ChatGPT

End-to-End Implementation Plan for PoolNet on YouTube-VIS Dataset

1. Data Preprocessing

Frame Extraction from Videos: Start by extracting individual frames from each video in the YouTube-VIS dataset. The YouTube-VIS dataset consists of 2,883 high-resolution YouTube videos (2,238 train, 302 val, 343 test) [1]. Each video is a short clip (up to ~30 seconds) containing diverse real-world scenes. We will use a tool like OpenCV or FFmpeg to convert these videos into sequences of image frames. For example, using FFmpeg one can run:

```
ffmpeg -i input_video.mp4 -vsync 0 -q:v 2 frames/frame_%04d.jpg
```

This command decodes all frames (`-vsync 0`) and saves them as JPEG images (`-q:v 2` for high quality). Using Python and OpenCV, the equivalent is:

```
import cv2
vidcap = cv2.VideoCapture('input_video.mp4')
success, image = vidcap.read()
frame_idx = 0
while success:
    cv2.imwrite(f"frames/frame_{frame_idx:04d}.png", image)
# save frame as PNG or JPEG
    success, image = vidcap.read()
    frame_idx += 1
```

This will produce a sequence of images for each video, e.g. `video_001_frame_0001.jpg, video_001_frame_0002.jpg, ...`. We repeat this for all videos in the 11 GiB subset. Ensure to maintain the **original frame rate** (do not drop or duplicate frames) so that consecutive frames remain temporally coherent.

Chunking into 20–30 Frame Sequences: Once frames are extracted, group them into chunks of 20–30 consecutive frames. Each chunk will serve as one data sample (representing a short subsequence of the video). Using contiguous frames preserves the temporal continuity needed for SfM. For example, if a video has 100 frames, it can be split into chunks like frame 0–19, 20–39, 40–59, 60–79, 80–99 (5 chunks of 20 frames each). If the last segment is shorter than 20, we can either pad it (by duplicating the last frame or reflecting frames) or discard it (to stick within the 20–30 range). Each chunk should be saved in a separate folder or with a naming convention that indicates which video and which frame range it covers. We also create a **metadata index** (e.g. a CSV file) listing each chunk with fields: `video_id, start_frame,`

`end_frame, chunk_id`. This will help track the origin of each chunk and later merge it with ground truth metrics.

Format Consistency (Resolution & Aspect Ratio): Videos in YouTube-VIS may have varying resolutions and aspect ratios. To ensure consistency, decide on a standard image size or preprocessing method for the frames: - One approach is to **rescale** or **pad** frames to a fixed resolution (e.g. 480p or 224x224 if we plan to feed directly to ViT). However, since PoolNet is designed to handle arbitrary image sizes using a patch-based ViT, we don't strictly require uniform resizing. We can maintain the original resolution and aspect ratio of each video frame to preserve all information. The ViT encoder will adapt via its patch mechanism (with position embeddings interpolation if needed). - What we do ensure is that all frames are in a consistent format (e.g. 3-channel RGB images in PNG/JPEG) and color space. If any frames are grayscale or have an alpha channel, convert them to RGB. - If we choose to standardize size for memory reasons, we could scale all frames so that the larger dimension is, say, 480 pixels (maintaining aspect ratio) and pad the smaller dimension with black borders to reach a square (if needed). This keeps the field of view roughly consistent across data. But in general, preserving aspect ratio is important, so padding is preferable to distortion.

After processing, each chunk will be a folder of 20–30 images with consistent formatting. The metadata file can also record each chunk's resolution (if we did not standardize it) and aspect ratio so that we know what was fed into the model.

2. COLMAP Integration for SfM Ground Truth

Automated COLMAP Reconstruction per Chunk: For each frame chunk, we will run COLMAP (a state-of-the-art Structure-from-Motion tool) to perform sparse 3D reconstruction. COLMAP will attempt to find feature correspondences between frames and estimate camera poses and a sparse point cloud. We will automate this via the COLMAP command-line interface for batch processing [2]. The steps for each chunk are: 1. **Feature Extraction:** `colmap feature_extractor --database_path <chunk_folder>/database.db --image_path <chunk_folder>/images --ImageReader.single_camera 1`. This extracts SIFT features from each frame (using `single_camera` if we assume all frames from one video share intrinsics, which is reasonable). COLMAP can use the GPU for feature extraction if built with CUDA – ensure to enable that for speed. 2. **Feature Matching:** `colmap exhaustive_matcher --database_path <chunk_folder>/database.db`. This finds matching feature points between all pairs of frames in the chunk (for 20–30 images, exhaustive matching is fine; for larger numbers, one could use vocabulary tree matching, but here it's manageable). 3. **Sparse Mapping:** `colmap mapper --database_path <chunk_folder>/database.db --image_path <chunk_folder>/images --output_path <chunk_folder>/sparse`. This runs the SfM incremental reconstruction. If successful, it will produce a sparse model in `<chunk_folder>/sparse/0` (with files like `cameras.bin`, `images.bin`, `points3D.bin`).

