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.

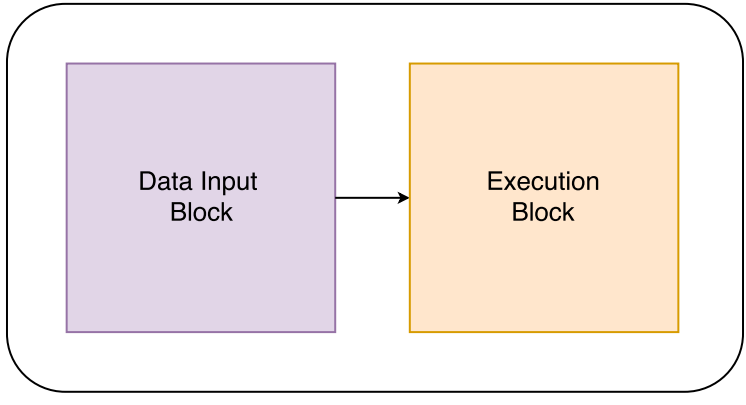

Fig. 1: Illustration of the two main components of TARP

3 Input Pipeline

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 are detailed below.

3.1 Read video data

When a training or testing instance is run on a model, the first step is to setup the tensorflow graph. The first node in this graph is the tensorflow tfrecords file reader. Video data, stored as tfrecords, are read into the system in a FIFO queue.

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. To allow for the loading of multiple clips for each loaded video, another FIFO queue is added to the pipeline to store individual clips. Fig. 2 details the data flow from the tfrecords video data queue, through the clip extraction algorithm, and into the clips queue. 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.

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

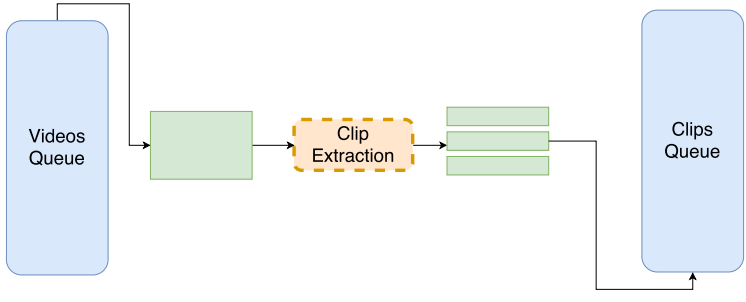

Fig. 2: The input pipeline allows for videos to be broken into clips according to defined specifications. Whenever a video has been completely processed and used for training or testing, a new video is loaded from the queue. The video is then passed through our clip extraction algorithm which enqueues 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.

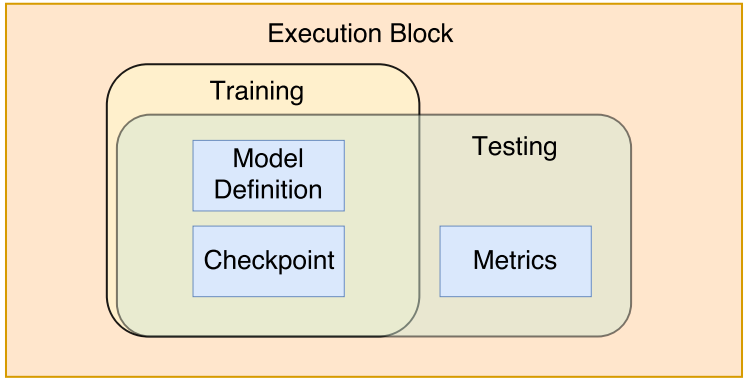

Fig. 3: Training and Testing functions are the two major phases within the execution block. Model definitions and checkpoint-based functions are part of both training and testing functions while metrics are calculated after models are tested.

resizing, or clip-wise 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.

4 Execution Block

Training and testing form the two main tenants of the execution block. Fig. 3 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.

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*. The main operation that gets executed within each iteration is the train_op. It is important to note that during the setup of th experiment, TARP does not offsr adaptive learning rate control. Instead, two alternative arguments are used to specify fixed epoch boundaries at which the learning rate can decay and the scaling factor by which it does.

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.

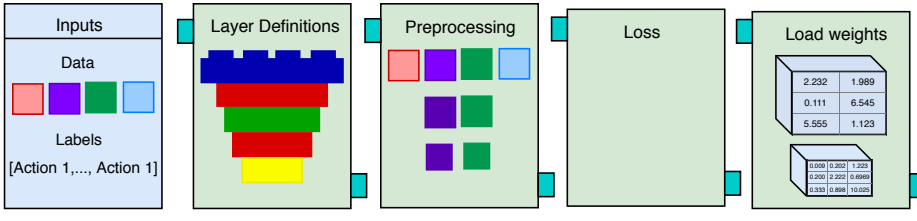

Fig. 4

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. Once a model has been completely defined, it gets 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 where three distinct files are generated.

- Checkpoint file: A replica of the checkpoint file generated by 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>.numpy.

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, it is a clear indication of a corrupted checkpoint.

Tensor Handling The tensor handling function is essentially 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. 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 denot the `W` and `b` matrices respectively.

4.4 Testing: Flowchart of processes

4.5 Metrics Submodule

5 Benchmarks