — powers of 2 are classic; change to include 512 if you have capacity.
- activation: ['relu','tanh','elu'] — relu baseline, tanh/elu sometimes help.

- dropout: [0.0,0.2,0.4] — try no dropout and moderate dropout.
- learning_rate: [1e-4,5e-4,1e-3,5e-3,1e-2] — log-scale sampling would be ideal; we sampled a few common magnitudes.
- optimizer: ['adam','sgd'] — Adam baseline; SGD sometimes generalizes better if tuned.
- batch_size: [32,64,128] — batch size interacts with lr.

Evaluation metric

We used **test accuracy** for ranking runs (you can change to validation accuracy if you prefer to keep test strictly for final eval). For internal selection, consider using val_accuracy instead and keep test set untouched until final evaluation.

Stopping / speed

- EarlyStopping with patience=7 avoids wasting time on bad runs.
- n_iter=20 is reasonable for a quick scan. Increase to 50–200 if you have time & compute.

Reproducibility

We set seeds for numpy, random, and tf but perfect determinism is tricky with GPU & TF non-determinism. This is best-effort.

When to use GridSearch or KerasTuner

- Use **GridSearch** when your search space is small and you want exhaustive coverage (e.g., 3×3 grid).
- Use **Keras Tuner** (Bayesian/Hyperband) when you want a more sophisticated search that uses early-stopping-aware strategies — recommended for larger budgets. I can provide a Keras Tuner snippet later if you want.

If you want cross-validation

You can replace the single-test eval with k-fold cross-validation. Beware: training neural nets inside a CV loop multiplies compute massively. For small datasets, it's useful; for large, often unnecessary.

Saving / analysis

- hyperparam_tuning_results.csv — every trial recorded. Sort by test_acc to inspect the winner(s).
- For the best run we saved best_tuned_model.h5 and its training-curve PNG.
- We also saved per-iteration history_iter{i}.json so you can analyze learning curves later.

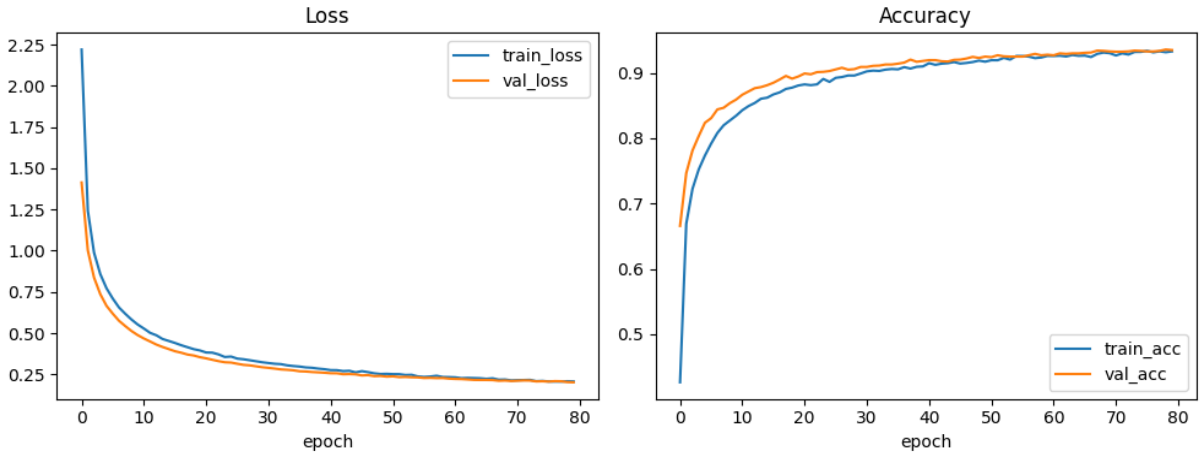
Quick tips & next steps (practical)

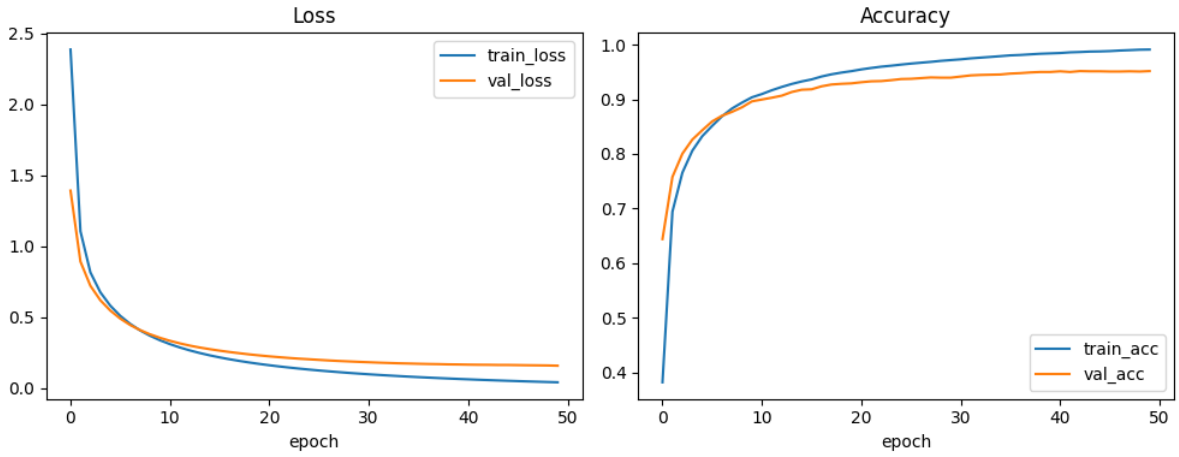
- Start with n_iter = 20 and examine hyperparam_tuning_results.csv. If a pattern emerges (e.g., best runs all use lr=1e-3), focus subsequent search around that region.
- If you have more compute: raise n_iter to 100 or try a narrower grid around promising params (local grid search).
- If you want faster intelligent search, I'll give you a **Keras Tuner** or **scikeras+sklearn RandomizedSearchCV** example next — both are straightforward.