We can automate these steps with a Python script using `subprocess` calls for each chunk, or use the `colmap automatic_reconstructor` command for simplicity:

```
colmap automatic_reconstructor \
    --workspace_path <chunk_folder> \
    --image_path <chunk_folder>/images \
```

```
    --workspace_format COLMAP \
    --SparseMapper.max_num_models 1
```

This single command performs feature extraction, matching, and mapping automatically [2], producing the result in `<chunk_folder>/outputs` (depending on COLMAP version). We use `max_num_models 1` to only get one model per chunk (COLMAP can sometimes produce multiple disconnected reconstructions; we focus on the largest one).

Collecting Ground Truth Metrics (Ts, To, Vr, Vt): After COLMAP runs on a chunk, we gather the following metrics, which will serve as ground truth labels for that sequence: - **Ts (Success/Failure flag):** This is a binary indicator of whether COLMAP succeeded in reconstructing the scene. If COLMAP registers at least a certain number of frames into a coherent model, we consider it a success (Ts = 1). If it fails to find a consistent model (e.g., zero or only 1 frame oriented), that's a failure (Ts = 0). In practice, we can define Ts = 1 if **>= 3 frames** were reconstructed (since a valid 3D reconstruction generally needs 3 or more images) or if COLMAP outputs a sparse model with points; Ts = 0 if fewer than 3 frames were registered. - **To (Used/Total frames ratio):** This is the fraction of frames in the chunk that were used in the final reconstruction [3]. COLMAP's output `images.bin` (or `images.txt` if converted to text) lists all frames that got camera poses. We compute `To = (number of frames with estimated pose) / (total frames in chunk)`. For example, if out of 25 frames in the chunk, COLMAP could only register 20 (perhaps others were blurry or had no matches), then `To = 20/25 = 0.8`. This ratio reflects how much of the data was useful. If Ts = 0 (failure), To can be considered 0 (0 frames used). - **Vr (Rotational diversity):** A measure of the range of camera orientations in the reconstruction [4]. Using the camera pose estimates from COLMAP, we calculate how varied the camera rotations are. Implementation-wise, we can take the set of rotation matrices or quaternions from COLMAP and compute, for instance, the maximum pairwise angle between camera orientations or the variance of the camera viewing directions. A simple proxy is to take the average angle difference (in degrees) between each camera's orientation and the first camera, or find principal components of the orientation distribution. For example, if the camera only pans slightly, Vr will be small; if the camera views the scene from many angles (e.g., turning around), Vr will be large. We will normalize Vr to a 0–1 range by dividing by 180° (since 180° is the maximum distinct rotation between two views). - **Vt (Translational diversity):** A measure of the range of camera positions (translation) in the reconstruction [4]. Using the camera centers from COLMAP (the translation components of the poses, ignoring the global scale), we compute how much the camera moved. For example, one can compute the diameter of the camera trajectory (max distance between any two camera positions) or the standard deviation of positions. Since SfM reconstruction is up to scale, we can scale the translations so that the average nearest-neighbor distance between camera positions is fixed (to normalize scale) before computing this diversity. Vt will be higher for sequences where the camera moves through the scene significantly, and lower if the camera stays roughly in one spot. We likewise normalize Vt into [0,1] (e.g., by dividing by an expected max distance or using a relative measure).

All these metrics can be obtained by parsing COLMAP's output. COLMAP provides a utility to convert the binary model to text (`colmap model_converter --input_path sparse/0 --output_path sparse/0 --output_type TXT`), yielding `cameras.txt`, `images.txt`, and `points3D.txt`. We can read `images.txt` to get the list of registered images and their poses:

```
# Image list with two lines of data per image:
# IMAGE_ID, QW, QX, QY, QZ, TX, TY, TZ, CAMERA_ID, NAME
1 0.9186 0.3951 -0.0317 -0.0045  0.12 0.05 1.37  1 frame_0000.jpg
...
```

From this, counting lines gives number of images used (for To and Ts), and QW-QZ (quaternion) plus T vector give orientation and position for Vr and Vt calculations. We will implement a small parser or use **PyCOLMAP** (a Python API for COLMAP) to retrieve these values programmatically. The output for each chunk will be recorded in our metadata as `Ts, To, Vr, Vt`. These serve as the **ground truth labels** that our PoolNet model will learn to predict for any given sequence.

