

```
In [1]: # =====
# Step 1: Mount Google Drive
# This step mounts Google Drive to access the saved models,
# class names, metrics files, and application resources.
# =====

from google.colab import drive

drive.mount("/content/drive")

Mounted at /content/drive
```

```
In [2]: # =====
# Step 2: Install Required Libraries
# This step installs Gradio, TIMM, and TextBlob for the
# multi-model ASL web application and spelling correction.
# =====

!pip -q install --upgrade gradio
!pip -q install timm
!pip -q install textblob

print("=" * 60)
print("Required Libraries Installed Successfully")
print("=" * 60)

[?]25l  [?]90m-----[?]0m [?]32m0.0/31.0 MB[?]0m [?]31m?[?]0m eta [?]36m-:--:--[?]0m
[?]2K  [?]91m-----[?]0m[?]90m-[?]0m[?]90m-----[?]0m [?]32m9.4/31.0 MB[?]0m [?]31m282.1 MB/s[?]0m eta [?]36m0:00:01[?]0m
[?]2K  [?]91m-----[?]0m[?]90m-[?]0m[?]90m-----[?]0m [?]32m19.6/31.0 MB[?]0m [?]31m290.1 MB/s[?]0m eta [?]36m0:00:01[?]0m
[?]2K  [?]91m-----[?]0m[?]90m-[?]0m[?]90m-[?]0m [?]32m29.8/31.0 MB[?]0m [?]31m291.0 MB/s[?]0m eta [?]36m0:00:01[?]0m
[?]2K  [?]91m-----[?]0m[?]91m-[?]0m [?]32m31.0/31.0 MB[?]0m [?]31m294.0 MB/s[?]0m eta [?]36m0:00:01[?]0m
[?]2K  [?]91m-----[?]0m[?]91m-[?]0m [?]32m31.0/31.0 MB[?]0m [?]31m294.0 MB/s[?]0m eta [?]36m0:00:01[?]0m
[?]2K  [?]90m-----[?]0m [?]32m31.0/31.0 MB[?]0m [?]31m94.9 MB/s[?]0m eta [?]36m0:00:00[?]0m
[?]25h=====
Required Libraries Installed Successfully
=====
```

```
In [3]: # =====
# Step 3: Import Required Libraries
# This step imports all libraries required for loading models,
# preprocessing images, prediction, spelling correction,
# metrics display, and Gradio interface development.
# =====

import os
import json
import gc
import warnings

import numpy as np
import pandas as pd

from PIL import Image

import tensorflow as tf

import torch
import torch.nn as nn

from torchvision import transforms

import timm
import gradio as gr

from textblob import TextBlob

from tensorflow.keras.applications.resnet50 import (
    preprocess_input as resnet50_preprocess
)

from tensorflow.keras.applications.mobilenet_v2 import (
    preprocess_input as mobilenetv2_preprocess
)

from tensorflow.keras.applications.efficientnet import (
    preprocess_input as efficientnetb0_preprocess
)

warnings.filterwarnings("ignore")

print("=" * 60)
print("Libraries Imported Successfully")
print("=" * 60)
print(f"Gradio Version      : {gr.__version__}")
print(f"TensorFlow Version: {tf.__version__}")
print(f"PyTorch Version      : {torch.__version__}")
print(f"TIMM Version         : {timm.__version__}")
print("=" * 60)

=====
Libraries Imported Successfully
=====
Gradio Version      : 6.20.0
TensorFlow Version: 2.20.0
PyTorch Version      : 2.11.0+cu128
TIMM Version         : 1.0.27
=====
```

```
In [4]: # =====
# Step 4: Configure Device and Application Settings
# This step configures the GPU or CPU device, image size,
# class labels, and default application settings.
# =====

IMAGE_SIZE = 224

CLASS_NAMES = [
    "0", "1", "2", "3", "4", "5", "6", "7", "8", "9",
    "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"
]

NUM_CLASSES = len(CLASS_NAMES)

TORCH_DEVICE = torch.device(
    "cuda"
    if torch.cuda.is_available()
    else "cpu"
)

tensorflow_gpu = tf.config.list_physical_devices("GPU")

print("=" * 60)
print("Application Configuration")
print("=" * 60)
print(f"Image Size          : {IMAGE_SIZE} x {IMAGE_SIZE}")
print(f"Number of Classes    : {NUM_CLASSES}")
print(f"PyTorch Device       : {TORCH_DEVICE}")
print(f"TensorFlow GPU Found : {'Yes' if tensorflow_gpu else 'No'}")

if torch.cuda.is_available():
    print(
        "GPU Name          :",
        torch.cuda.get_device_name(0)
    )
else:
    print("GPU Name          : CPU")

print("=" * 60)

=====
Application Configuration
=====
Image Size          : 224 x 224
Number of Classes    : 36
PyTorch Device       : cuda
TensorFlow GPU Found : Yes
GPU Name             : NVIDIA A100-SXM4-40GB
=====
```