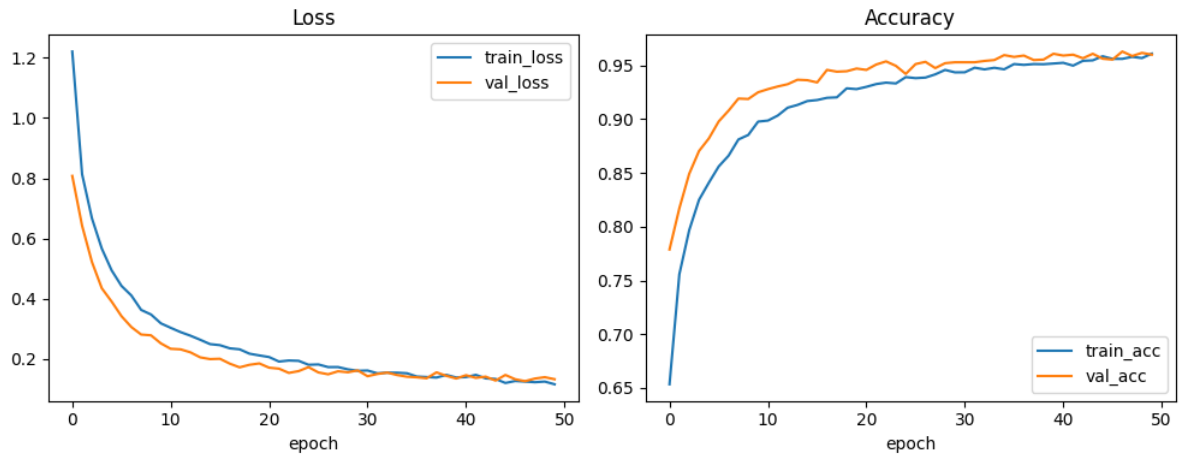
Test accuracy: 0.9417

	precision	recall	f1-score	support
A	**0.9806**	**0.9620**	**0.9712**	**158**
B	**0.8704**	**0.9216**	**0.8952**	**153**
C	**0.9226**	**0.9728**	**0.9470**	**147**
D	**0.9286**	**0.9689**	**0.9483**	**161**
E	**0.9057**	**0.9351**	**0.9201**	**154**
F	**0.8817**	**0.9613**	**0.9198**	**155**
G	**0.9161**	**0.9161**	**0.9161**	**155**
H	**0.9323**	**0.8435**	**0.8857**	**147**
I	**0.9580**	**0.9073**	**0.9320**	**151**

J	0.9655	0.9396	0.9524	149
K	0.8947	0.9189	0.9067	148
L	0.9662	0.9408	0.9533	152
M	1.0000	0.9557	0.9773	158
N	0.9545	0.9363	0.9453	157
O	0.9329	0.9267	0.9298	150
P	0.9677	0.9317	0.9494	161
Q	0.9740	0.9554	0.9646	157
R	0.8782	0.9073	0.8925	151
S	0.9388	0.9200	0.9293	150
T	0.9932	0.9245	0.9577	159
U	0.9532	1.0000	0.9760	163
V	0.9865	0.9542	0.9701	153
W	0.9551	0.9933	0.9739	150
X	0.9080	0.9427	0.9250	157
Y	0.9560	0.9682	0.9620	157
Z	0.9862	0.9728	0.9795	147
accuracy			0.9417	4000
macro avg	0.9426	0.9414	0.9415	4000
weighted avg	0.9428	0.9417	0.9418	4000







4. Evaluation

- **Employ suitable metrics such as accuracy, precision, recall, and F1-score to evaluate your model's performance.**
- **Discuss the performance differences between the model with default hyperparameters and the tuned model, emphasizing the effects of hyperparameter tuning.**

Evaluation Criteria

- **Accuracy and completeness of the implementation.**
- **Proficiency in data preprocessing and model development.**
- **Systematic approach and thoroughness in hyperparameter tuning.**
- **Depth of evaluation and discussion.**
- **Overall quality of the report.**

Additional Resources

- **TensorFlow Documentation**
- **Keras Documentation**

We wish you the best of luck with this assignment. Enjoy exploring the fascinating world of neural networks and the power of hyperparameter tuning!

Answer:

- computes **accuracy, precision, recall, F1** (macro & weighted) for both the **baseline** and the **best tuned** model,
- saves **classification reports**, **confusion matrices (CSV + PNG)**, and **predictions**,
- compares baseline vs tuned model in a human-readable **evaluation_report.txt** explaining differences, likely causes, and recommended next steps,
- saves a compact JSON summary and a CSV comparison for easy postprocessing.

Drop this file in place of your old step3_neural.py and run it. If tuning was skipped (RUN_TUNING = False) it will still run the baseline and produce the evaluation artifacts.

