README: VideoMoCo Framework for Saudi Arabic Sign Language Recognition

1. Dataset Details

The experiments in this project use the **KARSL-502** dataset, a curated collection of video samples for Saudi Arabic Sign Language (SASL) recognition. The dataset includes 502 unique sign classes performed by native users in controlled environments. Each sign is represented through RGB videos with consistent resolution and frame rates. These videos are divided into:

- train/: Labeled training videos categorized by sign label in subfolders.
- val/: Validation videos used for linear evaluation and tuning.
- test/: Held-out video samples for final model evaluation and inference.

Each video is assumed to be in .mp4 format and typically ranges between **1–3 seconds**. The dataset is organized to support both **self-supervised pretraining** (VideoMoCo) and **supervised fine-tuning** stages.

```
D:\Exams\45\451\NewProject\Programs\DataSets\KARSL-502\
 ├── train/
|   ├── Hello/
|   ├── ThankYou/
|   └── ...
 ├── val/
 ├── test/
```

2. General Purpose of the Code

This project implements a **self-supervised contrastive learning framework (VideoMoCo)** tailored for Saudi Arabic Sign Language Recognition. The goal is to:

- **Learn temporal and spatial representations** of sign gestures from unlabeled video clips using the Momentum Contrast (MoCo) approach.
- **Improve downstream performance** on SASL classification tasks with minimal labeled data by first pretraining on unlabeled videos.
- **Enable robust fine-tuning** and transfer learning using a lightweight 3D CNN backbone (e.g., R3D).
- **Evaluate model performance** through standard classification metrics and visualize learned embeddings for interpretability.

The full pipeline includes:

- Preprocessing video data into consistent tensor formats.
- Building dataloaders and encoders for contrastive learning.
- Pretraining with Momentum Contrast.
- Linear evaluation and fine-tuning.
- Quantitative evaluation (accuracy, F1-score, confusion matrix).
- Optional t-SNE or UMAP visualization and real-time inference on new videos.

3. Folder Structure for Correct Execution

- To ensure all scripts run smoothly and can locate the necessary data and checkpoints, the following folder structure **must be strictly followed**:

```
Project Root/

|

├── DataSets/

|   └── KARSL-502/                # Main dataset folder

|       ├── train/                # Training videos (or preprocessed clips)

|       ├── val/                  # Validation videos

|       ├── test/                 # Testing videos

|      └── labels/                # Corresponding labels (CSV or TXT)

|

├── Model/                        # Output path for checkpoints, logs, and results

|   ├── Pretrained/               # Stores pretrained backbone (from contrastive learning)

|   ├── Checkpoints/              # Fine-tuned and classifier weights

|   ├── Evaluation/               # Evaluation results (metrics, plots)

|   └── Visualizations/           # t-SNE, UMAP, and attention maps

|

├── Scripts/

|   ├── 1_preprocessing.py

|   ├── 2_dataloader.py

|   ├── 3_backbone_model.py

|   ├── 4_videomoco_model.py

|   ├── 5_train_moco.py

|   ├── 6_linear_eval.py
|   ├── 7_finetune.py
|   ├── 8_evaluate.py
|   └── 9_visualize.py
|
├── Inference/
|   ├── 10_infer.py              # Script to run inference on a new video
|   └── sample_video.mp4         # Example input video
|
└── README.md                    # This file
```

All output (checkpoints, logs, evaluation results) will be saved in:

E:\Mahmoud\Exams\46\461\New-Papers\Paper3-Atlam\Model

4. Order of Execution of the Codes

To correctly run the full experimental pipeline for **Self-Supervised Learning for Saudi Arabic Sign Language Recognition**, follow this step-by-step execution order:

Step 1: Preprocessing

- Run: 1_preprocessing.py
- Purpose: Extract video frames, apply resizing, normalization, and optional temporal slicing.
- Output: Preprocessed clips saved in DataSets/KARSL-502/train, val, and test.

Step 2: Data Loading Setup

- File: 2_dataloader.py
- Used as a module for all training and evaluation scripts.
- Purpose: Define the dataset class and augmentations for contrastive and supervised training.

Step 3: Backbone Definition

- File: 3_backbone_model.py
- Purpose: Defines the 3D CNN (R3D or I3D) used across all stages. No execution needed directly.

Step 4: Momentum Contrast (MoCo) Model

- File: 4_videomoco_model.py
- Purpose: Instantiates the MoCo framework for contrastive training. Also used as a module.

Step 5: Self-Supervised Pretraining

- Run: 5_train_moco.py
- Purpose: Trains the MoCo model using unlabeled video data.
- Output: Encoder checkpoints saved to Model/Pretrained.

Step 6: Linear Evaluation

- Run: 6_linear_eval.py
- Purpose: Freezes the encoder and trains a linear classifier on labeled data.
- Output: Accuracy metrics and weights saved to Model/Checkpoints.

Step 7: Fine-Tuning

- Run: 7_finetune.py
- Purpose: Fine-tunes both encoder and classifier using full labeled dataset.
- Output: Fine-tuned model saved to Model/Checkpoints.