Quality check: If some chunks are extremely challenging (e.g., all frames are blank or no overlap), COLMAP might produce no model at all. Those will have Ts=0, and we should include them as negative samples. We should verify the distribution of Ts (expect many successes for steady scenes and failures for very random/noisy ones) to ensure we have a mix for training.

3. Data Augmentation for Robustness

To improve the model's generalization and make it robust to various conditions, we will apply data augmentation to the frame sequences during training. The key principle is to **preserve geometric consistency across frames** – augmentations should not break the relative camera poses or the scene structure within a chunk: - **Spatial Shifts/Crops:** We apply a random crop or translation to the sequence as a whole. For example, randomly select an offset (dx, dy) (say up to 5-10% of image dimensions) and crop all frames in the chunk by that offset (i.e., shift the field of view). By applying the same crop to every frame in the sequence, we simulate a camera that was aimed slightly differently, without altering the *relative* motion between frames. This preserves SfM outcomes (the geometry of the sequence is unchanged aside from a consistent crop). We can implement this by determining the crop bounds once per sequence (e.g., for each sequence in a batch, randomly choose dx,dy) and using OpenCV or PIL to crop each image accordingly. - **Zoom (Scaling):** Similar to cropping, we can simulate a zoom-in by cropping and resizing frames, or a zoom-out by adding a border. Again, the same scaling is applied to all frames in the chunk. For instance, randomly choose a scale factor between 0.8 and 1.2 and resize all frames by that factor (possibly cropping or padding to return to original resolution). This changes the apparent field of view but keeps consistent geometry across frames. - **Contrast & Brightness Jitter:** We introduce photometric variations like adjusting brightness, contrast, or gamma. These can be applied *per frame* independently to mimic real conditions (like flickering lighting) because they don't affect geometry. However, to avoid excessively confusing the model with drastic per-frame changes, we might also apply a milder consistent adjustment across the whole sequence (e.g., slightly increase contrast on all frames, plus a small random per-frame noise). Libraries like Albumentations or `torchvision.transforms.ColorJitter` can be used to randomly perturb brightness, contrast, saturation, and hue. For example:

```
import torchvision.transforms as T
color_jitter = T.ColorJitter(brightness=0.3, contrast=0.3, saturation=0.2,
hue=0.1)
augmented_frames = [color_jitter(frame) for frame in frames]
```

This would apply random color augmentations to each frame. We could also fix one random setting per sequence for a consistent tint (e.g., give all frames a slightly warmer tint) to simulate a different camera color balance. - **Horizontal Flip (if applicable):** If the content orientation is not critical (non-text scenes), we could horizontally flip all frames in a sequence with 50% chance. This doubles the data and doesn't affect SfM geometry besides mirroring the coordinate system (which COLMAP results would equally mirror – since we're not rerunning COLMAP on augmented data, the ground truth is still valid if we consider that a mirrored sequence should have the same SfM success as original).

By applying these augmentations, we ensure the model sees a variety of appearances for the same underlying sequence geometry. **Crucially, spatial transforms are done identically on each frame in a chunk**, so that the relative camera poses remain as originally captured. This way, the COLMAP-derived ground truth (Ts, To, Vr, Vt) remains correct for the augmented sequence. (For example, if a sequence was successful, a cropped or brightness-changed version of it should also be successful in SfM, barring extreme changes). We will implement these augmentations in the data loading pipeline (so that each epoch, the sequences can be augmented on-the-fly, providing additional diversity).

Additionally, we maintain copies of the original sequences (or use augmentation probabilistically) so the model sometimes sees the raw data as well. All augmented data will be labeled with the same ground truth as the original sequence.

4. Model Architecture

Our model, **PoolNet**, will consist of a frame-wise encoder and a sequence aggregator to predict the SfM success and quality metrics for each chunk. The design choices, as described in the proposal, are: - A pre-trained Vision Transformer (ViT) as an image encoder, which can handle arbitrary image sizes by processing patches. - A temporal sequence model (LSTM) to pool information across multiple frames. - Multi-task output heads to predict the four target values (Ts, To, Vr, Vt).

Vision Transformer Encoder (Frame-level): We will use the `google/vit-base-patch16-224` model from Hugging Face Transformers as the backbone for frame encoding. This is a ViT base architecture (12 layers, hidden size 768, 16x16 patches) pre-trained on ImageNet-21k, which has ~86 million parameters. We choose this because it provides robust feature extraction and is already trained on a wide variety of images, saving us training time. The ViT will take each frame (after necessary preprocessing like normalization) and output a feature representation. Concretely, we will remove ViT's classification head and use the final hidden state of the `[CLS]` token (a 768-D vector) as the embedding for that frame. This embedding will encapsulate information about the frame's content in a way that is invariant to image size and fairly robust to varied scenes (thanks to pre-training).