```
In [5]: # =====
# Step 5: Define Model, Class, and Metrics Paths
# This step defines the saved files used by all six models
# and verifies that the required application files exist.
# =====

PROJECT_DIR = (
    "/content/drive/MyDrive/ASL_Final_Project"
)

MODELS_DIR = os.path.join(
    PROJECT_DIR,
    "02_Models"
)

METRICS_DIR = os.path.join(
    PROJECT_DIR,
    "03_Results",
    "Metrics"
)

MODEL_CONFIGS = {
    "ResNet50": {
        "framework": "tensorflow",
        "model_path": os.path.join(
            MODELS_DIR,
            "ResNet50",
            "final_resnet50_asl_model.keras"
        ),
        "class_names_path": os.path.join(
            MODELS_DIR,
            "ResNet50",
            "asl_class_names.txt"
        ),
        "metrics_path": os.path.join(
            METRICS_DIR,
            "resnet50_metrics.json"
        )
    },

    "MobileNetV2": {
        "framework": "tensorflow",
        "model_path": os.path.join(
            MODELS_DIR,
            "MobileNetV2",
            "final_mobilenetv2_asl_model.keras"
        ),
        "class_names_path": os.path.join(
            MODELS_DIR,
            "MobileNetV2",
            "mobilenetv2_class_names.txt"
        ),
        "metrics_path": os.path.join(
            METRICS_DIR,
            "mobilenetv2_metrics.json"
        )
    },

    "EfficientNetB0": {
        "framework": "tensorflow",
        "model_path": os.path.join(
            MODELS_DIR,
            "EfficientNetB0",
            "final_efficientnetb0_asl_model.keras"
        ),
        "class_names_path": os.path.join(
            MODELS_DIR,
            "EfficientNetB0",
            "efficientnetb0_class_names.txt"
        ),
        "metrics_path": os.path.join(
            METRICS_DIR,
            "efficientnetb0_metrics.json"
        )
    },

    "Vision Transformer": {
        "framework": "pytorch",
        "architecture": "vit_base_patch16_224",
        "model_path": os.path.join(
            MODELS_DIR,
            "VisionTransformer",
            "final_vit_asl_model.pth"
        ),
        "class_names_path": os.path.join(
            MODELS_DIR,
            "VisionTransformer",
            "vit_class_names.txt"
        ),
        "metrics_path": os.path.join(
            METRICS_DIR,
            "vit_metrics.json"
        )
    },

    "Swin Transformer": {
        "framework": "pytorch",
        "architecture": "swin_tiny_patch4_window7_224",
        "model_path": os.path.join(
            MODELS_DIR,
            "SwinTransformer",
            "final_swin_asl_model.pth"
        ),
        "class_names_path": os.path.join(
            MODELS_DIR,
            "SwinTransformer",
            "swin_class_names.txt"
        ),
        "metrics_path": os.path.join(
            METRICS_DIR,
            "swin_metrics.json"
        )
    },

    "Hybrid MobileNetV2-Swin": {
        "framework": "hybrid",
        "model_path": os.path.join(
            MODELS_DIR,
            "Hybrid",
            "final_hybrid_mobilenetv2_swin_asl_model.pth"
        ),
        "class_names_path": os.path.join(
            MODELS_DIR,
            "Hybrid",
            "hybrid_mobilenetv2_swin_class_names.txt"
        ),
        "metrics_path": os.path.join(
            METRICS_DIR,
            "hybrid_metrics.json"
        )
    }
}

missing_required_files = []

print("=" * 80)
print("Application Files Verification")
print("=" * 80)

for model_name, config in MODEL_CONFIGS.items():

    print(f"\nModel: {model_name}")

    model_exists = os.path.isfile(
        config["model_path"]
    )

    classes_exist = os.path.isfile(
        config["class_names_path"]
    )

    metrics_exist = os.path.isfile(
        config["metrics_path"]
    )

    print(
        f"Model File   : "
        f"{'Found' if model_exists else 'Missing'}"
    )

    print(
        f"Class Names   : "
        f"{'Found' if classes_exist else 'Missing'}"
    )

    print(
        f"Metrics File  : "
        f"{'Found' if metrics_exist else 'Missing'}"
    )
```

```
if not model_exists:
    missing_required_files.append(
        config["model_path"]
    )

if not classes_exist:
    missing_required_files.append(
        config["class_names_path"]
    )

if not metrics_exist:
    missing_required_files.append(
        config["metrics_path"]
    )

print("\n" + "=" * 80)

if missing_required_files:

    print("Application Status: INCOMPLETE")
    print("Missing Files:")

    for missing_file in missing_required_files:
        print(f"- {missing_file}")

    raise FileNotFoundError(
        "Some required application files are missing. "
        "Check the paths displayed above."
    )

else:

    print("Application Status: READY")
    print("All model, class, and metrics files were found.")

print("=" * 80)
```

```
=====
Application Files Verification
=====

Model: ResNet50
Model File   : Found
Class Names  : Found
Metrics File : Found

Model: MobileNetV2
Model File   : Found
Class Names  : Found
Metrics File : Found

Model: EfficientNetB0
Model File   : Found
Class Names  : Found
Metrics File : Found

Model: Vision Transformer
Model File   : Found
Class Names  : Found
Metrics File : Found

Model: Swin Transformer
Model File   : Found
Class Names  : Found
Metrics File : Found

Model: Hybrid MobileNetV2-Swin
Model File   : Found
Class Names  : Found
Metrics File : Found

=====
Application Status: READY
All model, class, and metrics files were found.