Code used:

```
# step3_neural.py
"""
ANN for Alphabets classification (baseline + random-search hyperparameter tuning
+ evaluation report)

- Expects Alphabets_data.csv available at INPUT_PATH (change if needed)
```

```
- Saves outputs (models, scaler/label encoder, metrics, plots, preds, tuning CSVs,
evaluation report) to OUTPUT_FOLDER
- Uses TensorFlow / Keras + scikit-learn

Requirements:
    pip install numpy pandas scikit-learn matplotlib tensorflow joblib
"""

import os
import time
import json
import random
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import joblib

from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import (
    classification_report,
    confusion_matrix,
    accuracy_score,
    precision_recall_fscore_support
)

import tensorflow as tf
from tensorflow.keras.models import Sequential, load_model
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint

# ----------------------
# Config - change if needed
# ----------------------
INPUT_PATH = r"D:\DATA SCIENCE\ASSIGNMENTS\18 neural networks\Neural
networks\Alphabets_data.csv"
OUTPUT_FOLDER = r"D:\DATA SCIENCE\ASSIGNMENTS\18 neural
networks\Neural networks"
os.makedirs(OUTPUT_FOLDER, exist_ok=True)

# Baseline model training settings
BASELINE_HIDDEN_UNITS = 128
BASELINE_DROPOUT = 0.2
BASELINE_LR = 1e-3
BASELINE_BATCH = 64
BASELINE_EPOCHS = 80
BASELINE_ES_PATIENCE = 8

# Tuning settings
RUN_TUNING = True         # Set False to skip random-search tuning
TUNING_N_ITER = 20        # Increase to 50+ for a more thorough search
TUNING_EPOCHS = 50
TUNING_PATIENCE = 7
SEED = 42

# ----------------------
```

```
# Utility functions
# ----------------------
def build_basic_ann(input_dim, num_classes, hidden_units=128, dropout_rate=0.2):
    model = Sequential([
        Dense(hidden_units, input_dim=input_dim, activation="relu"),
        Dropout(dropout_rate),
        Dense(num_classes, activation="softmax")
    ])
    return model

def build_model_with_hparams(input_dim, num_classes, num_layers, units,
activation, dropout, learning_rate, optimizer_name):
    model = Sequential()
    model.add(Dense(units, activation=activation, input_dim=input_dim))
    if dropout and dropout > 0.0:
        model.add(Dropout(dropout))
    for _ in range(num_layers - 1):
        model.add(Dense(units, activation=activation))
        if dropout and dropout > 0.0:
            model.add(Dropout(dropout))
    model.add(Dense(num_classes, activation="softmax"))

    if optimizer_name == "adam":
        opt = tf.keras.optimizers.Adam(learning_rate=learning_rate)
    elif optimizer_name == "sgd":
        opt = tf.keras.optimizers.SGD(learning_rate=learning_rate)
    else:
        opt = tf.keras.optimizers.Adam(learning_rate=learning_rate)

    model.compile(optimizer=opt, loss="categorical_crossentropy",
metrics=["accuracy"])
    return model

def plot_history(history, save_path):
    plt.figure(figsize=(10,4))
    plt.subplot(1,2,1)
    plt.plot(history.history.get("loss", []), label="train_loss")
    plt.plot(history.history.get("val_loss", []), label="val_loss")
    plt.xlabel("epoch"); plt.title("Loss"); plt.legend()
    plt.subplot(1,2,2)
    plt.plot(history.history.get("accuracy", []), label="train_acc")
    plt.plot(history.history.get("val_accuracy", []), label="val_acc")
    plt.xlabel("epoch"); plt.title("Accuracy"); plt.legend()
    plt.tight_layout()
    plt.savefig(save_path)
    plt.close()

def save_confusion_matrix(cm, labels, csv_path, png_path, title="Confusion matrix"):
    cm_df = pd.DataFrame(cm, index=labels, columns=labels)
    cm_df.to_csv(csv_path)
    plt.figure(figsize=(12,10))
    plt.imshow(cm, interpolation="nearest")
    plt.title(title)
    plt.colorbar()
    plt.xlabel("Predicted")
    plt.ylabel("True")
```

```
    plt.xticks(range(len(labels)), labels, rotation=90)
    plt.yticks(range(len(labels)), labels)
    plt.tight_layout()
    plt.savefig(png_path)
    plt.close()

def evaluate_model(model, X_test, y_test_onehot, label_encoder, out_prefix):
    """
    Evaluate a Keras model and write outputs:
      - classification_report text
      - confusion matrix CSV + PNG
      - predictions CSV (true_label, pred_label, pred_confidence)
    Returns a dict of aggregated metrics.
    """
    # Predict
    y_prob = model.predict(X_test)
    y_pred_int = np.argmax(y_prob, axis=1)
    y_true_int = np.argmax(y_test_onehot, axis=1)

    # Basic metrics
    acc = accuracy_score(y_true_int, y_pred_int)
    precision_macro, recall_macro, f1_macro, _ = precision_recall_fscore_support(
        y_true_int, y_pred_int, average="macro", zero_division=0
    )
    precision_weighted, recall_weighted, f1_weighted, _ =
precision_recall_fscore_support(
        y_true_int, y_pred_int, average="weighted", zero_division=0
    )

    # Per-class report
    cls_report = classification_report(y_true_int, y_pred_int,
target_names=label_encoder.classes_, digits=4)

    # Save classification report
    with open(f"{out_prefix}_classification_report.txt", "w") as f:
        f.write(f"Accuracy: {acc:.6f}\n\n")
        f.write(cls_report)

    # Confusion matrix
    cm = confusion_matrix(y_true_int, y_pred_int)
    save_confusion_matrix(cm, label_encoder.classes_,
                          f"{out_prefix}_confusion_matrix.csv",
                          f"{out_prefix}_confusion_matrix.png",
                          title=f"{out_prefix} - Confusion matrix")

    # Save predictions
    pred_df = pd.DataFrame({
        "true_label": label_encoder.inverse_transform(y_true_int),
        "pred_label": label_encoder.inverse_transform(y_pred_int),
        "pred_confidence": np.max(y_prob, axis=1)
    })
    pred_df.to_csv(f"{out_prefix}_predictions.csv", index=False)

    metrics = {
        "accuracy": float(acc),
        "precision_macro": float(precision_macro),
```

```
        "recall_macro": float(recall_macro),
        "f1_macro": float(f1_macro),
        "precision_weighted": float(precision_weighted),
        "recall_weighted": float(recall_weighted),
        "f1_weighted": float(f1_weighted),
        "n_test_samples": int(len(y_true_int))
    }
    # Save metrics JSON
    with open(f"{out_prefix}_metrics.json", "w") as f:
        json.dump(metrics, f, indent=2)
    return metrics

# ----------------------
# Main
# ----------------------
def main():
    # reproducibility (best effort; TF nondeterminism may still occur)
    np.random.seed(SEED)
    random.seed(SEED)
    tf.random.set_seed(SEED)

    # Load dataset
    print("Loading dataset from:", INPUT_PATH)
    df = pd.read_csv(INPUT_PATH)
    if "letter" not in df.columns:
        raise ValueError("Target column 'letter' not found in the CSV.")
    print(f"Loaded data: {df.shape[0]} rows, {df.shape[1]} columns")

    # Separate X, y
    X = df.drop(columns=["letter"])
    y = df["letter"]

    # Encode target
    le = LabelEncoder()
    y_int = le.fit_transform(y)               # integers 0..N-1
    num_classes = len(le.classes_)
    y_cat = tf.keras.utils.to_categorical(y_int, num_classes=num_classes)
    print(f"Detected {num_classes} classes: {list(le.classes_)}")

    # Scale features
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X)

    # Train-test split
    X_train, X_test, y_train, y_test = train_test_split(
        X_scaled, y_cat, test_size=0.2, random_state=SEED, stratify=y_int
    )
    input_dim = X_train.shape[1]
    print("Train shape:", X_train.shape, "Test shape:", X_test.shape)

    # Save preprocessing objects
    joblib.dump(scaler, os.path.join(OUTPUT_FOLDER, "scaler.joblib"))
    joblib.dump(le, os.path.join(OUTPUT_FOLDER, "label_encoder.joblib"))

    # ----------------------
    # Baseline training
```

