TARP: Tensorflow-based Activity Recognition Platform

Eric Hofesmann, Madan Ravi Ganesh, and Jason J. Corso

University of Michigan

Abstract. **Keywords:**

1 Introduction

2 Overview of TARP

TARP is comprised of two main components, 1) the data input block, and 2) the execution block, as shown in Fig. 1. The pipeline contained within the data input block can be divided into three simple stages,

1. Read video data from disk
2. Extract the desired number of clips from a given video
3. Preprocess the frames of clips using a selected model's preprocessing strategy.

The execution block houses all of the code required to setup, train, test as well as log the outputs of a chosen model. This includes defining the layers that comprise the model, training the model up to a predetermined number of epochs, saving parameter values of a model are regular intervals and finally testing the performance of the trained model over a variety of recognition metrics. The following sections provide an in depth discussion of the setup and structure of various components of the platform.

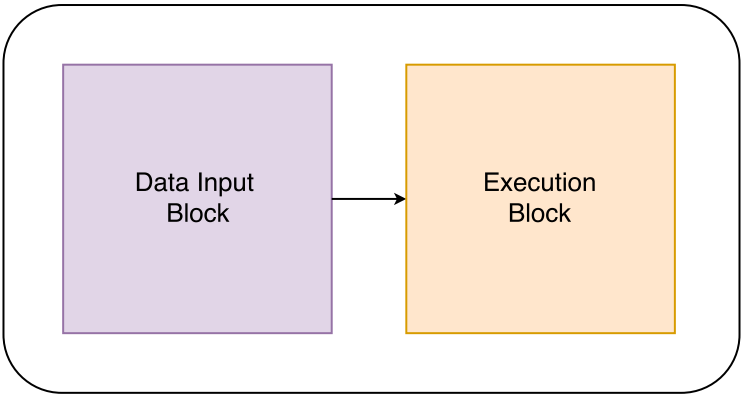
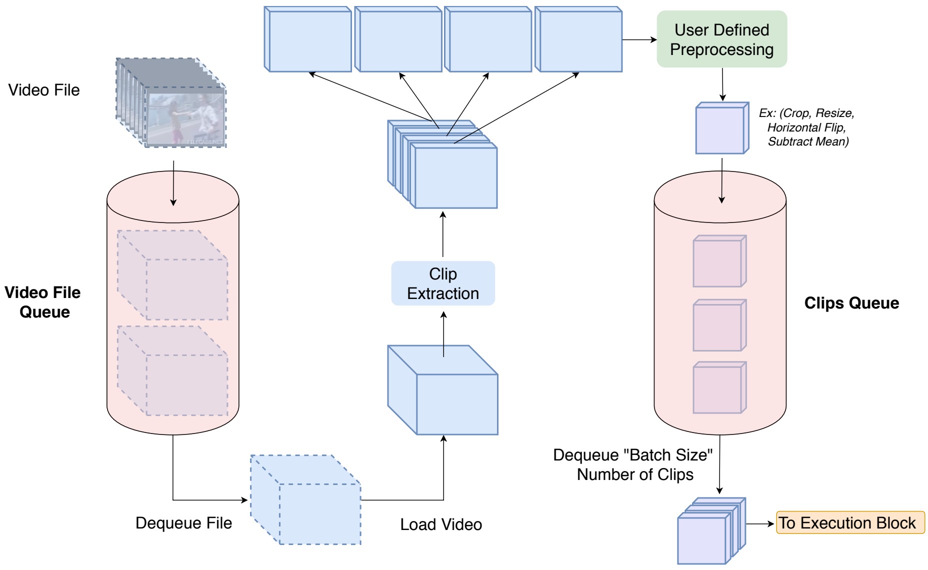


Fig. 1: Illustration of the two main components of TARP

3 Input Data Block

Included in the input pipeline are all of the steps required to pass a video into a model in the proper format. The three stages of the input pipeline, including reading video data, extracting clips, and preprocessing clips, are detailed below.



