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 offers an approach towards 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 cannot be overstated. 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 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 fully pre-defined training and testing blocks 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 performance 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

The research community is focused on exploring concepts at the frontiers of science and helping solve problems that bog our development down. An important part of research is the analysis of currently proposed theories and further improving upon them. This necessitates the use and re-use of models and theories that are currently state-of-the-art (SOTA). However, given the strong emphasis on

concepts and modelling, coding practices are relegated to a lower priority. Thus, there isn't a universal standard guideline for coding practice and this leads to strong variations from place to place and even between communities within a university. Further, not all the components that are needed to re-create the results published are provided to the research community, which takes away from valuable time spent on research and instead is used to replicate the published results through common knowledge, details published in the paper and a little guesswork. In this paper, we propose to solve this problem in the context of action recognition, in the field of computer vision.

There has been a recent positive trend towards open sourcing code in the computer vision community. However, even when the same language is used to develop software, custom coding practices make it extremely difficult to quickly assimilate and integrate code. When custom coding styles are combined with the vast array of languages available, e.g., tensorflow [1], pytorch [**?**], caffe [**?**] in deep learning, it adds to the complexity since key words in languages tend to vary in meaning and this necessitates an extremely deep understanding of each and every language. Subtle differences, such as, the keyword "`dropout ratio`" can mean the ratio of nodes to drop or keep, can have massive impacts on the final results. When put into the context of feature extraction, each deep learning language has a model zoo to access a number of staples. However, we aren't aware of such a model zoo for a task like activity recognition. Given the popularity of activity recognition, as a precursor to video understanding, and the prevalence of deep learning it is surprising.

We introduce TARP, which is a platform that allows the user to solely focus on the creation or fine-tuning of models by abstracting away unnecessary peripheral details like data input, computing metrics, and extracting features. In order to avoid the complication of parsing code between different language and the complications that can arise from doing so, we use a single language, tensorflow, for our platform. We further supplement it with a suite of SOTA activity recognition models which remove the necessity of replication and the end user can focus their time on implementing deep learning models and conduct research.

The main contributions of the platform are:

- Extract features from SOTA activity recognition models.
- Quickly generate results across a combination of a variety of datasets and ANN models.
- Single node multi-GPU expansion support to train and fine-tune models.
- A custom input block 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 model weights, stored in our custom Numpy dictionary format, to alternative formats used in other deep learning languages..
- A vast array of customized options for neural network layer definitions and preprocessing functions.

Within this report, we use the following formatting techinques,

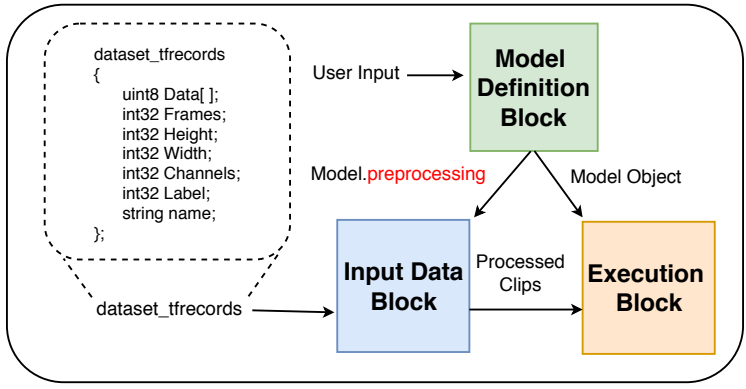

Fig. 1: Illustration of the three main components of TARP. The presented color scheme will be common throught all figures in this paper. Blue, green, and orange identify components of the **Input Data Block**, **Model Definition Block**, and **Execution Block** repectively.

- loss - To denote functions
- file.npy - To denote different file names
- `loss_type` - To denote keywords in the code
- **Model block** - To denote Blocks, submodules and any higher level construct

2 Overview of TARP

As shown in Fig. 1 TARP is comprised of three main components, 1) the **Input Data Block**, 2) **Model Definition Block** and 3) the **Execution Block**. The **Input Data Block** processes datasets in the form of tfrecords present within in a single directory. The inner works of the **Input Data Block** can be divided into two simple stages,

1. Read video data from disk
2. Extract the desired number of clips from a given video and process them.

The **Model Definition Block** contains functions and code that relate to the definition of a model, it's loss, data flow between different layers and the preprocessing pipeline used on input data. 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 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**, **Model Definition**, and **Execution Blocks**.