=====
```

In [6]:

```
# =====
# Step 6: Load Class Names and Model Metrics
# This step loads the class names and evaluation metrics
# associated with each trained ASL recognition model.
# =====

MODEL_CLASS_NAMES = {}
MODEL_METRICS = {}

def load_class_names(
    class_names_path
):

    with open(
        class_names_path,
        "r"
    ) as class_file:

        loaded_class_names = [
            line.strip()
            for line in class_file.readlines()
            if line.strip()
        ]

        if len(loaded_class_names) != NUM_CLASSES:

            raise ValueError(
                f"Expected {NUM_CLASSES} class names, "
                f"but found {len(loaded_class_names)} in:\n"
                f"{class_names_path}"
            )

        return loaded_class_names

for model_name, config in MODEL_CONFIGS.items():

    MODEL_CLASS_NAMES[model_name] = (
        load_class_names(
            config["class_names_path"]
        )
    )

    with open(
        config["metrics_path"],
        "r"
    ) as metrics_file:

        MODEL_METRICS[model_name] = json.load(
            metrics_file
        )

print("=" * 70)
print("Class Names and Model Metrics Loaded Successfully")
print("=" * 70)

for model_name in MODEL_CONFIGS:

    accuracy = (
        MODEL_METRICS[model_name]["test_accuracy"]
        * 100
    )

    print(
        f"{model_name:<28}: "
        f"{len(MODEL_CLASS_NAMES[model_name])} classes | "
        f"Accuracy: {accuracy:.2f}%"
    )

print("=" * 70)
```

```
=====
Class Names and Model Metrics Loaded Successfully
=====
ResNet50           : 36 classes | Accuracy: 92.46%
MobileNetV2        : 36 classes | Accuracy: 91.87%
EfficientNetB0      : 36 classes | Accuracy: 89.88%
Vision Transformer  : 36 classes | Accuracy: 98.81%
Swin Transformer    : 36 classes | Accuracy: 99.60%
Hybrid MobileNetV2-Swin : 36 classes | Accuracy: 99.80%
=====
```

```
In [7]: # =====
# Step 7: Define the Hybrid Model Architecture
# This step recreates the Hybrid MobileNetV2-Swin architecture
# required to load and use the saved model weights.
# =====

class HybridMobileNetSwin(nn.Module):

    def __init__(
        self,
        num_classes
    ):

        super().__init__()

        self.mobilenet = timm.create_model(
            "mobilenetv2_100",
            pretrained=False,
            num_classes=0,
            global_pool="avg"
        )

        self.swin = timm.create_model(
            "swin_tiny_patch4_window7_224",
            pretrained=False,
            num_classes=0,
            global_pool="avg"
        )

        mobilenet_features = (
            self.mobilenet.num_features
        )

        swin_features = (
            self.swin.num_features
        )

        combined_features = (
            mobilenet_features
            + swin_features
        )

        self.classifier = nn.Sequential(
            nn.Linear(
                combined_features,
                512
            ),

            nn.ReLU(),

            nn.Dropout(0.5),

            nn.Linear(
                512,
                num_classes
            )
        )

    def forward(
        self,
        images
    ):

        mobilenet_features = self.mobilenet(
            images
        )

        swin_features = self.swin(
            images
        )

        fused_features = torch.cat(
            (
                mobilenet_features,
                swin_features
            ),
            dim=1
        )

        outputs = self.classifier(
            fused_features
        )

        return outputs

print("=" * 60)
print("Hybrid Model Architecture Defined Successfully")
print("=" * 60)
```

```
=====
Hybrid Model Architecture Defined Successfully
=====
```