Video File
Video File Queue
Dequeue File
Load Video
Clip Extraction
User Defined Preprocessing
Ex: (Crop, Resize, Horizontal Flip, Subtract Mean)
Clips Queue
Dequeue "Batch Size" Number of Clips
To Execution Block

Fig. 2: This figure depicts the data flow from loading video files to passing a processed batch of clips into the specified model. The input pipeline revolves around two separate queues, one queue for storing file names of all tfrecords in the specified dataset and another queue for storing processed clips waiting to be passed to the model. During runtime, the video file queue gets filled with all tfrecords filenames, which then get dequeued one by one. The corresponding video to that file gets loaded into the system and it gets broken down into clips as described in Sec. 3.2. Each clip then gets individually preprocessed according to a set of user defined steps as described in Sec. 4.2. The processed queues then get loaded into the `clips_queue` so that exactly one `batch_size` of clips is available to be loaded into the model.

3.1 Read video data

In TARP, video data is stored in the form of tfrecords files. These files can be generated using the provided code given that the original video datasets have been acquired. The contents of these files include the video height, width, channels, number of frames, BGR data, name of the video file, and the label of the action class related to the video. Video data is stored in BGR format due to the use of OpenCV for converting the given video files to python arrays. However, the first step of the input pipeline after reading in the tfrecords data is to convert the video to RGB.

Reading tfrecords files is accomplished through the use of a `tfrecord_file_queue` which stores the names of all tfrecords in the requested dataset split. Enqueue and dequeue operations to this queue are built into the tensorflow graph to allow for efficient parallel data loading. After individuals are able to be loaded from this queue, these videos must now be broken down into clips and preprocessed.

To allow for the loading of multiple clips for each loaded video, another FIFO queue is added to the pipeline to store individual clips.

Note: When loading the dataset HMDB51, videos are stored in a 30 frames-per-second (fps) format while most models require 25 fps. A function, _reduce_fps is used to remove every sixth frame from HMDB51 videos in order to reduce the fps.

3.2 Extract clips

Certain models, for example C3D **??**, require their inputs to be in the form of multiple clips extracted from a single video. Fig. 3 details the various arguments that are available to break down a video into clips. These methods shown are able to be used in conjunction with one another to give the user the flexibility needed to extract clips with different shapes and sizes.

By default, the clip extraction algorithm is set to process and entire video at a time and only extract clips if the required arguments have been specified. In addition to the options shown in Fig. 3, TARP also provides options for random selection of clips.

3.3 Preprocess clips

Activity recognition employ a number of various preprocessing methods on a frame or clip-based level. These can include frame-wise cropping, flipping, and resizing, or clip-wise cropping, flipping, temporal offsets, resampling, and looping. In order to incorporate these methods as desired per model, we create one or multiple preprocessing files for each model. The desired preprocessing file for a model can be defined within the model and then called from the inputs call

TARP includes implementations of all of these options which can easily be added to a model's preprocessing file.

Note: Due to the nature of tensorflow enqueue operations, errors that occur within an enqueue operation do not properly throw an error. The enqueue is simply canceled and an error will occur when the dequeue attempts to access an empty queue. This effect occurs within the `clips_q` if an error occurs in the user defined preprocessing submodule. In order to allow proper debugging of the preprocessing submodule, the argument `preprocDebugging` is available to bypass the `clips_q` and see any error tracebacks. Since the queue used to store clips is removed, the requirement is set that `batch_size` must equal `num_clips` while debugging.

4 Execution Block

Training and testing form the two main tenants of the execution block. Fig. 4 illustrates this partition along with the largest submodules present within each part. Details regarding the flow of data and processes between various submodules and the entire execution block are provided below.

