Awesome—here’s your script with **deep, line-by-line comments** so you can understand exactly what every part does, plus short explanations of each library and how the data flows through the pipeline.

Tip: I kept your logic the same so it runs as you expect. I’ve added optional notes (clearly marked) where a small tweak could improve things.

# ================================================

# ASL Alphabet Transfer Learning with MobileNetV2

# ================================================

# Goal:

# - Load images from folders using tf.data

# - Normalize and (optionally) augment images

# - Fine-tune a pre-trained MobileNetV2 backbone

# - Train, evaluate, plot metrics

# - Save + reload model safely under mixed precision

# -----------------------------------------------

# ---- Core scientific stack ----

import numpy as np # Arrays, math ops (not strictly needed in this script, but handy)

import pandas as pd # Tabular data (not used below; safe to remove if you like)

import matplotlib.pyplot as plt # Plotting training curves

# ---- Deep learning frameworks ----

import tensorflow as tf # TensorFlow: tensors, GPU execution, tf.data, Keras runtime

import keras # Keras (TF-backed): model/layer APIs (in TF 2.x / Keras 3 this imports TF-Keras)

# ---- Keras building blocks ----

from keras import layers # Common layers (Conv, Dense, Dropout, etc.)

from keras.applications import MobileNetV2 # Pretrained CNN (Imagenet weights)

from keras.layers import Dense, Dropout, GlobalAveragePooling2D # Head layers used here

from keras.models import Sequential # Simple container that stacks layers linearly

from keras.optimizers import Adam, RMSprop # Optimizers; you picked RMSprop

from keras.utils import image\_dataset\_from\_directory # Build tf.data Dataset from folders

# ---- Mixed precision (FP16 on GPU) ----

from keras import mixed\_precision

mixed\_precision.set\_global\_policy('mixed\_float16')

# ^ Enables "mixed precision": most compute in float16 on supported GPUs (e.g., T4/V100/A100) for speed.

# Keras automatically keeps certain numerically sensitive parts (like softmax/logits) in float32

# IF you request dtype='float32' for that last Dense (you did—good!).

# Automatic loss scaling is also handled for stability.

# ---- Paths to your Kaggle dataset ----

train\_dir = "/kaggle/input/signlanguagetransl/asl\_alphabet\_train/asl\_alphabet\_train"

test\_dir = "/kaggle/input/signlanguagetransl/asl\_alphabet\_test/asl\_alphabet\_test"

# NOTE: In this script we use train\_dir and create a validation split from it.

# test\_dir is defined but not used below (you can add a final test evaluation if desired).

# ---- Build training/validation datasets from folders ----

# image\_dataset\_from\_directory:

# - Expects structure: root/class\_name/\*.jpg

# - Returns a tf.data.Dataset yielding (images, labels) batches.

# - Labels are integer indices corresponding to class\_names.

train\_ds = image\_dataset\_from\_directory(

train\_dir,

image\_size=(224, 224), # Images resized to MobileNetV2's expected input size

batch\_size=32, # Number of images per batch

validation\_split=0.2, # 80/20 split for train/val (based on filenames & seed)

seed=42, # Keep split deterministic across runs

subset="training" # This creates the training portion

)

validation\_ds = image\_dataset\_from\_directory(

train\_dir,

image\_size=(224, 224),

batch\_size=32,

validation\_split=0.2,

seed=42,

subset="validation" # This creates the validation portion

)

# ---- Normalization / Preprocessing ----

# Option A (your current choice): scale pixel values to [0,1]

normalization\_layer = tf.keras.layers.Rescaling(1./255)

# NOTE: MobileNetV2's canonical preprocessing is to scale to [-1, 1].

# If you want to match pretrained weights more closely, you can instead do:

# from keras.applications.mobilenet\_v2 import preprocess\_input

# preprocessing = tf.keras.layers.Lambda(preprocess\_input)

# and replace `normalization\_layer` with `preprocessing` below.

# Apply normalization to datasets (keeps labels 'y' unchanged)

train = train\_ds.map(lambda x, y: (normalization\_layer(x), y))

validation = validation\_ds.map(lambda x, y: (normalization\_layer(x), y))

# Mixed precision note: images are float32 after Rescaling; Keras will feed them to the model

# and cast to float16 automatically under the global policy.

# ---- Performance: pipeline tuning ----

AUTOTUNE = tf.data.AUTOTUNE

# Cache + prefetch:

# - cache(): keep preprocessed batches in memory (fast but can be heavy for large datasets)

# - prefetch(): overlaps data preparation with model execution (keeps GPU busy)

train = train.cache().prefetch(buffer\_size=AUTOTUNE)

validation = validation.cache().prefetch(buffer\_size=AUTOTUNE)