To support **variable image resolutions**, we will leverage the ViT's patch mechanism. The model's positional embeddings will be interpolated if the number of patches differs from 14x14 (for 224×224) – the Hugging Face `ViTModel` can handle this internally by resizing position embeddings. Alternatively, we could rescale frames to 224×224 for input; but to preserve detail, we will feed images at a moderate resolution (e.g., 224 or 384 on the long side) and let the ViT handle the patches. This preserves the spirit of scale-invariant encoding.

We may **fine-tune** the ViT on our data or at least the later layers, since our task (SfM outcome prediction) is quite different from ImageNet classification. To manage training time on a single GPU, we might freeze the early layers of ViT and fine-tune only the higher layers and subsequent components, reducing the number of trainable parameters initially.

Sequence Aggregation with LSTM: To aggregate frame-level information, we use a Long Short-Term Memory network (LSTM), a type of RNN well-suited for sequence data. The LSTM will process the sequence of frame embeddings (the 768-D vectors from ViT for each frame in order) and produce a single representation for the entire sequence. Specifically, we can use a single-layer (or multi-layer) LSTM with a hidden size of, say, 256 or 512 (or even 768 to match the ViT output size for ease). The LSTM will read in frames 1 through N (N ~20–30) and we will use the **final hidden state** as the sequence embedding representing the whole chunk. This final LSTM state should capture the temporal dynamics and context across frames – e.g., camera motion, scene changes, etc., which correlate with SfM success. An alternative could be to use transformers for sequence modeling or even simple pooling, but an LSTM is a straightforward choice given the moderate sequence length and its ability to handle variable-length input.

The LSTM allows **variable sequence lengths** by design. We will use batch processing with padding if necessary: frames in a batch can be padded to the length of the longest sequence in that batch, and we'll use PyTorch's `pack_padded_sequence` so the LSTM ignores the padding. Since our chunks are all 20–30 frames, length variability is small, but we will implement this to be safe (especially if some sequences at dataset edges have fewer frames).

Multi-Task Output Heads: From the LSTM's final output (or we could also use a pooled representation of all LSTM outputs, but final output is simplest as it already has seen all frames), we will have two heads: - A **binary classification head** for **Ts**. - A **regression head** for **To, Vr, Vt**.

The classification head will be a single fully connected layer (Linear layer) that maps the LSTM hidden state to a single logit, which we will apply a sigmoid to get $P(\text{success})$. Alternatively, we could output two logits and softmax for [failure, success], but a single sigmoid output is sufficient for binary classification.

The regression head can be a fully connected layer that outputs 3 values (for To, Vr, Vt). We might apply an activation if we want to constrain outputs (e.g., a sigmoid to output 0–1 since our targets are normalized between 0 and 1). However, it may be better to allow the network to output any real number and just train it to match the targets (which are in [0,1]). For simplicity, we can have the regression head be linear (no activation) and use appropriate loss normalization.

By using separate heads, we allow the network to specialize: one part of the weights to handle the binary decision and another to handle numeric estimations. The earlier layers (ViT + LSTM) are shared, learning a representation useful for all tasks. This is a **multi-task learning** setup.

Below is a high-level skeleton of the model in PyTorch-like pseudocode:

```
import torch
import torch.nn as nn
from transformers import ViTModel
```

```
class PoolNetModel(nn.Module):
    def __init__(self):
        super().__init__()
        # Load pre-trained ViT model (base patch16)
        self.vit = ViTModel.from_pretrained('google/vit-base-patch16-224-in21k')
        # Optionally freeze lower layers of ViT to save memory
        # for param in self.vit.parameters():
        #     param.requires_grad = False
        # (We can selectively unfreeze later.)

        vit_hidden_dim = self.vit.config.hidden_size  # 768 for vit-base
        self.lstm = nn.LSTM(input_size=vit_hidden_dim,
hidden_size=vit_hidden_dim//2,
                            num_layers=1, batch_first=True,
bidirectional=False)

# Use hidden_dim/2 for LSTM to reduce parameters; can also use 768 (same as ViT
output).

        # Output heads
        self.fc_ts = nn.Linear(vit_hidden_dim//2, 1)         # Ts head (1 logit)
        self.fc_reg = nn.Linear(vit_hidden_dim//2, 3)        # To, Vr, Vt head
(3 values)

    def forward(self, frame_batch):
        # frame_batch shape: (B, T, C, H, W) where B = batch size, T = #frames
        B, T, C, H, W = frame_batch.shape
        # Merge batch and time for ViT processing
        frames_flat = frame_batch.view(B*T, C, H, W)
        # Pass through ViT to get sequence of patch embeddings; take CLS token
output
        vit_outputs = self.vit(frames_flat)
        frame_feats = vit_outputs.last_hidden_state[:, 0, :]   # (B*T, 768)
        frame_feats = frame_feats.view(B, T, -1)               # (B, T, 768)

        # Pass frame features through LSTM
        # We assume sequences are already padded to equal length T here.
        lstm_out, (h_n, c_n) = self.lstm(frame_feats)          # lstm_out: (B,
T, hidden), h_n: (1, B, hidden)
        seq_emb = lstm_out[:, -1, :]                           # take output at
last time-step (B, hidden_dim)

        # Heads
        ts_logit = self.fc_ts(seq_emb).squeeze(-1)             # (B,) binary
logit
        preds_reg = self.fc_reg(seq_emb)                       # (B, 3) for To,
Vr, Vt
        return ts_logit, preds_reg
```