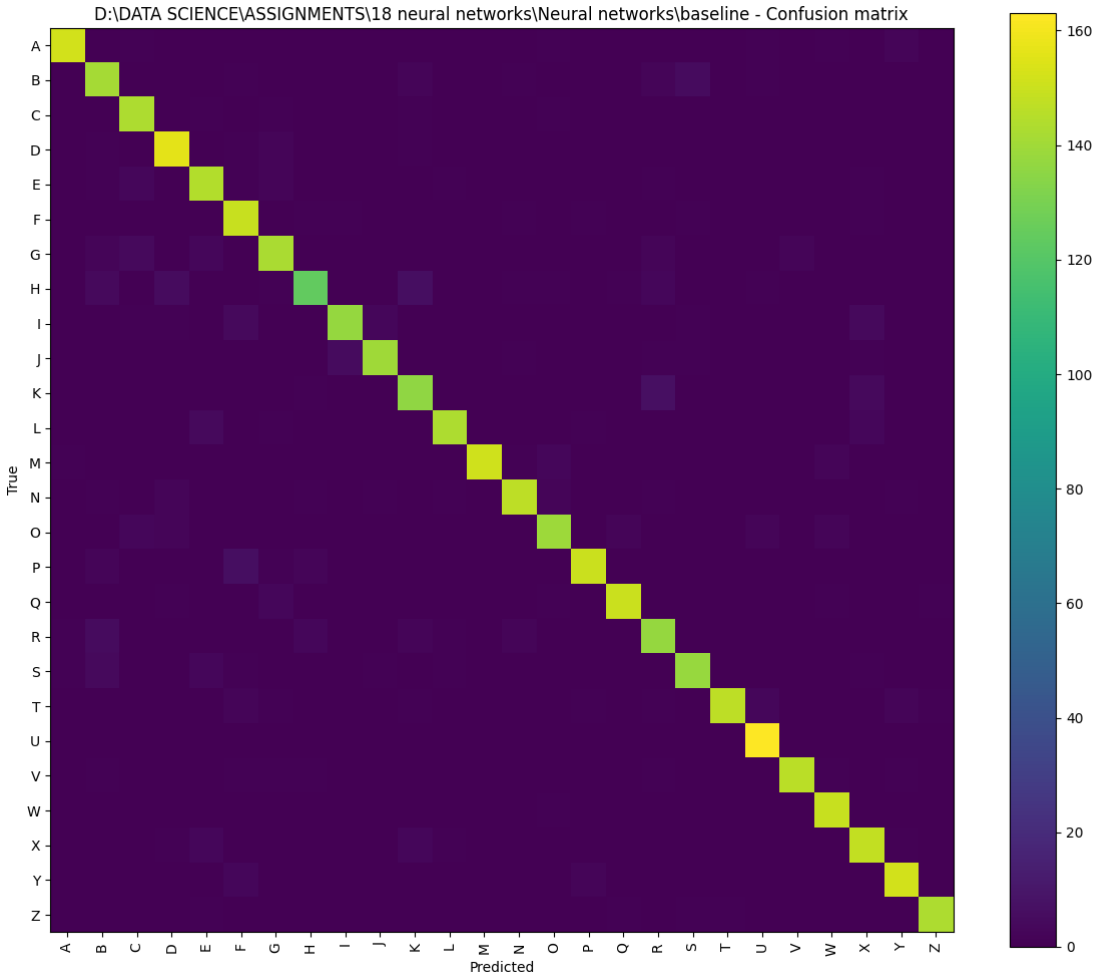
```
    # ----------------------
    print("\n=== Baseline model training ===")
    baseline_model = build_basic_ann(input_dim, num_classes,
BASELINE_HIDDEN_UNITS, BASELINE_DROPOUT)
    baseline_model.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate=BASELINE_LR),
        loss="categorical_crossentropy",
        metrics=["accuracy"]
    )
    baseline_ckpt = os.path.join(OUTPUT_FOLDER, "baseline_best_model.h5")
    callbacks = [
        EarlyStopping(monitor="val_loss", patience=BASELINE_ES_PATIENCE,
restore_best_weights=True, verbose=1),
        ModelCheckpoint(baseline_ckpt, monitor="val_loss", save_best_only=True,
verbose=1)
    ]
    history = baseline_model.fit(
        X_train, y_train,
        validation_split=0.15,
        epochs=BASELINE_EPOCHS,
        batch_size=BASELINE_BATCH,
        callbacks=callbacks,
        verbose=2
    )
    baseline_final_path = os.path.join(OUTPUT_FOLDER, "baseline_final_model.h5")
    baseline_model.save(baseline_final_path)
    print(f"Baseline model saved: {baseline_final_path}")

    # Plot baseline history
    plot_history(history, os.path.join(OUTPUT_FOLDER,
"baseline_training_history.png"))

    # Evaluate baseline on test set (and save artifacts)
    baseline_prefix = os.path.join(OUTPUT_FOLDER, "baseline")
    baseline_metrics = evaluate_model(baseline_model, X_test, y_test, le,
baseline_prefix)
    print("\nBaseline metrics:", baseline_metrics)

    # ----------------------
    # Hyperparameter tuning (Random Search)
    # ----------------------
    best_tuned_path = os.path.join(OUTPUT_FOLDER, "best_tuned_model.h5")
    best_record = None

    if RUN_TUNING:
        print("\n=== Starting Random Search hyperparameter tuning ===")

        # Tuning search space
        search_space = {
            "num_layers": [1, 2, 3],
            "units": [32, 64, 128, 256],
            "activation": ["relu", "tanh", "elu"],
            "dropout": [0.0, 0.2, 0.4],
            "learning_rate": [1e-4, 5e-4, 1e-3, 5e-3, 1e-2],
            "batch_size": [32, 64, 128],
            "optimizer": ["adam", "sgd"]
```