```
In [8]: # =====
# Step 8: Create Image Preprocessing Functions
# This step creates the TensorFlow and PyTorch preprocessing
# pipelines required by the six ASL recognition models.
# =====

PYTORCH_IMAGE_TRANSFORM = transforms.Compose([
    transforms.Resize(
        (IMAGE_SIZE, IMAGE_SIZE)
    ),

    transforms.ToTensor(),

    transforms.Normalize(
        mean=[
            0.485,
            0.456,
            0.406
        ],

        std=[
            0.229,
            0.224,
            0.225
        ]
    )
])

def prepare_tensorflow_image(
    image,
    model_name
):

    if not isinstance(
        image,
        Image.Image
    ):

        image = Image.fromarray(
            np.asarray(image)
        )

    image = image.convert(
        "RGB"
    )

    image = image.resize(
        (IMAGE_SIZE, IMAGE_SIZE)
    )

    image_array = np.asarray(
        image,
        dtype=np.float32
    )

    image_array = np.expand_dims(
        image_array,
        axis=0
    )

    if model_name == "ResNet50":

        image_array = resnet50_preprocess(
            image_array
        )

    elif model_name == "MobileNetV2":

        image_array = mobilenetv2_preprocess(
            image_array
        )

    elif model_name == "EfficientNetB0":

        image_array = efficientnetb0_preprocess(
            image_array
        )

    else:

        raise ValueError(
            f"Unsupported TensorFlow model: "
            f"{model_name}"
        )

    return image_array

def prepare_pytorch_image(
    image
):

    if not isinstance(
        image,
        Image.Image
    ):

        image = Image.fromarray(
            np.asarray(image)
        )

    image = image.convert(
        "RGB"
    )

    image_tensor = PYTORCH_IMAGE_TRANSFORM(
        image
    )

    image_tensor = image_tensor.unsqueeze(
        0
    )

    image_tensor = image_tensor.to(
        TORCH_DEVICE
    )

    return image_tensor

print("=" * 60)
print("Image Preprocessing Functions Created Successfully")
print("=" * 60)
```

```
=====
Image Preprocessing Functions Created Successfully
=====
```

```
In [9]: # =====
# Step 9: Create the Smart Model Loader
# This step loads only the selected model and releases the
# previous model to reduce RAM and GPU memory consumption.
# =====

ACTIVE_MODEL = None
ACTIVE_MODEL_NAME = None
ACTIVE_FRAMEWORK = None

def clear_active_model():

    global ACTIVE_MODEL
    global ACTIVE_MODEL_NAME
    global ACTIVE_FRAMEWORK

    ACTIVE_MODEL = None
    ACTIVE_MODEL_NAME = None
    ACTIVE_FRAMEWORK = None

    tf.keras.backend.clear_session()

    gc.collect()

    if torch.cuda.is_available():
        torch.cuda.empty_cache()

def load_torch_checkpoint(
    checkpoint_path
):

    try:

        checkpoint = torch.load(
            checkpoint_path,
            map_location="cpu",
            weights_only=True
        )

    except TypeError:

        checkpoint = torch.load(
            checkpoint_path,
            map_location="cpu"
        )

    if (
        isinstance(checkpoint, dict)
        and "state_dict" in checkpoint
    ):

        checkpoint = checkpoint[
            "state_dict"
        ]

    cleaned_checkpoint = {}

    for key, value in checkpoint.items():

        cleaned_key = key

        if cleaned_key.startswith(
            "module."
        ):

            cleaned_key = cleaned_key[
                len("module.") :
            ]

        cleaned_checkpoint[
            cleaned_key
        ] = value

    return cleaned_checkpoint

def create_pytorch_model(
    model_name
):

    if model_name == "Vision Transformer":

        pytorch_model = timm.create_model(
            "vit_base_patch16_224",
            pretrained=False,
            num_classes=NUM_CLASSES
        )

    elif model_name == "Swin Transformer":

        pytorch_model = timm.create_model(
            "swin_tiny_patch4_window7_224",
            pretrained=False,
            num_classes=NUM_CLASSES
        )

    elif model_name == (
        "Hybrid MobileNetV2-Swin"
    ):

        pytorch_model = HybridMobileNetSwin(
            num_classes=NUM_CLASSES
        )

    else:

        raise ValueError(
            f"Unsupported PyTorch model: "
            f"{model_name}"
        )

    return pytorch_model

def load_selected_model(
    model_name
):

    global ACTIVE_MODEL
    global ACTIVE_MODEL_NAME
    global ACTIVE_FRAMEWORK

    if (
        ACTIVE_MODEL is not None
        and ACTIVE_MODEL_NAME == model_name
    ):

        return (
            ACTIVE_MODEL,
            ACTIVE_FRAMEWORK
        )

    if model_name not in MODEL_CONFIGS:

        raise ValueError(
            f"Unknown model: {model_name}"
        )

    clear_active_model()

    config = MODEL_CONFIGS[
        model_name
    ]

    framework = config[
        "framework"
    ]

    model_path = config[
        "model_path"
    ]

    print(
        f>Loading {model_name}..."
    )

    if framework == "tensorflow":

        loaded_model = tf.keras.models.load_model(
            model_path,
            compile=False
        )

    elif framework in [
        "pytorch",
        "hybrid"
    ]:
```

```
loaded_model = create_pytorch_model(
    model_name
)

state_dictionary = load_torch_checkpoint(
    model_path
)

loaded_model.load_state_dict(
    state_dictionary,
    strict=True
)

loaded_model = loaded_model.to(
    TORCH_DEVICE
)

loaded_model.eval()

else:

    raise ValueError(
        f"Unsupported framework: "
        f"{framework}"
    )

ACTIVE_MODEL = loaded_model
ACTIVE_MODEL_NAME = model_name
ACTIVE_FRAMEWORK = framework

print(
    f"{model_name} Loaded Successfully"
)

return (
    ACTIVE_MODEL,
    ACTIVE_FRAMEWORK
)

print("=" * 60)
print("Smart Model Loader Created Successfully")
print("=" * 60)
```

```
=====
Smart Model Loader Created Successfully
=====
```

```
In [10]: # =====
# Step 10: Create Prediction Functions
# This step predicts the ASL class of an uploaded image and
# returns the predicted letter, confidence, and top results.
# =====

def predict_probabilities(
    image,
    model_name
):

    if image is None:

        raise ValueError(
            "Please upload an image or use the camera."