In this design, we flatten the batch of frames to feed through ViT in one go (which is efficient, as ViT can process a batch of images). We then reshape back to sequence [B, T, features] for LSTM. The LSTM's output at the final time step (`seq_emb`) is used for prediction. We output `ts_logit` (which will be fed to a sigmoid in the loss function for classification) and `preds_reg` (which are raw values for To, Vr, Vt).

Note on dimensions: We chose LSTM hidden size = ViT hidden/2 (384) to reduce compute; one could also use the full 768 for the LSTM hidden size for simplicity (and possibly better performance at cost of memory).

Memory Considerations: A 20-frame sequence fed as 20 separate forward passes through ViT is somewhat inefficient. In the above approach, we feed B*T images at once. For example, with batch size B=4 and T=25, that's 100 images through ViT in parallel, which a 3090 GPU can handle if images are 224px. If images are larger (say 480px), the patch count increases (e.g., 30x30 patches = 900 patches per image vs 196 for 224px), which increases ViT memory usage. We might need to adjust image size or batch size accordingly. The model as defined can be trained on a single RTX 3090 with a reasonable batch size (likely 1–4 sequences per batch depending on resolution).

5. Training Protocol

Loss Functions: We are dealing with a multi-task problem (one classification and three regression targets). We will define a composite loss $\mathcal{L} = \mathcal{L}*{Ts} + \lambda \mathcal{L}*$, balancing classification and regression: - For **Ts (success/failure)**, use **binary cross-entropy (BCE) loss**. If `p` is the predicted probability (after sigmoid) for success and `y` is 0/1 ground truth, then $\mathcal{L}*{Ts} = -[y \log p + (1-y)\log(1-p)]$. In PyTorch, we can use `nn.BCEWithLogitsLoss` (which applies sigmoid internally). - For **To, Vr, Vt**, use a regression loss. A good choice is **Mean Squared Error (MSE)** or **Mean Absolute Error (L1)** for each. MSE will heavily penalize larger errors, while MAE is more robust to outliers. We can start with MSE: $ \mathcal{L}$ are predictions and m ground truth values (all in [0,1] range). Alternatively, treat each with its own weight if they have different scales of importance. } = \frac{1}{3}\sum_{m \in {To,Vr,Vt}} (\hat{m} - m)^2$, where $\hat{m

We might weight the regression vs classification losses. Since Ts is our primary objective (according to the proposal) [5], we ensure $\mathcal{L}*{Ts}$ has a high priority. One strategy is: - Always include $\mathcal{L}*`. - Only compute the regression loss for sequences where Ts = 1 (i.e., only when a reconstruction was successful do we care about the quality metrics). This matches the idea that Vr and Vt are only meaningful if SfM succeeded [5]. Implementation: if a sample has ground truth Ts=0, we can set its regression loss weight to 0 (so failing sequences do not influence To, Vr, Vt learning). - Use a weight $\lambda < 1$ (e.g., 0.5) to down-weight regression a bit relative to classification, so that the network first and foremost learns to distinguish success vs failure. For example: $\mathcal{L} = \text{BCE}*{Ts} + 0.5 \times (\text{MSE}*} + \text{MSE}*{Vr} + \text{MSE}*)$ (if Ts=1 for that sample; if Ts=0, skip the second term or consider it zero).

We will monitor both parts during training to ensure the model is learning all tasks appropriately.