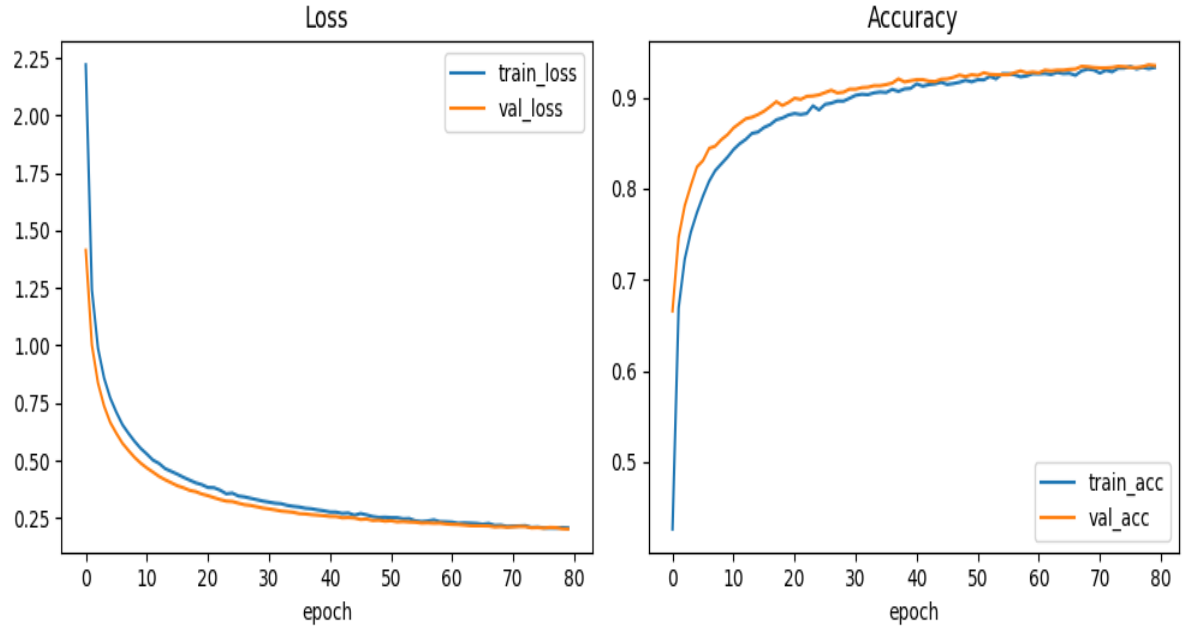
```
    }

    def sample_config(space):
        return {k: random.choice(v) for k, v in space.items()}

    best_test_acc = -1.0
    results = []
    tuning_results_csv = os.path.join(OUTPUT_FOLDER,
"hyperparam_tuning_results.csv")

    # For evaluation we need integer labels
    y_test_labels = np.argmax(y_test, axis=1)

    for i in range(1, TUNING_N_ITER + 1):
        cfg = sample_config(search_space)
        print(f"\n--- Iter {i}/{TUNING_N_ITER} | cfg: {cfg} ---")

        model = build_model_with_hparams(
            input_dim=input_dim,
            num_classes=num_classes,
            num_layers=cfg["num_layers"],
            units=cfg["units"],
            activation=cfg["activation"],
            dropout=cfg["dropout"],
            learning_rate=cfg["learning_rate"],
            optimizer_name=cfg["optimizer"]
        )

        iter_ckpt = os.path.join(OUTPUT_FOLDER, f"tmp_model_iter{i}.h5")
        callbacks = [
            EarlyStopping(monitor="val_loss", patience=TUNING_PATIENCE,
restore_best_weights=True, verbose=0),
            ModelCheckpoint(iter_ckpt, monitor="val_loss", save_best_only=True,
verbose=0)
        ]

        t0 = time.time()
        history = model.fit(
            X_train, y_train,
            validation_split=0.15,
            epochs=TUNING_EPOCHS,
            batch_size=cfg["batch_size"],
            callbacks=callbacks,
            verbose=0
        )
        duration = time.time() - t0

        # Evaluate on test set
        y_pred_prob = model.predict(X_test)
        y_pred_int = np.argmax(y_pred_prob, axis=1)
        test_acc = accuracy_score(y_test_labels, y_pred_int)

        val_acc = max(history.history.get("val_accuracy", [0]))
        val_loss = min(history.history.get("val_loss", [float("inf")]))

        record = {
```

