TARP: Tensorflow-based Activity Recognition Platform

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

University of Michigan

Abstract. Action recognition is a widely known and popular task that is essential to build up knowledge for video understanding. The absence of an easy-to-use platform containing state-of-the-art (SOTA) models presents an issue for the community. Given that individual research code is not written with an end user in mind and in certain cases code is not released, even for published articles, the importance of a common unified platform capable of delivering results while removing the burden of developing an entire system is further underlined. To try and overcome these issues, we develop a tensorflow-based unified platform to abstract away unnecessary overheads in terms of an end-to-end pipeline setup in order to allow the end user to quickly and easily prototype action recognition models. With the use of a consistent coding style across different models and seamless data flow between various submodules, the platform lends itself to the quick generation of results on a wide range of SOTA methods across a variety of datasets. All of these features are made possible through the use of a fully pre-defined training and testing pipeline built on top of a small but powerful set of modular functions that handle asynchronous data loading, model initializations, metric calculations, saving and loading of checkpoints, and logging of results. The platform is geared towards easily creating models, with the minimum requirement being the definition of a network architecture and preprocessing steps from a large custom selection of layers and preprocessing functions. TARP currently houses four SOTA activity recognition models which include, I3D, C3D, ResNet50+LSTM and TSN. The recognition performed achieved by these models are, $XX\%$ for ResNet50+LSTM and $YY\%$ for C3D on XXXXX while I3D and TSN achieve $ZZ\%$ and $QQ\%$ on YYYYY respectively.

Keywords: tensorflow, activity recognition, framework, state-of-the-art models, C3D, I3D, Temporal Segment Networks, ConvNet+LSTM

1 Introduction

Problem Space: The wider research comminity has a tendency to follow their own internal guildines and coding practices which vary form place to place adn even between professors. Given this, the vast expanse of sota models for any given task that get published tend to 1. look different 2. not all components required for experiment get published Hence, these two factors in combination

make it difficult to reproduce results from papers related to these models. We are trying to solve this problem in the context of action recognition in computer vision.

How others have attempted it, Why they're bad: There has been a recent trend in the computer vision researh community for open sourcing code. The problem being that code is not written in a consistent way. When a new sota model gets released with no way to reproduce results, there will be researchers that take on the challenge of replicating that work in order to be able to compare against it. This takes away valuable time to analyze and improve upon these models and instead focuses on the task of replication. The wider research code has their own style of writing using a variety of languages. Wherein each of these languages there can exist subtle differences, for example dropout meaning how much to keep vs how much to throw out btwn caffe and tf. In terms of a task like feature extraction, each language will have a model zoo to access various models, however this does not exist for the task of activity reocgntiion.

How we do it, why it's better: To avoid the complication of parsing between languages and for there to be a miscommunication, we use a single language and provide multiple sota activity recognition models. The existence of sota models in it removes the necessity of replication and the researcher can focus their time on improving upon existing concepts and models. For this purpose, we provide TARP which is a platform that allows the user to solely focus on the creation or fine-tuning of models by abstracting away uncessecary periferal details about input, computing metrics, and output.

Our contributions: general feature extraction from sota models quickly generate results across a variety of models and datasets. multi-gpu support to train and finetune models an input pipeline that condenses the loading and processing of videos into a set of flexible and easy-to-use options for the user. The ability to convert models trained in another language to work with this framework using only a common NumPy structure. layers and preprocessing utils

2 Overview of TARP

TARP is comprised of two main components, 1) the **Input Data Block**, and 2) the **Execution Block**, as shown in Fig. 1. The pipeline contained within the **Input Data 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.

On the other hand, 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 that make up the **Input Data** and **Execution Blocks**.

3 Input Data Block

Included in the **Input Data Block** are all of the steps required to load a video and process it into the proper format. Fig. 2 shows the structure and flow of data as it passes through the **Input Data Block**. The three stages of the **Input Data Block**, reading video data, extracting clips, and preprocessing clips, are detailed below.

3.1 Read video data

In TARP, video data is stored in the form of tfrecords files to allow the entire **Input Data Block** to be constructed into the tensorflow graph for efficient parallel data loading. These files can be generated using provided scripts and examples, assuming 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 original video files to python arrays. 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 given dataset directory. After a file name has been extracted form this queue, the corresponding video will then be broken down into clips and preprocessed. To allow for the loading of multiple clips from each video, a `clips_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