Evaluation Metrics: During training/validation, we will measure: - **Accuracy or F1 for Ts:** How often the model correctly predicts success vs failure. Because the dataset might be imbalanced (perhaps more successes than failures or vice versa), we'll track overall accuracy and possibly the confusion matrix. If failures are rare, we might use F1-score for the "failure" class to ensure we capture that performance. - **Regression error for To, Vr, Vt:** We can report Mean Absolute Error (MAE) or Root Mean Squared Error

(RMSE) for each of these on the validation set. MAE is intuitive (average absolute difference in the ratio or diversity metrics). Since these metrics are normalized 0–1, an MAE of 0.1 means on average the prediction is off by 10% of the range. - Additionally, we can compute **R^2 (coefficient of determination)** for the regression targets to see how much variance is explained by the model, and **binary AUC** for Ts if needed. - If point cloud size or other metrics were of interest, we could include them, but our focus is Ts, To, Vr, Vt.

Training Procedure: 1. **Train/Validation Split:** We will use the official YouTube-VIS training set for training and the validation set for validation (and hold out the test set for final evaluation). This ensures no overlap of videos between train/val. Concretely, use the 2,238 training videos' chunks as training data, and the 302 validation videos' chunks as the validation set. This is important because multiple chunks often come from the same video, and we don't want chunks from one video in both train and val (to avoid the model memorizing video-specific characteristics). Our metadata index from preprocessing can help filter chunks by video ID for this split. 2. **Batching and DataLoader:** We will implement a PyTorch `Dataset` that, given an index, returns a sequence of frames (as tensors) and the target values (Ts, To, Vr, Vt). The dataset **getitem** will: - Load the frames for that chunk (either from disk or from a cached list if memory allows). We can use PIL or OpenCV to read images, apply any **random augmentations** (for training), then normalize pixel values (using ImageNet means/std for ViT). - Stack frames into a tensor of shape (T, 3, H, W). If we have varying T per sample, we can make T max (30) and pad shorter sequences with zeros (and inform the DataLoader via a custom collate function to pack sequences). - Return the frames tensor and a target tensor [Ts, To, Vr, Vt].

We will use a custom collate to batch these. Alternatively, since our sequences are similar length, we might simply shuffle and batch sequences of the same length together to avoid padding (though that's minor optimization).

The DataLoader will handle shuffling for training. For validation, no augmentations and no shuffle (or fixed shuffle) to evaluate on consistent data.

1. **Optimizer and Training Schedule:** Given the model size (ViT ~86M, LSTM ~ a few million, heads small), training on a single RTX 3090 is feasible. We will use an optimizer like **AdamW** (Adam with weight decay, commonly used for transformer fine-tuning) with an initial learning rate, e.g., 1e-4 for the heads and LSTM, and a smaller LR (1e-5) for the ViT backbone if we are fine-tuning it, to avoid forgetting pre-trained features. We can employ a learning rate scheduler (e.g., cosine decay or step decay) over the course of training.

We'll train for a sufficient number of epochs until convergence. Since the dataset (2,238 training videos, each potentially yielding several chunks) might yield on the order of ~10k chunks (just an estimate; e.g., if each video ~30 frames on average, one chunk each; if longer videos produce multiple chunks, could be more), the dataset size is moderate. We might train for, say, 20–50 epochs, monitoring validation loss to avoid overfitting.

If class imbalance is significant (say COLMAP fails on far fewer sequences than it succeeds, or vice versa), we may incorporate techniques like class weights in the BCE loss or oversample failure cases so the model gets enough signal on them.

1. **Validation and Tuning:** After each epoch (or every few epochs), evaluate on the validation set: compute Ts accuracy and regression MAE. Use this to tune any hyperparameters (like loss weighting

λ, learning rate, etc.). We expect the model to gradually learn to predict these measures. If we see that Ts accuracy is high but regression error is poor, we might increase weight on regression losses. If Ts accuracy is lagging, increase its weight or oversample Ts=1 vs Ts=0 as needed.

Throughout training, we also ensure to log metrics and perhaps save model checkpoints. We can use a framework like PyTorch Lightning or just manual training loop with periodic `torch.save` of the model state dict.

6. Evaluation and Scaling

Benchmarking Against COLMAP: The ultimate purpose of PoolNet is to serve as a fast predictor of SfM success to avoid running COLMAP unnecessarily on poor data [6] [7]. Therefore, we will compare the inference time of our trained model to COLMAP's SfM processing time: - For a given sequence (e.g., 25 frames), measure how long COLMAP takes to produce the result (feature extraction + matching + mapping). Suppose COLMAP takes *X* seconds (which could be, for example, 5-30 seconds depending on sequence length and content on a CPU). - Measure how long our model takes to compute Ts, To, Vr, Vt for the same sequence. This involves a forward pass through ViT for 25 images and the LSTM, on GPU. This might take on the order of 100ms–300ms for 25 images on an RTX 3090 (rough estimate, depending on resolution and batch optimizations). We will empirically time this using the PyTorch `torch.cuda.Event` or Python timers. - The **speed-up** is roughly `X / model_time`. We expect a significant speed-up (potentially 10× or more). For example, if COLMAP took 10 seconds and the model 0.2 seconds, that's a 50× speed-up. This demonstrates the efficiency gain.