        )

    if model_name not in MODEL_CONFIGS:

        raise ValueError(
            "Please select a valid model."
        )

    model, framework = load_selected_model(
        model_name
    )

    model_class_names = MODEL_CLASS_NAMES[
        model_name
    ]

    if framework == "tensorflow":

        prepared_image = (
            prepare_tensorflow_image(
                image,
                model_name
            )
        )

        probabilities = model.predict(
            prepared_image,
            verbose=0
        )[0]

    else:

        prepared_image = (
            prepare_pytorch_image(
                image
            )
        )

        model.eval()

        with torch.inference_mode():

            logits = model(
                prepared_image
            )

            probabilities = torch.softmax(
                logits,
                dim=1
            )[0]

            probabilities = (
                probabilities
                .detach()
                .cpu()
                .numpy()
            )

        probabilities = np.asarray(
            probabilities,
            dtype=np.float64
        )

        if len(probabilities) != len(
            model_class_names
        ):

            raise ValueError(
                "The number of model outputs does not match "
                "the number of class names."
            )

        return (
            probabilities,
            model_class_names
        )

def predict_single_image(
    image,
    model_name
):

    try:

        probabilities, model_class_names = (
            predict_probabilities(
                image,
                model_name
            )
        )

        predicted_index = int(
            np.argmax(probabilities)
        )

        predicted_letter = (
            model_class_names[
                predicted_index
            ].upper()
        )

        confidence = float(
            probabilities[
                predicted_index
            ]
        )

        top_indices = np.argsort(
            probabilities
        )[-5:][::-1]

        top_predictions = {
            model_class_names[
                index
            ].upper():
            float(
                probabilities[index]
            )
            for index in top_indices
        }

        status_message = (
            f"Prediction completed successfully "
            f"using {model_name}."
        )

        return (
            top_predictions,
            predicted_letter,
            f"{confidence * 100:.2f}%",
            status_message
        )

    except Exception as error:

        return (
            {},
            "",
            "",
            f"Prediction Error: {error}"
        )

print("=" * 60)
print("Prediction Functions Created Successfully")
print("=" * 60)
```

```
=====
Prediction Functions Created Successfully
=====
```

```
In [11]: # =====
# Step 11: Create Word Formation and Spelling Correction
# This step creates functions to build a word from predicted
# letters, remove letters, clear the word, and correct spelling.
# =====

def correct_spelling(
    raw_word
):
    raw_word = (
        raw_word or ""
    ).strip()

    if not raw_word:
        return ""

    # Do not modify sequences containing numbers.
    if any(
        character.isdigit()
        for character in raw_word
    ):
        return raw_word.upper()

    try:
        corrected_word = str(
            TextBlob(
                raw_word.lower()
            ).correct()
        )

        return corrected_word.upper()

    except Exception:
        return raw_word.upper()

def append_predicted_letter(
    predicted_letter,
    current_word
):
    predicted_letter = (
        predicted_letter or ""
    ).strip().upper()

    current_word = (
        current_word or ""
    ).strip().upper()

    if not predicted_letter:

        corrected_word = correct_spelling(
            current_word
        )

        return (
            current_word,
            current_word,
            corrected_word,
            "No predicted letter is available."
        )

    updated_word = (
        current_word
        + predicted_letter
    )

    corrected_word = correct_spelling(
        updated_word
    )

    return (
        updated_word,
        updated_word,
        corrected_word,
        f"Letter {predicted_letter} added successfully."
    )

def remove_last_letter(
    current_word
):
    current_word = (
        current_word or ""
    ).strip().upper()

    updated_word = current_word[:-1]

    corrected_word = correct_spelling(
        updated_word
    )

    return (
        updated_word,
        updated_word,
        corrected_word,
        "Last letter removed."
    )

def clear_current_word():
    return (
        "",
        "",
        "",
        "Word cleared successfully."
    )

def correct_current_word(
    current_word
):
    current_word = (
        current_word or ""
    ).strip().upper()

    corrected_word = correct_spelling(
        current_word
    )

    return (
        corrected_word,
        "Spelling correction completed."
    )

print("=" * 60)
print("Word Formation and Spelling Functions Created Successfully")
print("=" * 60)
```

```
=====
Word Formation and Spelling Functions Created Successfully
=====
```

```
In [12]: # =====
# Step 12: Create Multiple-Image Word Prediction
# This step predicts multiple uploaded images, combines their
# predicted classes, and applies spelling correction.
# =====

def get_uploaded_file_path(
    uploaded_file
):
    if isinstance(
        uploaded_file,
        str
    ):
        return uploaded_file

    if hasattr(
        uploaded_file,
        "name"
    ):
        return uploaded_file.name

    if hasattr(
        uploaded_file,
        "path"
    ):
        return uploaded_file.path

    raise ValueError(
        "Unsupported uploaded file format."