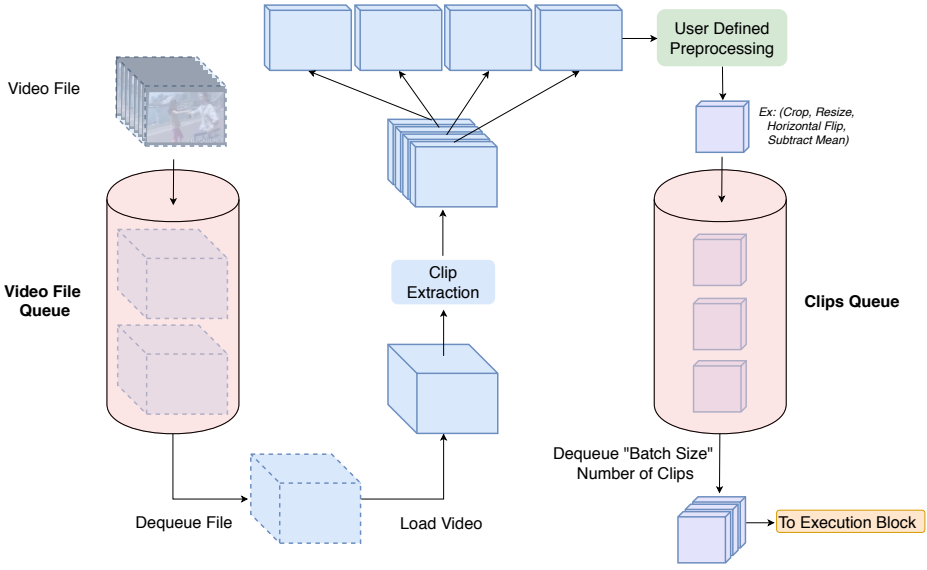

Fig. 2: This figure depicts the data flow from loading video files to passing a processed batch of clips into the specified model. The **Input Data Block** is structured around two separate queues, one queue for storing file names of all tfrecords in the specified dataset and another queue for storing 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. 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 Input Data Block

Included in the **Input Data Block** are all of the steps required to load a video and process it into the platform's desired format. Fig. 2 shows the structure and flow of data as it passes through the **Input Data Block**. The two stages of the **Input Data Block**, reading video data and extracting 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 as can be seen in Fig. 1. Video data is stored in BGR format due to the use of OpenCV [2] for converting the original video files to python arrays. The first step of the **Input Data Block** after reading in 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 **Input Data Block** 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 [3], require their inputs to be in the form of multiple clips extracted from a single video. Fig. 3 details the various methods of breaking down a video into clips. The methods shown can 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 an 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 an option for random clip selection.

Once clips have been extracted, they are preprocessed using a selected model's preprocessing pipeline. The desired preprocessing file for a model can be defined within the **Model Submodule** and it is used automatically in the **Input Data Block**. 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 the number of input and output dimensions of a clip are retained. The `clips_queue` is designed to be compatible with the shape [`input_dimensions`, `size`, `size`, `sequence_length`] for the entirety of a training or testing session.

Note: Due to the nature of tensorflow enqueue operations, errors that occur within an enqueue operation do not throw an appropriate error message. The enqueue is simply canceled and an error will occur when the dequeue attempts to access an empty queue. For example, this effect can be seen 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 stack tracebacks. Since the queue used to store clips is removed, the requirement is set that `batch_size` must equal `num_clips` while debugging.