We will report average COLMAP time vs average model prediction time on a sample of sequences. Since COLMAP's runtime might vary with content (more features = longer matching, etc.), we can take an average over many sequences for a fair comparison. The model's runtime should be quite stable per sequence size, and importantly it scales roughly linearly with number of frames and image size (which we can also test).

Prediction Quality vs Ground Truth: We also evaluate the model's predictive performance: - Classification: accuracy, precision/recall for predicting SfM success (Ts). We want the model to catch most failures (high recall on Ts=0) and not flag too many false failures (precision). If the model predicts Ts=0 for sequences that COLMAP would succeed on, that's a false alarm (could cause us to skip usable data). If it predicts Ts=1 for sequences that actually fail in COLMAP, that's a missed detection. We will analyze these cases on the validation (and test) sets. - Regression: correlation between predicted To, Vr, Vt and the ground truth. We can plot predicted vs true values to see how well the model is calibrating these. We may find, for instance, that the model can predict To (the fraction of images used) with a small error (e.g., ±0.1), which could be very useful in deciding data value. Similarly, we check if predicted Vr and Vt align with actual diversity measures (these might be harder to predict, but we aim for reasonable approximation).

Scaling to Longer Sequences: Our model is trained on 20–30 frame chunks, but we should test its performance on longer sequences as well. Because of the LSTM architecture, in principle we can feed a longer sequence (say 50 frames) and it will process it (with the caveat of fixed positional embeddings of ViT needing interpolation, which is handled). We will experiment by feeding some sequences of, e.g., 60 frames (maybe by concatenating two chunks or using a longer continuous segment) and see how the model behaves: - For speed, a 60-frame sequence means 60 ViT forwards. This might still be fine if done in batches (e.g., batch of 60 images). - If memory is an issue (60 high-res images might not fit at once), we could split into two passes (or process sequentially). - We expect the model's Ts prediction to still be meaningful – since

Ts is largely about whether SfM would succeed. If anything, more frames could only help SfM, so a long sequence might virtually always have Ts=1 if any subset would. The model might not have seen 60-length in training, but LSTM can handle it; we need to see if it generalizes the concept. We can fine-tune or at least verify on some known long sequences (perhaps using part of ScanNet data which have 1000 frames but can choose a segment).

If needed, we could adopt a sliding window approach for long videos: run the model on overlapping 30-frame windows and aggregate the predictions (e.g., a long video is "good for SfM" if most windows are predicted Ts=1). But this is more of an application detail; the model itself can be tested directly on longer input.

Scaling Batch Size: The model can also be applied to many sequences in parallel (batching). On a 24GB GPU, we might handle, say, 4 sequences of 25 frames (100 images) at once as mentioned. For inference throughput, this is beneficial. We should test memory usage: - If each image is 224×224, ViT patch count = 196, the model is quite memory-light. If images are larger (say 480p ~ 854×480, patch count ~ (854/16) *(480/16) ≈ (53*30) = 1590 patches per image), that significantly increases memory. We might then only batch 1–2 sequences at a time to avoid OOM. - We can profile how memory scales with image size and decide on a fixed input resolution for the model if needed (e.g., maybe standardize all frames to 320px on the long side to balance detail vs memory). - Training batch size was likely small, but for **inference** we can utilize more of the GPU.

Ultimately, on a single RTX 3090, we anticipate being able to process sequences much faster than COLMAP. We will document that, for example: "Our model processes ~5 sequences per second, whereas COLMAP processes ~1 sequence per minute on CPU – yielding an estimated 300× speed-up in filtering capability." (Exact numbers to be measured.)

We will also evaluate on the **test set** (the 343 test videos of YouTube-VIS, if available in the subset) to report final generalization performance, using the model that performed best on validation.