4.1 Training: Algorithmic flow of processes

Training, within the context of the execution block, follows the process flow outlined in Alg. **??**. The beginning of every training experiment is aligned with a chosen model. Here, basic model parameters such as input/output data dimensions, batch normalization, dropout rate, and etc. are passed into the *Model submodule* in order to select and setup a desired model according to well defined prerequisites.

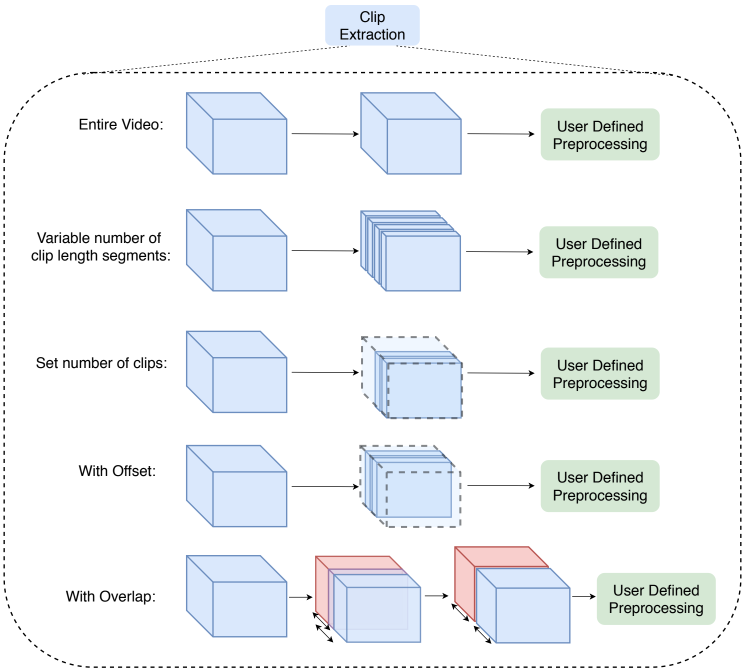

Clip Extraction
Entire Video:
User Defined Preprocessing
Variable number of clip length segments:
User Defined Preprocessing
Set number of clips:
User Defined Preprocessing
With Offset:
User Defined Preprocessing
With Overlap:
User Defined Preprocessing

Fig. 3: The clip extraction function allows for videos to be broken into clips according to defined specifications. Whenever a video has been completely processed and used for training or testing, a new video is loaded from the queue. The video is then passed through our clip extraction algorithm which enqueues the clips individually into a secondary queue. Clips are then dequeued from the clips queue according to the number of GPUs being used and the required clip batch size.

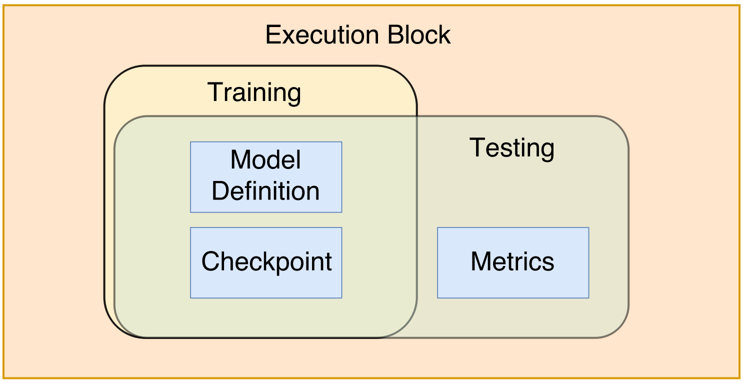


Fig. 4: Training and Testing functions are the two major phases within the execution block. Model definitions and checkpoint-based functions are part of both training and testing functions while metrics are calculated after models are tested.

Once the model has been set up, the next step is to ensure that the parameter weights for the chosen model are retrieved. If the model has been pre-trained using TARP and the user desires to further fine-tune the model, then the model parameter's weights from the latest checkpoint will be restored. By default, the standard pre-trained weights, such as Kinetics [] for I3D [], Sports1M [] for C3D [], ImageNet [] for ResNet50 [] and TSN [] are loaded into the model. Alternatively, a random initialization option, for users to train from scratch is also provided. Note: To ensure that there is a backup when a specified checkpoint is unavailable, the random intialization option is used, with the system providing a warning to the user about the setup used in training.