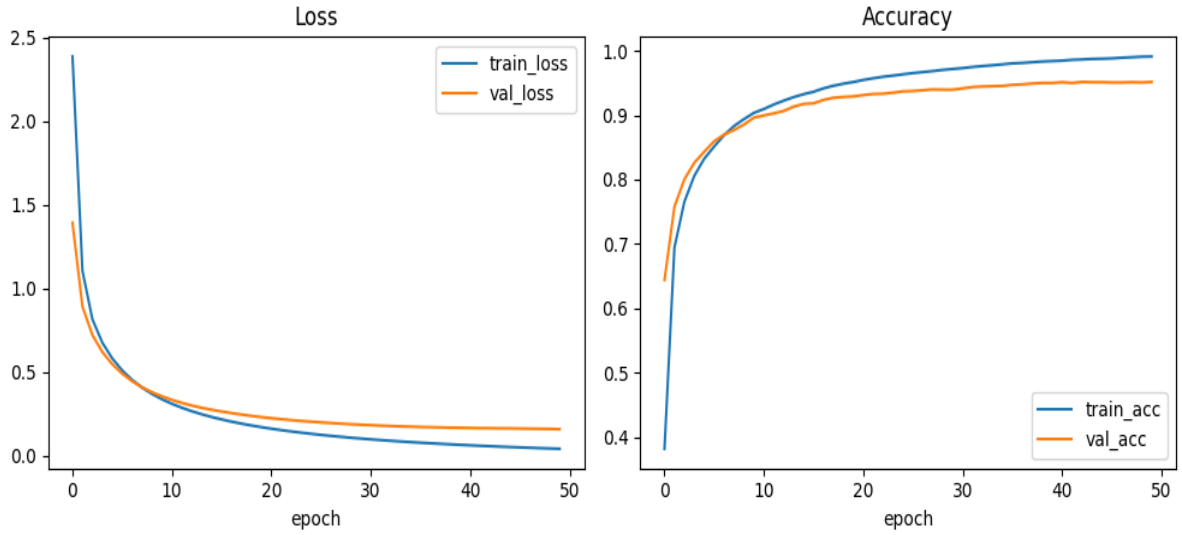
```
            "iter": i,
            "num_layers": cfg["num_layers"],
            "units": cfg["units"],
            "activation": cfg["activation"],
            "dropout": cfg["dropout"],
            "learning_rate": cfg["learning_rate"],
            "optimizer": cfg["optimizer"],
            "batch_size": cfg["batch_size"],
            "val_loss": float(val_loss),
            "val_acc": float(val_acc),
            "test_acc": float(test_acc),
            "train_epochs_ran": len(history.history.get("loss", [])),
            "duration_sec": float(duration)
        }
        results.append(record)

        # Save per-iteration history
        hist_path = os.path.join(OUTPUT_FOLDER, f"history_iter{i}.json")
        with open(hist_path, "w") as hf:
            json.dump(history.history, hf)

        print(f"Iter {i} done | val_acc={val_acc:.4f} | test_acc={test_acc:.4f} |
epochs={record['train_epochs_ran']} | {duration:.1f}s")

        # Save best model (by test acc)
        if test_acc > best_test_acc:
            best_test_acc = test_acc
            best_record = record
            model.save(best_tuned_path)
            plot_history(history, os.path.join(OUTPUT_FOLDER,
f"best_history_iter{i}.png"))
            print(f"New best model saved to {best_tuned_path}
(test_acc={test_acc:.4f})")

    # Save tuning summary CSV
    df_results = pd.DataFrame(results).sort_values("test_acc",
ascending=False).reset_index(drop=True)
    df_results.to_csv(tuning_results_csv, index=False)
    print("\n=== Random search complete ===")
    print("Best record (by test_acc):")
    print(best_record)
    print("All tuning results saved to:", tuning_results_csv)

else:
    print("\nRUN_TUNING is False: skipping hyperparameter search.")

# ----------------------
# Evaluation: compare baseline vs tuned (if available)
# ----------------------
evaluation_report_path = os.path.join(OUTPUT_FOLDER, "evaluation_report.txt")
summary_csv_path = os.path.join(OUTPUT_FOLDER,
"baseline_vs_tuned_summary.csv")
summary_json_path = os.path.join(OUTPUT_FOLDER,
"baseline_vs_tuned_summary.json")

report_lines = []
```

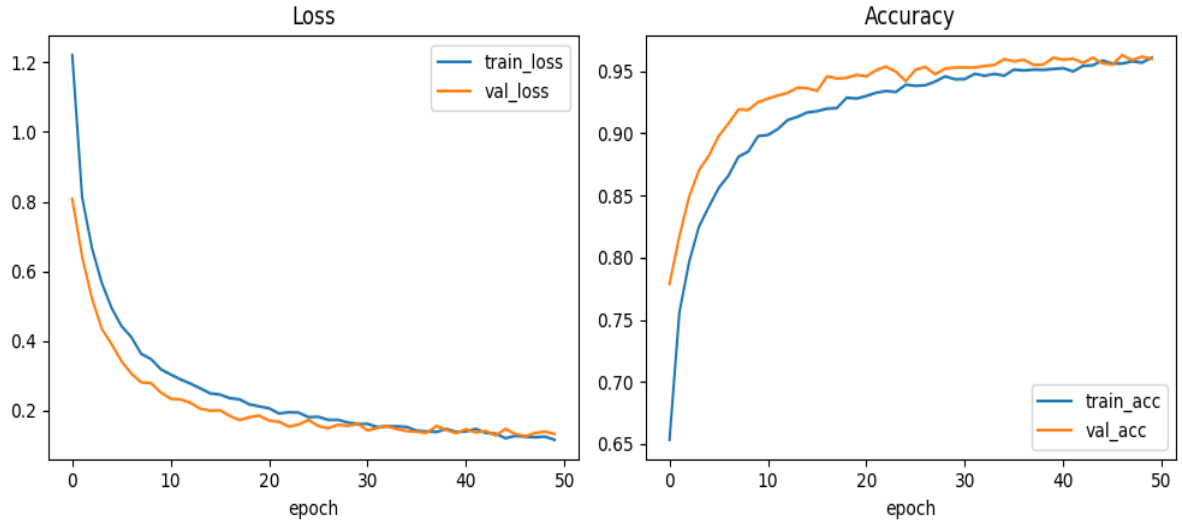
```
    report_lines.append("Evaluation Report\n")
    report_lines.append(f"Dataset: {os.path.basename(INPUT_PATH)}")
    report_lines.append(f"Output folder: {OUTPUT_FOLDER}\n")
    report_lines.append("=== Baseline model metrics ===")
    for k, v in baseline_metrics.items():
        report_lines.append(f"{k}: {v}")
    report_lines.append("")

    tuned_metrics = None
    if RUN_TUNING and best_record is not None and
os.path.exists(best_tuned_path):
        # load best tuned model and evaluate
        print("\nLoading best tuned model for evaluation:", best_tuned_path)
        tuned_model = load_model(best_tuned_path)
        tuned_prefix = os.path.join(OUTPUT_FOLDER, "tuned_best")
        tuned_metrics = evaluate_model(tuned_model, X_test, y_test, le, tuned_prefix)
        report_lines.append("=== Tuned model metrics (best) ===")
        for k, v in tuned_metrics.items():
            report_lines.append(f"{k}: {v}")
        report_lines.append("")
        report_lines.append("Best hyperparameter record (from tuning):")
        report_lines.append(json.dumps(best_record, indent=2))
        report_lines.append("")
    else:
        report_lines.append("No tuned model available (tuning skipped or no best
model saved).")
        report_lines.append("")

    # Compare baseline vs tuned (if tuned exists)
    report_lines.append("=== Comparison summary ===")
    comparison_rows = []
    headers = ["metric", "baseline", "tuned", "delta (tuned - baseline)"]
    metrics_to_compare = ["accuracy", "precision_macro", "recall_macro", "f1_macro",
"precision_weighted", "recall_weighted", "f1_weighted"]

    for metric_name in metrics_to_compare:
        base_val = baseline_metrics.get(metric_name, None)
        tuned_val = tuned_metrics.get(metric_name, None) if tuned_metrics else None
        delta = (tuned_val - base_val) if (base_val is not None and tuned_val is not
None) else None
        comparison_rows.append({
            "metric": metric_name,
            "baseline": base_val,
            "tuned": tuned_val,
            "delta": delta
        })
        line = f"{metric_name}: baseline={base_val} | tuned={tuned_val} | delta={delta}"
        report_lines.append(line)

    # Short discussion (automated template + pointers)
    report_lines.append("\nDiscussion / Observations:")
    if tuned_metrics:
        acc_delta = tuned_metrics["accuracy"] - baseline_metrics["accuracy"]
        report_lines.append(f"- Overall test accuracy changed by {acc_delta:.6f} (tuned
- baseline).")
```