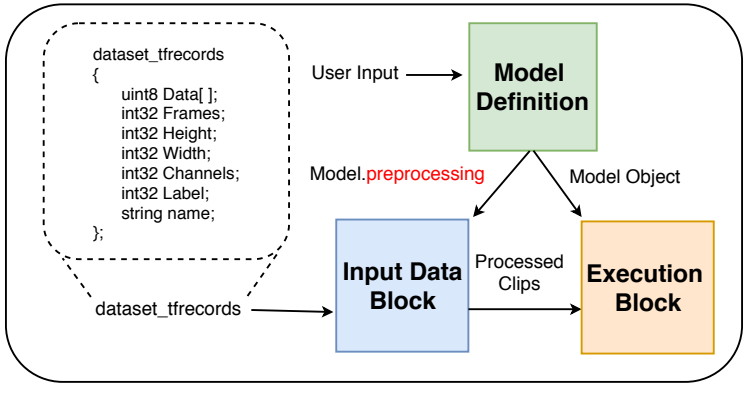

Fig. 1: Illustration of the two main components of TARP

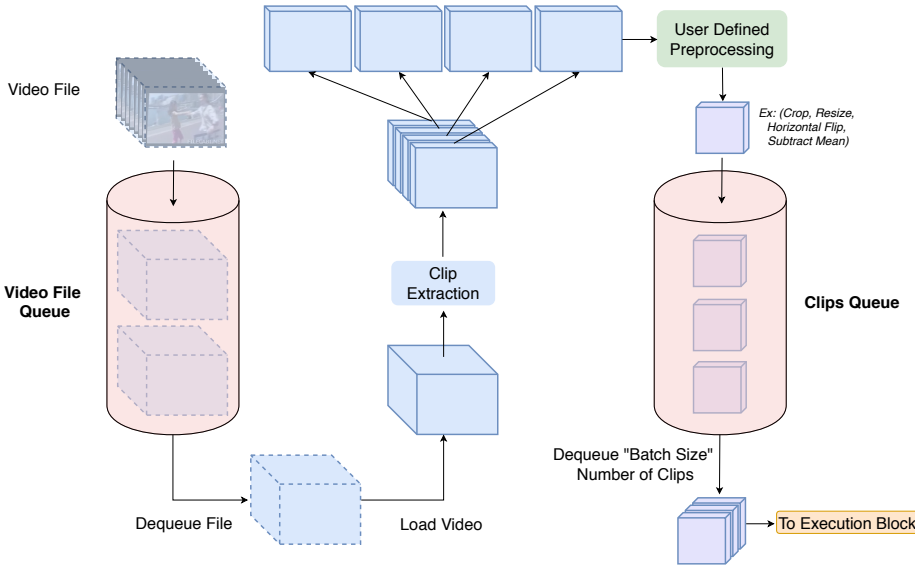

Fig. 2: This figure depicts the data flow from loading video files to passing a processed batch of clips into the specified model. The input pipeline is structured around two separate queues, one queue for storing file names of all tfrecords in the specified dataset and another queue for storing processed clips waiting to be passed to the model. During runtime, the `video_file_queue` gets filled with all available tfrecords filenames in a given dataset, which then get dequeued one by one. The video corresponding to a dequeued file name gets loaded into the system and broken down into clips as described in Sec. 3.2 and Fig. 3. 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.

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 dimensions. All of this can be done through the use of only a handful of arguments including `clip_length`, `num_clips`, `clip_offset`, and `clip_stride` 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 models employ a number of various preprocessing methods on a frame or clip-based level often used for data augmentation. These can include frame-wise cropping, flipping, and resizing, or clip-wise cropping, flipping, temporal offsets, resampling, looping, and more. TARP includes implementations of all of these options which can easily be added to a model's preprocessing file. The desired preprocessing file for a model can be defined within the **Model Submodule** and will be used automatically in the **Input Data Block** after a video has been broken down into clips. Preprocessed clips can significantly vary in shape when compared to the input video. Any change in frame height, width, or number of frames is allowed as long as clips are not broken down further. The `clips_queue` is designed to be compatible with any dimensionality of clips, as long as all clips that are being stored have the same dimensionality. More specifically, processed clips must be of the shape [`input_dimensions`, `size`, `size`, `sequence_length`] for the entirety of a training or testing session.