After the preparation of the model is complete, data tensors are retrieved from the **Data Input Block**. The main data retuned from **Data Input Block** are the video frames, their corresponding labels and video names. With the availability of the model and data, the next stage is creating a copy of the model on the selected number of GPUs within a compute node. TARP only supports the extension of a model within a compute node of a cluster. Currently, it cannot be distrbute a model across mutiple nodes or split a model across multiple GPUs within a node.

Within each GPU, the inference function of *Model submodule* is used to generate a copy of the layer definitions. Here, the variable names are re-used to ensure that the same copy of weights are associated throughout all the GPUs. From each of the model copies, we retrieve a desired layer's output. Given that the most common approach to training a network is through the use of logits returned from the final layer, we use the variable logits associated with the last layer to obtain a loss value. Gradient computation based on the loss calculated is also performed. Finally, the loss and gradients across all the model copies are accumulated into the loss_array and gradients_array respectively. The contribution of each model copy is weights equally and the final gradient used to update the weights of the model is obtained by taking the average of value stored in the gradients_array. The is the final stage in the definition of the tensorflow graph required for training.

```
procedure Training
    Model_definition.setup(model_params)
    Checkpoint.load_param_weights(default or specific file)
    Data, Labels = DIB.load_data(expt_params)

    for each GPU within given node do
        Model_definition.inference(Data, model_params)
        tower_loss = Model_definition.loss(Labels, returned_logits)
        tower_gradients = gradients(tower_loss)
        loss_array.store(tower_loss)
        gradients_array.store(tower_gradients)
    end for

    grad = average_gradients(gradients_array)
    train_op = apply_gradients(grad)

    while no_videos_loaded < (total_epochs × videos_in_dataset) do
        Checkpoint.save() @ regular intervals
        sess.run(train_op)
        update(no_videos_loaded)
    end while
end procedure
```

After the tensorflow graph for training has been defined, the next step is to execute the graph. Usually, models are trained for a typical number of epochs, iterations over the entire dataset. Within each epoch, based on a set frequency of iterations, we save the model parameter's weights using the save function within *Checkpoint submodule*. Additionally, we use a tensorboard logging system to record the loss, training accuracy, and training time along with any specified model parameters at every iteration. The main operation that gets executed within each iteration is the train_op. It is important to note that during the setup of the experiment, TARP offers adaptive learning rate control which steps down the learning rate by a factor of 0.1 when the loss plateaus.

4.2 Model Description Submodule

The class definition of a model within TARP can be divided into four main components as shown in Fig. ??. The first step to incorporating a model in TARP is to define the network architecture within an inference function. The layers used must strictly be called from within the set of definitions provided within the layer_utils file. This ensures that layer definitions are not specific to any network and become available to all models defined within TARP. Further, it is essential that the inference function returns the outcome of any and all desired layers from within the defined network. Any accompanying functions to help quickly and/or recursively define a network can be added outisde of the inference function.

Once the network has been defined, the next step is to setup the network's desired preprocessing formulation. Concrete definitions of each preprocessing step can be included in a separate file. We provide a standard list of preprocessing functions within the preprocessing_utils file that can be used to construct the entire preprocessing pipeline. The expected output from a preprocessing function is the preprocessed input data. Multiple preprocessing pipelines can be included within the same model with the use of the simple *if-else* construct to quickly expand a model's abilities. Within the model definition file, it is sufficient to import and call a preprocessing method defined in the adjacent file.

In TARP, we attach the definition of any loss to a model to avoid cluttering and increasing the complexity of the main training file. Thus, a basic loss function needs to be defined within a models class, with the expected return being logits. However, the basic loss function can altenratively be used to call different loss functions by using the internally defined `loss_type` keyword.