Step 8: Evaluation

- Run: 8_evaluate.py
- Purpose: Evaluate model using Top-1, Top-5 accuracy, F1-score, confusion matrix, and precision-recall plots.
- Output: All results saved under Model/Evaluation.

Step 9: Visualization (Optional)

- Run: 9_visualize.py
- Purpose: Generate t-SNE/UMAP plots of embeddings or temporal attention maps.
- Output: Plots saved to Model/Visualizations.

Step 10: Inference

- Run: 10_infer.py
- Purpose: Predict sign label for a new video clip using trained model.
- Input: Place new video in Inference/, results will be printed or saved.

5. Requirements for Each Code

This section lists the **hardware**, **software**, and **data requirements** for successfully executing each component in the Saudi Arabic Sign Language Recognition pipeline.

🔧 Hardware Requirements

- **GPU**: NVIDIA GPU with at least 8GB VRAM (e.g., RTX 3060 or better) is highly recommended for training.
- **RAM**: Minimum 16 GB.
- **Storage**: At least 50 GB of free space (for videos, features, and model checkpoints).
- **CPU**: Quad-core or better.

💻 Software Requirements

- Python version: **3.8+**
- Required Libraries:

```
pip install torch torchvision torchaudio

pip install opencv-python pandas scikit-learn matplotlib seaborn umap-learn

pip install tqdm einops

pip install plotly tensorboard
```

Data Requirements and Folder Structure

Main Dataset Path:

makefile

CopyEdit

```
D:\Exams\45\451\NewProject\Programs\DataSets\KARSL-502
```

Expected Substructure:

```
KARSL-502/
 ├── train/
```

```
|    ├── class_001/
|   |    ├── video1.mp4
|   |    ├── video2.mp4
|   └── ...
 ├── val/
|    ├── class_001/
|   |    ├── video3.mp4
|   └── ...
 ├── test/
|    ├── class_002/
|   |    ├── video4.mp4
|   └── ...
```

Output Paths:
All processed data, models, results, and visualizations are saved under

E:\Mahmoud\Exams\46\461\New-Papers\Paper3-Atlam\Model

Subfolders will be created dynamically:

```
Model/
 ├── Pretrained/         # Checkpoints from self-supervised MoCo training
 ├── Checkpoints/        # Fine-tuned models and linear classifier
 ├── Evaluation/         # Accuracy, confusion matrix, metrics
 ├── Visualizations/     # t-SNE, UMAP plots or attention maps
 ├── Inference/          # Output for new predictions
```

6. Considerations and Notes

This section outlines important considerations, implementation details, and best practices to ensure successful execution and reproducibility.

✅ General Considerations

- **Frame Rate and Clip Length**: All video clips are assumed to be preprocessed to the same frame rate (e.g., 25 FPS) and clip length (e.g., 16 or 32 frames). Ensure uniformity across the dataset.
- **Video Format**: Input videos must be in .mp4 format, readable by OpenCV.
- **Label Consistency**: Folder names under train/, val/, and test/ are treated as class labels. Ensure consistent naming conventions across all splits.
- **Contrastive Learning Setup**: Ensure that the batch size is sufficient to populate the momentum queue effectively during self-supervised training.

⚠️ Key Notes

- **GPU Memory Management**: Training the 3D CNN (especially I3D) can be memory-intensive. Adjust batch_size and clip resolution if you encounter out-of-memory errors.
- **MoCo Queue Length**: The dynamic queue size in the MoCo implementation should be tuned based on the dataset size. A typical size is 4096–8192.
- **Training Duration**: Self-supervised MoCo training may require significant epochs (~200+) for convergence.
- **Pretraining and Transfer**: The pretrained backbone must be correctly frozen during linear evaluation, and then unfrozen for fine-tuning.
- **UMAP/t-SNE**: These visualizations are compute-intensive. Reduce the number of samples or use PCA as a preprocessing step to improve performance.

📌 Reproducibility

- Set a consistent random seed (torch.manual_seed) across all scripts.

- Log experiment configurations and hyperparameters.
- Use torch.backends.cudnn.deterministic = True if exact reproducibility is required.

Supplementary Note on CatsDogs_Model1.py and cats_and_dogs_small_1.h5

The files CatsDogs_Model1.py and its trained model cats_and_dogs_small_1.h5 are included as part of the supplementary materials to illustrate the initial prototype pipeline used during the early phase of our research. This preliminary model was developed to validate key components of the training infrastructure—such as data loading routines, early stopping criteria, model checkpointing, and GPU-accelerated training—in a controlled environment using a well-known benchmark dataset.

While the final implementation is a sophisticated self-supervised 3D CNN architecture for video-based sign language recognition, the core experimental scaffolding (e.g., data generators, Keras-based training logic, and evaluation scripts) was initially verified through the static image classification task. This validation step ensured the correctness and robustness of training utilities before scaling up to high-dimensional video representations and contrastive learning objectives.

Therefore, the inclusion of this earlier code and model provides valuable context for the design evolution and infrastructure decisions that led to the final architecture, and supports reproducibility by offering a stepwise trace of the experimental pipeline's development.