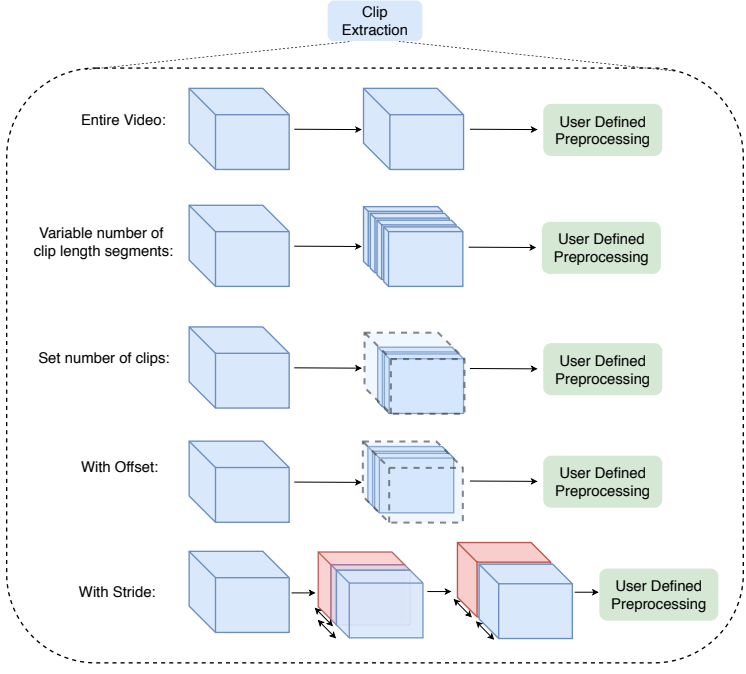

Fig. 3: This figure shows the varying ways that a video can be broken into clips. By default the entire video is passed directly on to the user defined preprocessing without any modifications. A variable number of clips can be extracted from a video if only the `clip_length` parameter is given. Due to this variablity, the implementation of the `clips_queue` is critical

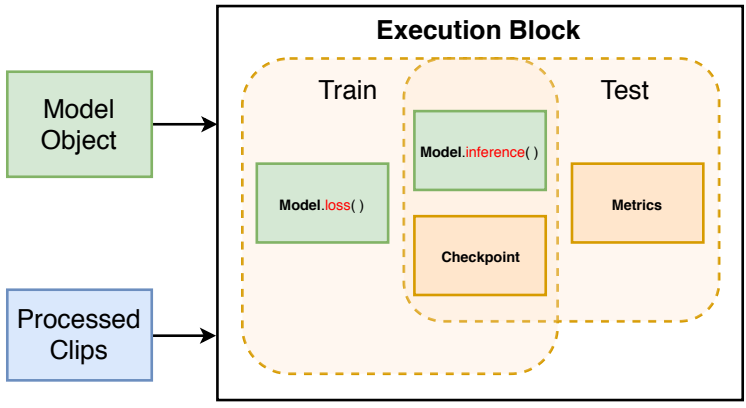

Fig. 4: Training and Testing functions are the two major phases within the execution block. Model definitions and checkpoint-based functions are part of both training and testing functions while metrics are calculated after models are tested.

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

The execution block form the largest component of the entire platform. Its code can be broken down into parts that are used during two alternate phases, training and testing. Fig. 4 illustrates this concept and highlights the submodules used within each phase. An algorithmic overview of each phase, followed by a detailed description of the various submodules used within them are provided in the following sections.

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 phase is associated with the selection and set up of a model. Basic model parameters such as input/output data dimensions, batch normalization, dropout rate, and etc. are passed into the **Model submodule** for this purpose.

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 are loaded into the model. Alternatively, a random initialization option, for users to train a model from scratch is also provided. As a backup when a specified checkpoint is unavailable, the system defaults to the random intialization option, with an additional warning provided to the user about the setup being used.

After the preparation of the model is complete, data tensors are retrieved from the **Data Input Block** and interfaced to the model. The main data re-tuned from **Data Input Block** are the video frames, their corresponding labels and video names. With the availability of the model and data, the next step is to create a copy of the model on each of the selected number of GPUs within a compute 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. 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.

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. Finally, the loss and gradients, computed from the loss, across all the model copies are accumulated into the loss_array and gradients_array respectively. The contribution of each model copy is weighted 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 **Checkpoint submodule**.save(). The main operation that gets executed within each iteration is the training operation that links the calculation of losses, gradients and their application. It is important to note that during the setup of the experiment, TARP offers adaptive learning rate control which steps down the learning rate when the loss plateaus. The repeated application of train_op forms the most crucial and final step in the training process flow.

4.2 Model Description Submodule

The class definition of a model within TARP can be divided into four main components as shown in Fig. **??**. Preprocessing is the first module in the **Model submodule**that gets used within **Data Input Block**. In keeping with the modular design paradigm, we use individual functions to define each variant of the preprocessing pipeline. By keeping each variant within its own file, marked with the suffix "_preprocessing.py", we increase their reusability and ensure debugging them is easy. Within the class definition of a model, multiple preprocessing pipelines can be included using a simple *if-else* construct. The expected final product from any preprocessing function is the complete preprocessed input data that needs to be passed into a network's input layer.