The final and optional component of any model definition within TARP is the definition of load_default_weights function. This function is defined to ensure that a predetermined set of weights can be loaded onto the network defined. The explicit rules required to define the layer names and ensure their respective weights can be assigned are provided in Section 4.3. The final values returned by this function are the parameter weights loaded from an external file.

In order to quickly define and integrate any model into TARP's pipeline, we provide a template file named models_abstract.py which provides the outline for the user to quickly fill out the inference, preprocessing, loss and load weights functions. A separate file named models_preprocessing_template.py provides a template for defining the preprocessing pipeline.

Note: 1) Any model defined must be placed within the models folder under a directory with the same name as that of the model, 2) No explicit model class attributes can be added to a specific model. Instead, they must be added to the model_abstract.py file and made available to any and all model definitions within TARP. The models that come standard with TARP include the state-of-the-art activity recognition architectures I3D [], C3D [], ResNet50 [], and TSN [].

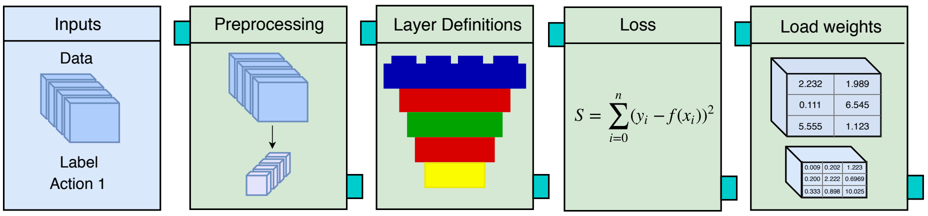


Fig. 5

4.3 Checkpoint Submodule

4.4 Testing: Flowchart of processes

Testing begins similarly to training in that a specified model gets defined and initialized according to given parameters. The model parameters during testing are identical to those used to instantiate a model during training. Other input arguments that do exist when testing that are not used during training include `metrics_method`, `return_layer`, and `avg_clips`.

After a model object has been instantiated, the model weights are loaded. The option exists to either load the weights from the default weight file specified within the **Model Submodule** or to initialize the model to random weights if that is desired. Most commly, however, the **Checkpoint Submodule** is used to load the checkpoints of a model that has been trained from scratch or fine-tuned.

Once the model weights are accessible, the dataset will be loaded into the system following the process described in the **Data Input Block** in Sec. **??**. Videos are not shuffled when being loaded for testing so that all testing instances are directly comparable. The shape of this input data is set to allow for any number of different videos or multiple clips of a single video to be loaded for testing. More specifically, the `batch_size` of input data can be used to load multiple clips of a single video in order to determine a consensus across these clips which will act as the prediction for the entire video as explained in C3D []. In addition, this `batch_size` parameter can also load multiple videos simultaneously if videos are broken into a number of clips which is smaller than the batch. No models currently use this multiple video functionality for testing, videos are loaded one at a time for testing and only batched for training, but it is available if needed in the future.

Before the data can be loaded into the model, the model must first be copied onto the GPU. While only a single GPU is used for testing, a GPU list can still be given to TARP to determine which GPU the testing instance will be assigned to. If the list contains more than one GPU device id, the first element in that list will be used. The process of copying the model to the GPU is based around the **Model_definition.**inference call which will return the output of the layer specified in the `return_layer` argument. By default, the layer that will be returned by **Model_definition.**inference are of the shape [`batch_size`, `sequence_length`, `output_dimensions`] and then converted to action class predictions using the softmax function. This shape is only required when the `logits` layer is returned from the inference call and the application of softmax is dependent on the `useSoftmax` argument. With the model output acquired, the final stepis of preparation for testing is to set up the output directory structure, the **Metrics Submodule** detailed in Sec. 4.5, and to load weights into the model as described in the **Checkpoints Submodule** in Sec. 4.3.