    )

def predict_multiple_images(
    uploaded_files,
    model_name,
    progress=gr.Progress()
):
    if not uploaded_files:
        return (
            "",
            "",
            [],
            "Please upload one or more ASL images."
        )

    predicted_letters = []
    gallery_results = []

    total_files = len(
        uploaded_files
    )

    for image_index, uploaded_file in enumerate(
        uploaded_files,
        start=1
    ):
        progress(
            image_index / total_files,
            desc=(
                f"Processing image "
                f"{image_index} of {total_files}"
            )
        )

        image_path = get_uploaded_file_path(
            uploaded_file
        )

        image = Image.open(
            image_path
        ).convert(
            "RGB"
        )

        (
            probabilities,
            model_class_names
        ) = predict_probabilities(
            image,
            model_name
        )

        predicted_index = int(
            np.argmax(
                probabilities
            )
        )

        predicted_letter = (
            model_class_names[
                predicted_index
            ].upper()
        )

        confidence = float(
            probabilities[
                predicted_index
            ]
        )

        predicted_letters.append(
            predicted_letter
        )

        caption = (
            f"Image {image_index}: "
            f"{predicted_letter} "
            f"({confidence * 100:.2f}%)"
        )

        gallery_results.append(
            (
                image,
                caption
            )
        )

    raw_word = "".join(
        predicted_letters
    )

    corrected_word = correct_spelling(
        raw_word
    )

    status_message = (
        f"{total_files} images processed successfully "
        f"using {model_name}."
    )

    return (
        raw_word,
        corrected_word,
        gallery_results,
        status_message
    )

print("=" * 60)
print("Multiple-Image Prediction Created Successfully")
print("=" * 60)
```

```
=====
Multiple-Image Prediction Created Successfully
=====
```

In [13]:

```
# =====
# Step 13: Create Model Information Functions
# This step creates automatic model information and comparison
# tables using the saved metrics JSON files.
# =====

def create_model_information(
    model_name
):
    if model_name not in MODEL_METRICS:
        return (
            """ Model information is unavailable.
            """
        )

    metrics = MODEL_METRICS[
        model_name
    ]

    accuracy = (
        metrics["test_accuracy"]
        * 100
    )

    precision = (
        metrics["weighted_precision"]
        * 100
    )

    recall = (
        metrics["weighted_recall"]
        * 100
    )

    f1_score_value = (
        metrics["weighted_f1_score"]
        * 100
    )

    time_per_image_ms = (
        metrics[
            "average_time_per_image_seconds"
        ]
        * 1000
    )

    parameters = int(
        metrics["parameters"]
    )

    model_size = float(
        metrics["model_size_mb"]
    )

    test_images = int(
        metrics["test_images"]
    )

    gpu_name = metrics.get(
        "gpu",
        "Not recorded"
    )

    framework = metrics.get(
        "framework",
        "Not recorded"
    )

    model_information = f"""
    ### Selected Model Information

    | Metric | Value |
    |---|---|
    | **Model** | {model_name} |
    | **Framework** | {framework} |
    | **Test Images** | {test_images:,} |
    | **Test Accuracy** | {accuracy:.2f}% |
    | **Weighted Precision** | {precision:.2f}% |
    | **Weighted Recall** | {recall:.2f}% |
    | **Weighted F1-Score** | {f1_score_value:.2f}% |
    | **Test Loss** | {metrics["test_loss"]:.4f} |
    | **Time per Image** | {time_per_image_ms:.3f} ms |
    | **Parameters** | {parameters:,} |
    | **Model Size** | {model_size:.2f} MB |
    | **Evaluation Device** | {gpu_name} |

    **Test Accuracy** is the model's performance on the complete test dataset.
    **Prediction Confidence** is the confidence for the currently uploaded image.
    """

    return model_information

def create_models_comparison_table():

    comparison_rows = []

    for model_name, metrics in MODEL_METRICS.items():

        comparison_rows.append({
            "Model": model_name,

            "Accuracy (%)": round(
                metrics["test_accuracy"]
                * 100,
                2
            ),

            "Precision (%)": round(
                metrics["weighted_precision"]
                * 100,
                2
            ),

            "Recall (%)": round(
                metrics["weighted_recall"]
                * 100,
                2
            ),

            "F1-Score (%)": round(
                metrics["weighted_f1_score"]
                * 100,
                2
            ),

            "Test Loss": round(
                metrics["test_loss"],
                4
            ),

            "Time/Image (ms)": round(
                metrics[
                    "average_time_per_image_seconds"
                ]
                * 1000,
                3
            ),

            "Parameters": int(
                metrics["parameters"]
            ),

            "Model Size (MB)": round(
                metrics["model_size_mb"],
                2
            )
        })

    comparison_table = pd.DataFrame(
        comparison_rows
    )

    comparison_table = (
        comparison_table
        .sort_values(
            by="Accuracy (%)",
            ascending=False
        )
        .reset_index(
            drop=True
        )
    )

    comparison_table.insert(
        0,
        "Rank",
        np.arange(
            1,
```

```
        len(comparison_table) + 1
    )
)

return comparison_table

def handle_model_change(
    model_name
):
    return (
        create_model_information(
            model_name
        ),
        (
            f"{model_name} selected. "
            f"The model will load when prediction starts."