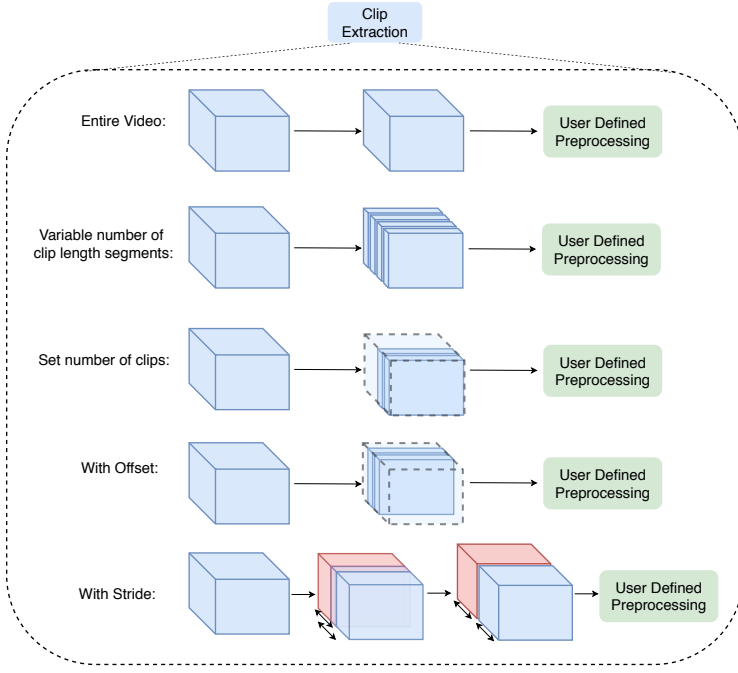

Fig. 3: This figure shows the various ways that a video can be broken into clips. By default the entire video is passed directly on to the user defined preprocessing without any modifications. A variable number of clips can be extracted from a video if only the `clip_length` parameter is given. Alternatively, a set number of clips of a certain length can be extracted from every video using the `num_clips` parameter. If a video does not contain enough frames to extract the requested number of clips, the video will be looped until the necessary frame count is reached. There also exists an `offset` parameter which can be set to randomly modify where from the beginning of the video the function begins to extract clips. The `stride` parameter can be used to set a number of frames that will be common between sequential clips. This `stride` parameter can be set to a negative value to skip a set number of frames between sequential clips.

4 Model Definition Block

4.1 Preprocessing

The class definition of a model within TARP can be divided into four main components as shown in Fig. 4. Preprocessing is the first module in the **Model Definition Block** that gets used within **Input Data Block**. Activity recognition models employ a variety of preprocessing methods on a frame or clip-based level for data augmentation. These can include frame-wise cropping, flipping, and resizing, or clip-wise cropping, flipping, temporal offsets, resampling, looping, and more. In keeping with the modular design paradigm, we use individual

functions to define each variant of the preprocessing block. An entire preprocessing pipeline and multiple variants of such pipelines are defined within a file marked with the suffix "_preprocessing.py". This is done to 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 in conjunction with the `preproc_method`. 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.

4.2 Model Architecture

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 all desired layers from within the defined network. Any accompanying functions to help quickly and/or recursively define a network can be added outside of the inference function.

4.3 Loss and Loading Weights

As a minor deviation from standard practices, we attach the definition of a loss to a model 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 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

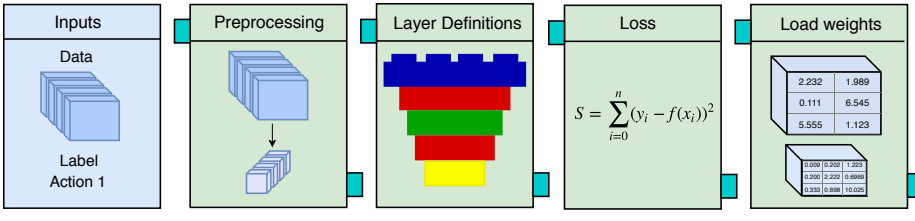

Fig. 4: The components of the **Model Definition Block**: 1) Preprocessing, 2) Layer definitions, 3) Loss, and 4) Loading weights, are shown in green. These four modules are to be implemented by the user when creating a new model.

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 5.3. The final values returned by this function are the parameter weights loaded from an external numpy file.

4.4 Automatic Template Generation

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 functions to define 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 SOTA activity recognition architectures I3D [4], C3D [3], ResNet50 [5], and TSN [6].

5 Execution Block

The **Execution Block** forms 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. 5 illustrates this concept and highlights the modules used within each phase. An algorithmic overview of each phase, followed by a detailed description of the various modules used within them are provided in the following sections.