The testing block iterates through the `num_vids` specified which generally includes an entire testing split of a dataset. Within this block, the tensorflow session is run to extract the model's `output_predictions`, ground truth `labels`, and all video `names` in this batch. If the `avg_clips` argument has been specified,

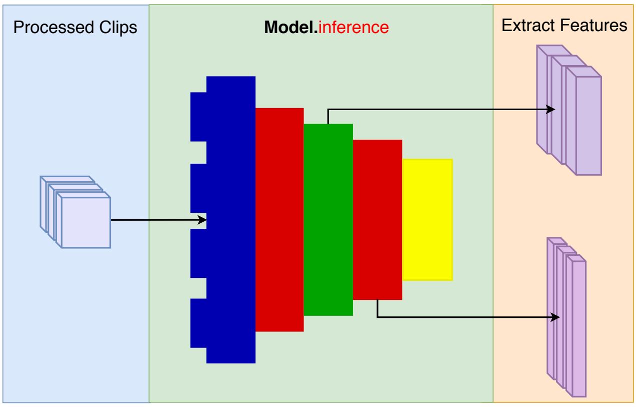


Fig. 6

then the output_predictions are averaged over their first dimension which is batch_size or in this case, the number of clips loaded for this testing iteration. A tracker counts the number of videos that have been loaded using the video names output. The output_predictions and ground truth labels are then fed into the **Metrics Submodule** for logging and storing. Finally, the thread coordinator for the input queues is stopped and the all predictions from the **Metrics Submodule** are saved as a numpy format.

4.5 Metrics Submodule

TARP offers a **Metrics Submodule** for testing and logging the activity recognition performance of a model as well as extracting features from a model. This **Metrics Submodule** is incorporated in the testing pipeline as a class which gets instantiated in test.py. During testing, the output prediction, correct label, and name of each video which is loaded and passed through the model gets stored in this object during every iteration. These performance metrics and other temporary data gets stored in an hdf5 file format and then deleted at the end of each testing session. Hdf5 files are used for storing large amounts of temporary data due to their quick reading and writing capabilities. Just before getting deleted, all relevent testing outputs are saved as numpy files. Additionally, during testing, the chosen metric and any other specified varaible will be logged automatically using tensorboard.

Available performance metrics include average pooling, last frame prediction, and training a linear SVM classifier. Average pooling is a common technique which uses the softmax predictions of a model's outputs and averages them across all frames of an input video. From these predictions, the maximum value is used as the overall class prediction for that video. The outputs of models can vary in shape so we standardize them to be in the form of [batch_size, sequence_length, number_of_classes]. The batch_size parameter slightly changes meaning depending on the model being used. For example for C3D, multiple clips can be loaded for a single video, the average can then be taken across all of these clips while the temporal dimensionality gets reduced from 16 frames to 1, removing the need for any average pooling or last frame prediction. On the other hand, TSN would take multiple videos and refer to each as a single clip, thus the batch of different videos would each be logged separately. TARP is robust enough to handle both such cases. Last frame classification is referenced originally in the ConvNet + LSTM model []. This technique calculates the argmax of solely the last frame of the input clip. In addition to these metrics which directly classify the output of a model, TARP also provides the option of training a linear SVM classifier based off of features extracted from any user specified layer within the model. A linear SVM was used to classify video in the UCF101 dataset in C3D []

Features can be extracted for a multitude of reasons other than just to train a linear SVM. This quality allows TARP to accomplish more than just activity recognition. Any of the state-of-the-art models that have been implemented, or even any custom model, can be used for feature extraction. With any dataset that has been converted to the proper tfrecords format, the correct input arguments allow the user to extract features from any layer in any available model. Due to the large size that features can become over modern datasets with thousands and even millions of videos, it is crucial to be able to read and write them to memory quickly which is why we use an hdf5 file format for writing this data.

5 Benchmarks