# If you ever see memory issues on Kaggle, remove .cache() or use disk caching:

# train = train.cache('/kaggle/working/train.cache').prefetch(AUTOTUNE)

# ---- Load pretrained CNN backbone (transfer learning) ----

conv\_base = MobileNetV2(

weights="imagenet", # Start from ImageNet weights (general visual features)

include\_top=False, # Exclude the original classifier so we can add our own

input\_shape=(224, 224, 3)

)

conv\_base.summary()

# Fine-tuning policy:

conv\_base.trainable = True

for layer in conv\_base.layers[:-30]:

layer.trainable = False

# Explanation:

# - First freeze most layers to keep the generic features stable.

# - Unfreeze the last ~30 layers so the network can adapt to ASL images.

# - You can later unfreeze more layers once your new head is stable (common 2-phase strategy).

# ---- On-the-fly data augmentation (applied only during training) ----

data\_augmentation = keras.Sequential([

layers.RandomFlip("horizontal"),

layers.RandomRotation(0.1),

layers.RandomZoom(0.1),

layers.RandomTranslation(0.05, 0.05),

], name="data\_augmentation")

# These stochastic layers run in training mode only. At inference, they pass images unchanged.

# ---- Build the classifier head on top of the backbone ----

model = Sequential()

model.add(data\_augmentation) # 1) augment

model.add(conv\_base) # 2) feature extractor (conv\_base)

model.add(GlobalAveragePooling2D())# 3) spatially average feature maps -> feature vector

model.add(Dropout(0.3)) # 4) regularization to reduce overfitting

# Number of output units = number of classes. image\_dataset\_from\_directory gives class names:

num\_classes = len(train\_ds.class\_names)

# Mixed precision detail:

# Keep the final Dense in float32 to avoid numerical issues in softmax/logits under FP16.

model.add(Dense(num\_classes, activation="softmax", dtype="float32"))

# ---- Compile: choose optimizer, loss, metrics ----

model.compile(

optimizer=RMSprop(learning\_rate=1e-5), # Small LR for fine-tuning a pretrained net

loss='sparse\_categorical\_crossentropy', # Labels are integer-encoded (not one-hot)

metrics=['accuracy'] # Track accuracy during training

)

model.summary()

# ---- Train ----

# history records loss/accuracy per epoch for train & validation.

history = model.fit(

train,

epochs=5,

validation\_data=validation

)

# ---- Evaluate on validation set ----

loss, accuracy = model.evaluate(validation)

print("Validation Loss :", loss)

print("Validation Accuracy :", accuracy)

# ---- Plot learning curves ----

plt.plot(history.history["loss"], label="train")

plt.plot(history.history["val\_loss"], label="validation")

plt.legend()

plt.title("Training Loss vs Validation Loss")

plt.ylabel("Loss")

plt.xlabel("Epoch")

plt.show()

plt.plot(history.history["accuracy"], label="train")

plt.plot(history.history["val\_accuracy"], label="validation")

plt.legend()

plt.title("Training Accuracy vs Validation Accuracy")

plt.ylabel("Accuracy")

plt.xlabel("Epoch")

plt.show()

# ---- Save the trained Keras model ----

model.save("model.keras") # Keras v3 native format (recommended going forward)

# ---- Reload the model ----

# safe\_mode=False allows loading models that may contain TF ops or unregistered objects.

# It's useful if your graph includes things Keras doesn't serialize as standard layers.

loaded = tf.keras.models.load\_model("model.keras", safe\_mode=False)

# ---- (Optional) Rewrap to a "plain" single-input/single-output Keras Functional model ----

# Some environments/tools prefer a clean Keras graph with explicit inputs/outputs.

inp = loaded.inputs[0]

out = loaded.outputs[0]

fixed\_model = tf.keras.Model(inputs=inp, outputs=out)

# Save the "fixed" version

fixed\_model.save("model\_fixed.keras")