7. Tools, Libraries, and Hardware Resources

To implement this pipeline, we will utilize the following tools and libraries:

- **Python 3.x**: The primary language for glueing the pipeline.
- **OpenCV** (cv2): For video decoding and frame extraction, and possibly for some image augmentations and processing (cropping, resizing) in preprocessing. OpenCV provides convenient methods to read videos frame-by-frame and to write images to disk.
- **COLMAP**: The COLMAP CLI (installed from their GitHub or via package) is essential for ground truth SfM computations. We will install COLMAP on the system (ensuring it's built with CUDA support for using the GPU in feature extraction). We might also use **PyCOLMAP** (Python bindings) for easier result parsing, but the CLI plus manual parsing is sufficient. COLMAP's robust SfM results are the cornerstone for generating our training labels.
- **PyTorch**: The deep learning framework used to build and train PoolNet. We will leverage PyTorch for model definition (nn.Module), training loop, and GPU acceleration. Version 1.12+ or 2.0 can be used, with CUDA support enabled for the RTX 3090.

- **Hugging Face Transformers**: Provides the `ViTModel` implementation and pre-trained weights for ViT. We will use `transformers` library to load `google/vit-base-patch16-224-in21k` with `ViTModel.from_pretrained(...)`. This gives us a ready-made ViT backbone.
- **Torchvision/Albumentations**: Libraries for data augmentation routines. Torchvision has basic transforms and image reading utilities; Albumentations is another library with many handy augmentation functions that work with numpy arrays (compatible with OpenCV images). Either can be used to implement the described augmentations (color jitter, flips, crops, etc.).
- **NumPy and Pandas**: Useful for data manipulation, computing metrics from COLMAP output (e.g., calculating Vr, Vt from arrays of poses), and storing metadata (we might use Pandas DataFrame to organize the chunk metadata and ground truth table).
- **Matplotlib/Seaborn** (optional): For plotting results, such as loss curves during training or scatter plots of predicted vs actual values for analysis. While not essential to the pipeline, they are useful for the evaluation phase.

Hardware – Single RTX 3090 GPU Setup: We assume a machine with one NVIDIA RTX 3090 (24 GB VRAM) and a multi-core CPU (e.g., 12-core) for this project. The GPU will accelerate neural network training and inference. A single 3090 is sufficient given the model size: - ViT-base and LSTM easily fit in memory with batch size of a few sequences. 24 GB VRAM allows storing the model (~0.35 GB for ViT weights plus some overhead) and intermediate activations for perhaps up to 4 sequences of 20 frames at 224×224 at once. We will adjust batch size based on empirical memory usage. - During COLMAP processing, the GPU (if used for feature extraction) will also utilize the RTX 3090. COLMAP's feature extraction on GPU can significantly speed up that step for thousands of images. The bundle adjustment and other steps use CPU; having a decent CPU and parallel threads helps. We can configure COLMAP's `--SiftExtraction.num_threads` and other parallel options to fully utilize the CPU. - Disk space: The 11 GiB of video will expand when frames are extracted (each frame ~ a few hundred KB to a few MB depending on resolution and compression). We should have on the order of 50–100 GB free to store all frames and COLMAP results. After extracting frames, if space is an issue, we can delete the original videos. We will also generate COLMAP databases and models for each chunk; each such run might be tens of MB (SIFT features, etc.). It's wise to delete or compress those after extracting the needed metrics to save space (for example, we might not keep the full point cloud, only metrics). - **Software Environment:** Install PyTorch with CUDA 11 (for RTX 3090), install `transformers` library, OpenCV, etc. Also, install COLMAP (either via package or build from source). Ensure the environment is set up so that we can call `colmap` from the command line (or use pycolmap). We might create a conda environment listing these dependencies for reproducibility.

Using these tools and setup, we will implement the pipeline step by step: data preparation, ground truth computation, model training, and evaluation. By the end, we aim to have a trained PoolNet model that can rapidly predict SfM outcomes on new image sequences, thereby acting as a filter to save costly COLMAP computations on low-quality data [8] [7]. This will be the first component of a larger 3D data processing pipeline, demonstrating effective collaboration between classical 3D vision tools and modern deep learning.

Sources:

- The description of PoolNet's goals and design is informed by the project proposal [9], which outlines using COLMAP for ground truth and a ViT+LSTM architecture.
- COLMAP usage and capabilities are referenced from its documentation and the proposal.
- The YouTube-VIS dataset characteristics are noted from the TensorFlow Datasets catalog [1].

- The model architecture choices (ViT, LSTM) are justified by their ability to handle arbitrary image sizes and temporal data, as mentioned in the proposal.

---

1 youtube_vis | TensorFlow Datasets
https://www.tensorflow.org/datasets/catalog/youtube_vis

2 Command-line Interface — COLMAP 3.12.0.dev0 documentation
https://colmap.github.io/cli.html

3 4 5 6 7 8 9 Project_Proposal.pdf
file://file-3q4RetTVjGGJc4Nmtkp3gc