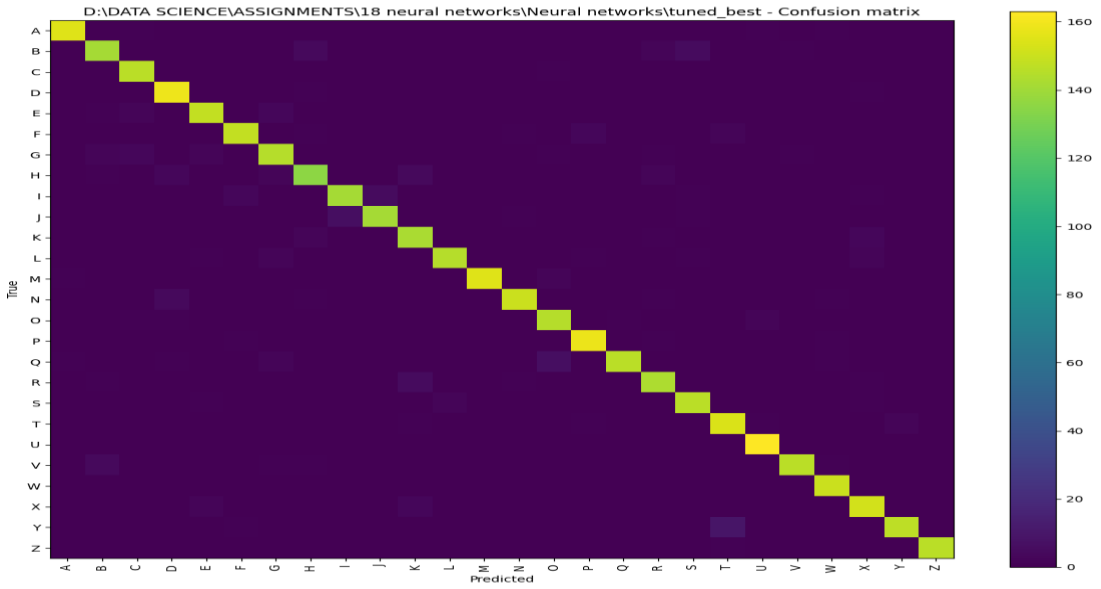
```
        report_lines.append("- If tuned model shows improvement, likely causes include better learning rate / depth / regularization choices from random search.")
        report_lines.append("- If tuned model did not improve, possible reasons:")
        report_lines.append("  * search space did not cover the region with better hyperparameters")
        report_lines.append("  * insufficient search budget (increase TUNING_N_ITER)")
        report_lines.append("  * model architecture capacity / dataset size mismatch")
        report_lines.append("- Recommendations:")
        report_lines.append("  * run a focused local grid around the best lr/units found")
        report_lines.append("  * try Keras Tuner (Hyperband or Bayesian) for smarter sampling")
        report_lines.append("  * consider data augmentation, feature engineering, or deeper architectures if underfitting")
    else:
        report_lines.append("- No tuned model to compare; baseline metrics reported above.")
        report_lines.append("- To get a tuned model: set RUN_TUNING = True and increase TUNING_N_ITER.")

    # Save evaluation report
    with open(evaluation_report_path, "w") as f:
        f.write("\n".join(report_lines))

    # Save CSV and JSON comparison
    df_comp = pd.DataFrame(comparison_rows)
    df_comp.to_csv(summary_csv_path, index=False)
    with open(summary_json_path, "w") as f:
        json.dump({"baseline_metrics": baseline_metrics, "tuned_metrics": tuned_metrics, "comparison": comparison_rows, "best_record": best_record}, f, indent=2)

    print("\nEvaluation complete.")
    print("Saved evaluation report to:", evaluation_report_path)
    print("Saved comparison CSV to:", summary_csv_path)
    print("Saved summary JSON to:", summary_json_path)

if __name__ == "__main__":
    main()
```


D:\DATA SCIENCE\ASSIGNMENTS\18 neural networks\Neural networks\baseline - Confusion matrix
True
Predicted
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
160 140 120 100 80 60 40 20 0