**What each library is doing (in plain English)**

* **NumPy (numpy)**: foundational array/matrix math. Often used implicitly by TensorFlow; you didn’t use it directly here.
* **Pandas (pandas)**: tables/dataframes. Not used in this script; you can remove it to keep things lean.
* **Matplotlib (matplotlib.pyplot)**: draws the training/validation curves.
* **TensorFlow (tensorflow)**: the engine that runs tensors on CPU/GPU/TPU, builds tf.data input pipelines, and executes your Keras model efficiently.
* **Keras (keras)**: high-level API for building, training, and saving models. In modern TF/Keras, keras uses TensorFlow under the hood.
* **keras.applications.MobileNetV2**: a lightweight CNN pretrained on ImageNet. We reuse its convolutional layers as a powerful feature extractor, then add our own classifier head for ASL letters.
* **Layers**:
  + GlobalAveragePooling2D: collapses each feature map into a single number (average), giving a compact, translation-robust feature vector.
  + Dropout: randomly disables a fraction of units during training to reduce overfitting.
  + Dense: fully connected layer for classification (one unit per class with softmax).
* **Optimizers (RMSprop, Adam)**: algorithms that update weights to minimize loss. You’re using **RMSprop** with a small LR since you’re fine-tuning a pretrained net. **Adam** is also popular; both are good choices.
* **image\_dataset\_from\_directory**: walks your folder tree, builds batches of images and integer labels, and returns a **tf.data.Dataset** that streams efficiently to the GPU.
* **Mixed Precision (mixed\_precision.set\_global\_policy('mixed\_float16'))**: runs most layers in **float16** for speed and lower memory use on supported GPUs; keeps numerically sensitive parts in float32 (we explicitly set the final Dense to dtype="float32").

**How the data flows, step by step**

1. **Read from disk**  
   image\_dataset\_from\_directory scans train\_dir and builds batches of (images, labels) where:
   * images: shape (batch, 224, 224, 3) in float32 after resizing.
   * labels: shape (batch,) with integer class IDs (0…num\_classes-1).  
     It also generates a **consistent** 80/20 train/validation split using validation\_split + seed.
2. **Normalize / preprocess**  
   We map a Rescaling(1./255) layer across the dataset so pixels go from [0,255] to [0,1].  
   *(Optional but recommended for MobileNetV2: use preprocess\_input to scale to [-1,1].)*
3. **Input pipeline optimizations**  
   .cache() keeps preprocessed batches in memory (fast but memory-heavy).  
   .prefetch(AUTOTUNE) overlaps CPU data prep with GPU training.
4. **Augment (training-time only)**  
   RandomFlip/Rotation/Zoom/Translation perturb the images each epoch to teach robustness.  
   During model.fit(), these layers perform stochastic transforms; during model.evaluate() and model.predict(), they pass through unchanged.
5. **Feature extraction (transfer learning)**  
   MobileNetV2(include\_top=False) processes images into rich feature maps learned from ImageNet.  
   We **freeze most layers** and **unfreeze the last 30** so the model adapts but doesn’t forget general vision features.
6. **Classification head**  
   GlobalAveragePooling2D turns feature maps into a 1-D vector, Dropout regularizes, and a Dense(num\_classes, softmax) predicts class probabilities.  
   We force this final Dense to float32 for numerical stability under mixed precision.
7. **Training loop**  
   model.fit(train, validation\_data=validation, epochs=5) iterates over batches, backpropagating gradients computed from sparse\_categorical\_crossentropy and updating weights with **RMSprop**.
8. **Evaluation + plots**  
   model.evaluate(validation) reports final metrics on the held-out validation set.  
   matplotlib plots how loss/accuracy evolved over epochs for both train and validation.
9. **Save / reload**  
   model.save("model.keras") writes the Keras v3 format.  
   load\_model(..., safe\_mode=False) reloads it even if there are TF ops or pieces Keras doesn’t recognize as standard layers.  
   Rewrapping as fixed\_model = tf.keras.Model(inputs=..., outputs=...) yields a clean, single-input / single-output Functional model, then saved again as model\_fixed.keras.

**Practical notes & small improvements (optional)**

* **MobileNetV2 preprocessing**: For best transfer performance, replace Rescaling(1./255) with preprocess\_input from keras.applications.mobilenet\_v2 (scales to [-1,1] as expected by the pretrained weights).
* **Caching memory**: ASL Alphabet train set is big; if you hit RAM limits on Kaggle, remove .cache() or cache to disk (.cache('/kaggle/working/train.cache')).
* **Longer training**: 5 epochs is just a quick warm-up. Try 15–25 epochs with **EarlyStopping** and **ReduceLROnPlateau** callbacks to get higher accuracy.
* **Test set**: You defined test\_dir; you can build a test\_ds similarly (no validation\_split) and run model.evaluate(test\_ds) for an unbiased final metric.
* **Learning rate**: With more frozen layers, you can start a bit higher (e.g., 3e-5) and then reduce when unfreezing more layers.
* **Class names**: Accessible via train\_ds.class\_names. Useful for mapping predictions back to letters.

If you want, say the word and I’ll switch the normalization to the MobileNetV2 preprocess\_input, add EarlyStopping, and include a proper test-set evaluation—all fully commented like above.