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.

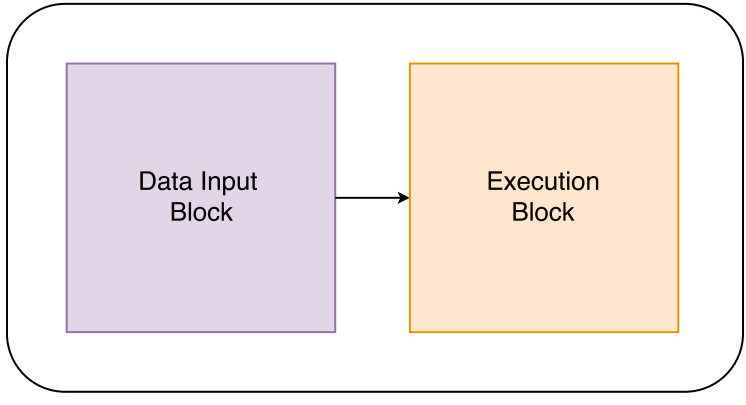

Fig. 1: Illustration of the two main components of TARP

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.

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

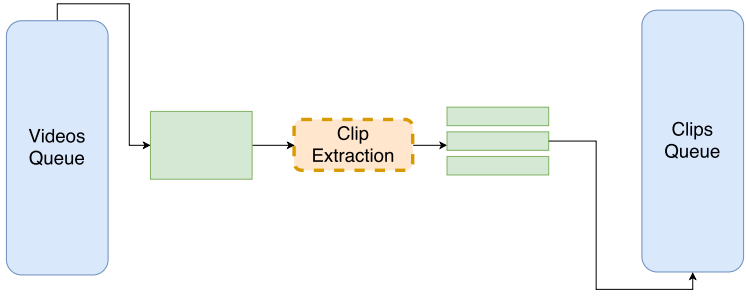

Fig. 2: The input pipeline allows for videos to be broken into clips according to defined specifications. Whenever a video has been completely processed and used for training or testing, a new video is loaded from the queue. The video is then passed through our clip extraction algorithm which enqueues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

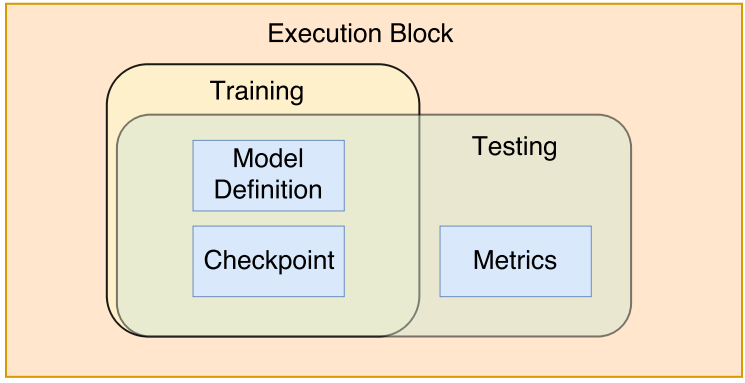

Fig. 3: Training and Testing functions are the two major phases within the execution block. Model definitions and checkpoint-based functions are part of both training and testing functions while metrics are calculated after models are tested.

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.inference(Data, model_params)
        tower_loss = Model.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 ubmodule*. 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

4.3 Checkpoint Submodule

4.4 Testing: Flowchart of processes

4.5 Metrics Submodule

5 Benchmarks