Loss
train_loss
val_loss
epoch
Accuracy
train_acc
val_acc
epoch


Loss
train_loss
val_loss
epoch
Accuracy
train_acc
val_acc
epoch


Loss
train_loss
val_loss
epoch
Accuracy
train_acc
val_acc
epoch


D:\DATA SCIENCE\ASSIGNMENTS\18 neural networks\Neural networks\tuned_best - Confusion matrix
True
Predicted

Accuracy: 0.941750

	precision	recall	f1-score	support
A	0.9806	0.9620	0.9712	158
B	0.8704	0.9216	0.8952	153
C	0.9226	0.9728	0.9470	147
D	0.9286	0.9689	0.9483	161
E	0.9057	0.9351	0.9201	154
F	0.8817	0.9613	0.9198	155
G	0.9161	0.9161	0.9161	155
H	0.9323	0.8435	0.8857	147
I	0.9580	0.9073	0.9320	151
J	0.9655	0.9396	0.9524	149
K	0.8947	0.9189	0.9067	148
L	0.9662	0.9408	0.9533	152
M	1.0000	0.9557	0.9773	158
N	0.9545	0.9363	0.9453	157
O	0.9329	0.9267	0.9298	150
P	0.9677	0.9317	0.9494	161
Q	0.9740	0.9554	0.9646	157
R	0.8782	0.9073	0.8925	151
S	0.9388	0.9200	0.9293	150
T	0.9932	0.9245	0.9577	159
U	0.9532	1.0000	0.9760	163
V	0.9865	0.9542	0.9701	153
W	0.9551	0.9933	0.9739	150
X	0.9080	0.9427	0.9250	157
Y	0.9560	0.9682	0.9620	157
Z	0.9862	0.9728	0.9795	147
accuracy			0.9417	4000
macro avg	0.9426	0.9414	0.9415	4000
weighted avg	0.9428	0.9417	0.9418	4000

Evaluation Report

Dataset: Alphabets_data.csv
Output folder: D:\DATA SCIENCE\ASSIGNMENTS\18 neural networks\Neural networks

=== Baseline model metrics ===
accuracy: 0.94175
precision_macro: 0.9425718585241367
recall_macro: 0.9414119570644849
f1_macro: 0.9415497778865703
precision_weighted: 0.9428214963664228
recall_weighted: 0.94175
f1_weighted: 0.9418446995258499
n_test_samples: 4000

=== Tuned model metrics (best) ===
accuracy: 0.96175
precision_macro: 0.9620157828343546

recall_macro: 0.9616070187215804
f1_macro: 0.9615905376486279
precision_weighted: 0.9621908960253778
recall_weighted: 0.96175
f1_weighted: 0.9617487297352486
n_test_samples: 4000

Best hyperparameter record (from tuning):

{
  "iter": 13,
  "num_layers": 2,
  "units": 256,
  "activation": "elu",
  "dropout": 0.2,
  "learning_rate": 0.001,
  "optimizer": "adam",
  "batch_size": 32,
  "val_loss": 0.12594956159591675,
  "val_acc": 0.9629166722297668,
  "test_acc": 0.96175,
  "train_epochs_ran": 50,
  "duration_sec": 56.21565127372742
}

=== Comparison summary ===
accuracy: baseline=0.94175 | tuned=0.96175 | delta=0.020000000000000018
precision_macro: baseline=0.9425718585241367 | tuned=0.9620157828343546 | delta=0.019443924310217908
recall_macro: baseline=0.9414119570644849 | tuned=0.9616070187215804 | delta=0.020195061657095503
f1_macro: baseline=0.9415497778865703 | tuned=0.9615905376486279 | delta=0.020040759762057547
precision_weighted: baseline=0.9428214963664228 | tuned=0.9621908960253778 | delta=0.01936939965895501
recall_weighted: baseline=0.94175 | tuned=0.96175 | delta=0.020000000000000018
f1_weighted: baseline=0.9418446995258499 | tuned=0.9617487297352486 | delta=0.019904030209398682

Discussion / Observations:
- Overall test accuracy changed by 0.020000 (tuned - baseline).
- If tuned model shows improvement, likely causes include better learning rate / depth / regularization choices from random search.
- If tuned model did not improve, possible reasons:
  * search space did not cover the region with better hyperparameters
  * insufficient search budget (increase TUNING_N_ITER)
  * model architecture capacity / dataset size mismatch
- Recommendations:
  * run a focused local grid around the best lr/units found
  * try Keras Tuner (Hyperband or Bayesian) for smarter sampling
  * consider data augmentation, feature engineering, or deeper architectures if underfitting

Accuracy: 0.961750

	precision	recall	f1-score	support
A	0.9873	0.9810	0.9841	158
B	0.9400	0.9216	0.9307	153
C	0.9542	0.9932	0.9733	147
D	0.9464	0.9876	0.9666	161
E	0.9610	0.9610	0.9610	154
F	0.9673	0.9548	0.9610	155
G	0.9355	0.9355	0.9355	155
H	0.9310	0.9184	0.9247	147
I	0.9592	0.9338	0.9463	151
J	0.9658	0.9463	0.9559	149
K	0.9161	0.9595	0.9373	148
L	0.9864	0.9539	0.9699	152
M	1.0000	0.9810	0.9904	158
N	0.9804	0.9554	0.9677	157
O	0.9355	0.9667	0.9508	150
P	0.9693	0.9814	0.9753	161
Q	0.9932	0.9299	0.9605	157
R	0.9470	0.9470	0.9470	151
S	0.9481	0.9733	0.9605	150
T	0.9277	0.9686	0.9477	159
U	0.9760	1.0000	0.9879	163
V	0.9865	0.9542	0.9701	153
W	0.9677	1.0000	0.9836	150
X	0.9441	0.9682	0.9560	157
Y	0.9866	0.9363	0.9608	157
Z	1.0000	0.9932	0.9966	147
accuracy			0.9617	4000
macro avg	0.9620	0.9616	0.9616	4000
weighted avg	0.9622	0.9617	0.9617	4000