        )
    )

print("=" * 60)
print("Model Information Functions Created Successfully")
print("=" * 60)

display(
    create_models_comparison_table()
)
```

=====
Model Information Functions Created Successfully
=====

	Rank	Model	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)	Test Loss	Time/Image (ms)	Parameters	Model Size (MB)
0	1	Hybrid MobileNetV2-Swin	99.80	99.81	99.80	99.80	0.0112	4.986	30810782	117.86
1	2	Swin Transformer	99.60	99.60	99.60	99.60	0.0189	3.191	27547038	105.15
2	3	Vision Transformer	98.81	98.83	98.81	98.81	0.0427	7.642	85826340	327.46
3	4	ResNet50	92.46	93.36	92.46	92.42	0.3774	4.542	24121508	96.72
4	5	MobileNetV2	91.87	92.86	91.87	91.93	0.3565	3.600	2595172	13.03
5	6	EfficientNetB0	89.88	91.42	89.88	89.51	0.5200	3.504	4386759	20.11

```
In [14]: # =====
# Step 14: Define Application Styling
# This step defines the visual appearance of the multi-model
# ASL recognition web application.
# =====

CUSTOM_CSS = """
.gradio-container {
    max-width: 1280px !important;
    margin: auto !important;
}

#main-title {
    text-align: center;
    margin-bottom: 5px;
}

#main-subtitle {
    text-align: center;
    opacity: 0.85;
    margin-bottom: 22px;
}

.predicted-letter textarea {
    font-size: 34px !important;
    font-weight: bold !important;
    text-align: center !important;
}

.confidence-box textarea {
    font-size: 24px !important;
    font-weight: bold !important;
    text-align: center !important;
}

.word-output textarea {
    font-size: 28px !important;
    font-weight: bold !important;
    text-align: center !important;
}

.status-output textarea {
    font-size: 15px !important;
}

.footer-text {
    text-align: center;
    opacity: 0.75;
    margin-top: 25px;
}
"""

APPLICATION_TITLE = (
    "ASL Multi-Model Recognition System"
)

APPLICATION_DESCRIPTION = """
Select one of the trained models, upload an ASL image or use
the camera, and view the predicted letter with its confidence.
You may also combine several predictions into a word and apply
automatic English spelling correction.
"""

print("=" * 60)
print("Application Styling Created Successfully")
print("=" * 60)
```

=====
Application Styling Created Successfully
=====

In [15]:

```
# =====
# Step 15: Build the Gradio Web Application
# This step creates the complete multi-model ASL recognition
# website with image, camera, word, batch, and comparison modes.
# =====

with gr.Blocks(
    title=APPLICATION_TITLE,
    css=CUSTOM_CSS
) as demo:

    current_word_state = gr.State(
        value=""
    )

    gr.Markdown(
        f"""
# {APPLICATION_TITLE}
### American Sign Language Classification, Word Formation, and Spelling Correction
"""
    ,
    elem_id="main-title"
    )

    gr.Markdown(
        APPLICATION_DESCRIPTION,
        elem_id="main-subtitle"
    )

    with gr.Row():

        model_selector = gr.Dropdown(
            choices=list(
                MODEL_CONFIGS.keys()
            ),
            value="Swin Transformer",
            label="Select Recognition Model",
            interactive=True,
            scale=2
        )

        application_status = gr.Textbox(
            value="Application is ready.",
            label="Application Status",
            interactive=False,
            scale=2,
            elem_classes=[
                "status-output"
            ]
        )

        model_information = gr.Markdown(
            value=create_model_information(
                "Swin Transformer"
            )
        )

    with gr.Tabs():

        # =====
        # Tab 1: Single Image or Camera
        # =====

        with gr.Tab(
            "Single Image / Camera"
        ):

            with gr.Row():

                image_input = gr.Image(
                    sources=[
                        "upload",
                        "webcam"
                    ],
                    type="pil",
                    label="Upload an ASL Image or Use Camera",
                    height=430
                )

                with gr.Column():

                    top_predictions_output = gr.Label(
                        label="Top Five Predictions",
                        num_top_classes=5
                    )

                    predicted_letter_output = gr.Textbox(
                        label="Predicted Letter",
                        interactive=False,
                        elem_classes=[
                            "predicted-letter"
                        ]
                    )

                    confidence_output = gr.Textbox(
                        label="Prediction Confidence",
                        interactive=False,
                        elem_classes=[
                            "confidence-box"
                        ]
                    )

                    predict_button = gr.Button(
                        "Predict Image",
                        variant="primary",
                        size="lg"
                    )

                gr.Markdown(
                    "## Word Formation"
                )

            with gr.Row():

                raw_word_output = gr.Textbox(
                    label="Raw Predicted Word",
                    interactive=False,
                    elem_classes=[
                        "word-output"
                    ]
                )

                corrected_word_output = gr.Textbox(
                    label="Corrected Word",
                    interactive=False,
                    elem_classes=[
                        "word-output"
                    ]
                )

            with gr.Row():

                append_letter_button = gr.Button(
                    "Append Predicted Letter",
                    variant="primary"
                )

                remove_letter_button = gr.Button(
                    "Remove Last Letter"
                )

                correct_word_button = gr.Button(
                    "Correct Word"
                )

                clear_word_button = gr.Button(
                    "Clear Word",
                    variant="stop"
                )

        # =====
        # Tab 2: Multiple Images to Word
        # =====

        with gr.Tab(
            "Multiple Images to Word"
        ):

            gr.Markdown(
                """
Upload multiple ASL images in the correct letter order.