The definition of a model's input layer and the subsequent layers that make up the model's network architecture should be present within the 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

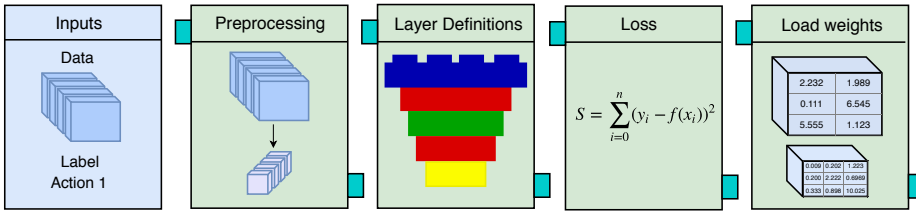

Fig. 5

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.

As a minor deviation from standard practices, we attach the definition of a loss to the model definition instead of the main training file. We do this to avoid cluttering and increasing the complexity of the main training file. Thus, a prime requirement within any model definition is the implementation of a loss function, with the expected return from this function being the final loss value. To add further flexibility in using multiple losses with a single model, the basic loss function can alternatively be overloaded to call different losses using the internally defined `loss_type` keyword.

The final and optional component of any model definition within TARP is the definition of the load_default_weights function. This function is defined to ensure that a predetermined set of weights can be loaded into a network's layers. 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.

Given that the focus of TARP is to help the end-user quickly define and integrate a model to the framework, we provide a template file named models_template.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 an outline of the necessary function to defining the preprocessing pipeline. Once a model has been completely defined, it is automatically imported into the framework through a recursive call strategy. To quickly track and log a model class variable an inbuilt dictionary named `track_variables` is available. Adding any pre-defined variable name and graph title allows the logging of the desired variable on tensorboard.

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) The actual model file must be named with a suffix "_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 [].

4.3 Checkpoint Submodule

The checkpoint submodule contains utility functions that handle all the essential functions w.r.t. checkpoints. Checkpoints themselves are stored in a custom format as shown in Fig. **??**. This format readily lends itself to a recursive access strategy while offering easy conversion capabilities from its native numpy format to ".ckpt",".caffemodel" and other formats. Among the variety of operations that can be performed on and using checkpoints, there are two broad categories, Load/Save and Tensor handling.

Load/Save At the beginning and end of every training processes, the ability to save and recover the current/last known state of the model parameters is paramount. In order to save the current state of a model's parameters, a new checkpoints folder is generated in the results directory. The folder structure generated to save the checkpoints is, <dataset>/<preprocessing_method_name>/<experiment_name>/<metric_name>/checkpoints. Within this folder three distinct files are generated.

- Checkpoint file: A replica of the checkpoint file obtained from native tensorflow code is generated by the save_checkpoint function. The current global step is saved within this file, named "checkpoint".
- Data file: A separate data file is used to store the current learning rate of the experiment. The file is named using the convention checkpoint-<global step number>.dat.
- Numpy weights file: Variables declared and available within the global scope of the experiment are retrieved, name and value, and stored within the numpy weights file. It follows a similar naming convention to the Data file, checkpoint-<global step number>.npy.

When a checkpoint is being loaded, the utility searches for each of the three files defined above to return a distinct variable holding the contents of the files. If any of the files are missing, the checkpoints fails to get loaded and it is a clear indication of a corrupted checkpoint.

Tensor Handling The tensor handler is a recursively defined function which takes names of tensors from the numpy file and loops over assigning the corresponding values associated with those tensor names through the function `tf.assign(tf.graph.get_tensor_by_name(<tensor_name>), numpy value)`. A model is initialized from a pre-existing numpy dictionary through the use of a depth first search and assign policy using the tensor handler. An example of this is shown in Fig. **??**.

An important rule for the definition of any tensor name is that the name must explicitly use the terms "`kernel`" and "`bias`" to denote the `W` (weight) and `b` (bias) matrices respectively. No other name is recognized within the framework.

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, 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.

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

    for one GPU **do**

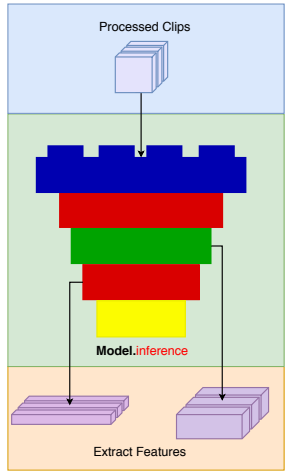

Fig. 6

```
        Model_definition.inference(Data, model_params, return_layer)
        if use_softmax then
            logits = softmax(logits)
        end if
    end for

    Metrics()

    while num_videos_loaded < videos_in_dataset do
        output_predictions = sess.run(logits)
        if average_over_clips then
            mean(output_predictions, 0)
        end if
        Metrics.log_prediction(output_predictions)
        update(num_videos_loaded)
    end while
end procedure
```

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