Each image will be classified, and the predicted letters will
be joined into one word.
"""
            )

            multiple_images_input = gr.File(
                label="Upload Ordered ASL Images",
                file_count="multiple",
                file_types=["image"],
                type="filepath"
            )

```

```
)

multiple_predict_button = gr.Button(
    "Predict Complete Word",
    variant="primary",
    size="lg"
)

with gr.Row():

    multiple_raw_word_output = gr.Textbox(
        label="Raw Predicted Word",
        interactive=False,
        elem_classes=[
            "word-output"
        ]
    )

    multiple_corrected_word_output = gr.Textbox(
        label="Corrected Word",
        interactive=False,
        elem_classes=[
            "word-output"
        ]
    )

    multiple_gallery_output = gr.Gallery(
        label="Individual Image Predictions",
        columns=4,
        rows=2,
        height="auto",
        object_fit="contain"
    )

    multiple_status_output = gr.Textbox(
        label="Multiple-Image Prediction Status",
        interactive=False,
        elem_classes=[
            "status-output"
        ]
    )

# =====
# Tab 3: Models Comparison
# =====

with gr.Tab(
    "Models Comparison"
):

    gr.Markdown(
        """
## Saved Model Performance

This table is generated automatically from the six JSON
metrics files created after model training and evaluation.
        """
    )

    comparison_table_output = gr.Dataframe(
        value=create_models_comparison_table(),
        label="ASL Models Comparison",
        interactive=False,
        wrap=True
    )

    gr.Markdown(
        """
---
**ASL Recognition Research Application**
Static image classification for 36 classes: A-Z and 0-9.
        """
    )

    elem_classes=[
        "footer-text"
    ]

# =====
# Event Connections
# =====

model_selector.change(
    fn=handle_model_change,
    inputs=[
        model_selector
    ],
    outputs=[
        model_information,
        application_status
    ]
)

predict_button.click(
    fn=predict_single_image,
    inputs=[
        image_input,
        model_selector
    ],
    outputs=[
        top_predictions_output,
        predicted_letter_output,
        confidence_output,
        application_status
    ]
)

append_letter_button.click(
    fn=append_predicted_letter,
    inputs=[
        predicted_letter_output,
        current_word_state
    ],
    outputs=[
        current_word_state,
        raw_word_output,
        corrected_word_output,
        application_status
    ]
)

remove_letter_button.click(
    fn=remove_last_letter,
    inputs=[
        current_word_state
    ],
    outputs=[
        current_word_state,
        raw_word_output,
        corrected_word_output,
        application_status
    ]
)

correct_word_button.click(
    fn=correct_current_word,
    inputs=[
        current_word_state
    ],
    outputs=[
        corrected_word_output,
        application_status
    ]
)

clear_word_button.click(
    fn=clear_current_word,
    inputs=[],
    outputs=[
        current_word_state,
        raw_word_output,
        corrected_word_output,
        application_status
    ]
)

multiple_predict_button.click(
    fn=predict_multiple_images,
    inputs=[
        multiple_images_input,
        model_selector
    ],
    outputs=[
        multiple_raw_word_output,
        multiple_corrected_word_output,
        multiple_gallery_output,
        multiple_status_output
    ]
)
```

```
print("=" * 60)
print("Gradio Web Application Built Successfully")
print("=" * 60)
```

```
=====
Gradio Web Application Built Successfully
=====
```

In []:

```
# =====
# Step 16: Launch the Gradio Website
# This step launches the multi-model ASL recognition website
# and creates a temporary public Gradio link from Google Colab.
# =====

demo.queue(
    default_concurrency_limit=1
)

demo.launch(
    share=True,
    debug=True,
    show_error=True
)
```

Colab notebook detected. This cell will run indefinitely so that you can see errors and logs. To turn off, set debug=False in launch().
* Running on public URL: <https://ad503daab2525af179.gradio.live>

This share link is temporary and will last for up to 1 week (best effort). For free permanent hosting and GPU upgrades, run `gradio deploy` from the terminal in the working directory to deploy to Hugging Face Spaces (<https://huggingface.co/spaces>)

Loading Swin Transformer...
Swin Transformer Loaded Successfully