5.1 Training: 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 Definition Block** 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

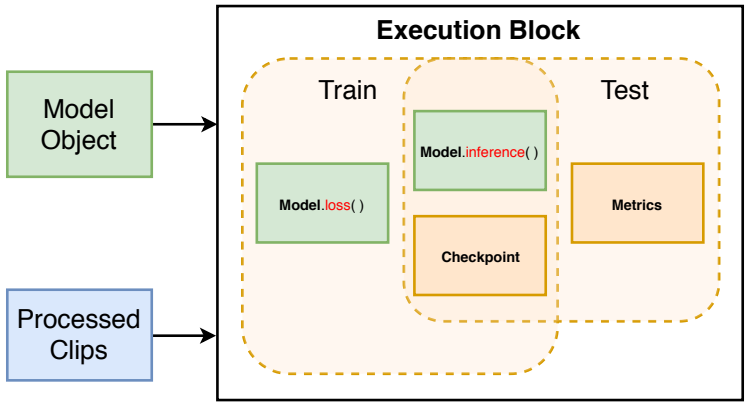

Fig. 5: Training and Testing are the two major phases within the execution block. Checkpoint-based functions are part of both phases while metrics are calculated during the testing phase, after models are trained.

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 **Input Data Block** and interfaced to the model. The main data retuned from **Input Data 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 Definition Block** 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 values stored in the gradients_array. This the final stage in the definition of the tensorflow graph required for training.

procedure TRAINING
    Model_Definition_Block.setup(model_params)

Checkpoint.load_param_weights(default or specific file)
Data, Labels = **Data_Input_Block**.load_data(expt_params)

for each GPU within given node **do**
Model_Definition_Block.inference(Data, model_params)
tower_loss = **Model_Definition_Block**.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 module**.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 training loss plateaus. The repeated application of train_op forms the most crucial and final step in the training process flow.

5.2 Testing: Flow of processes

Testing begins similarly to training in that the specified model gets defined and initialized according to the **Model Definition Block**. After a model object has been instantiated, the model weights are loaded. There is an option to either load the weights from the default weights file specified within the **Model Definition Block**or to initialize the model to random weights. Most commly, however, the **Checkpoint module**is used to load the weights of a model that has been trained from scratch or fine-tuned within TARP.

Once the model weights are accessible, the dataset will be loaded into the system following the process described in the **Input Data Block**in Sec. 3. The shape of input data is set to allow for any number of different videos or clips to be loaded for testing within a single `batch`. This is useful in C3D for example, where a consensus across the outputs for multiple clips is calculated which will act as the prediction for an entire video.

Before 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 "Device ID" list can still be given to TARP to determine the correct GPU to be used. 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 calling **Model_Definition_Block.**inference in the scope of each ID.

The **Model_Definition_Block.**inference call will return the output of the layer specified in the `return_layer` argument. By default, the `logits` layer is returned by inference and it is of the shape [`batch_size`, `sequence_length`, `output_dims`]. In general, a softmax function is applied whenever `logits` are returned. However, the application of softmax can be removed using the `useSoftmax` argument. With the model output acquired, the final steps of preparation for testing is to set up the output directory structure (**Metrics module** detailed in Sec. 5.4), and to load weights into the model (**Checkpoint module**in Sec. 5.3).

The testing block iterates through the `num_vids` specified, which generally includes an entire testing split of a dataset. Within each iteration, the tensorflow session is run to extract the model's `output_predictions`, ground truth `labels`, and all video `names` in the current batch. If the `avg_clips` argument has been specified, then the `output_predictions` are averaged over their first dimension, which is 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 module**for logging and storage.

```
procedure TESTING
    Model_Definition_Block.setup(model_params)
    Checkpoint.load_param_weights(default or specific file)
    Data, Labels = Data_Input_Block.load_data(expt_params)

    for one GPU do
        Model_Definition_Block.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, axis=0)
        end if
        Metrics.log_prediction(output_predictions)
        update(num_videos_loaded)
    end while
```

end procedure

Once all videos have been tested, the thread coordinator for the input queues is stopped and the all predictions from the **Metrics module**are saved in a numpy format.

5.3 Checkpoint Module

The **Checkpoint module**contains utility functions that handle all the essential operations w.r.t. checkpoints. Checkpoints themselves are stored in a custom format as shown in Fig. 6. 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 each 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. 6.

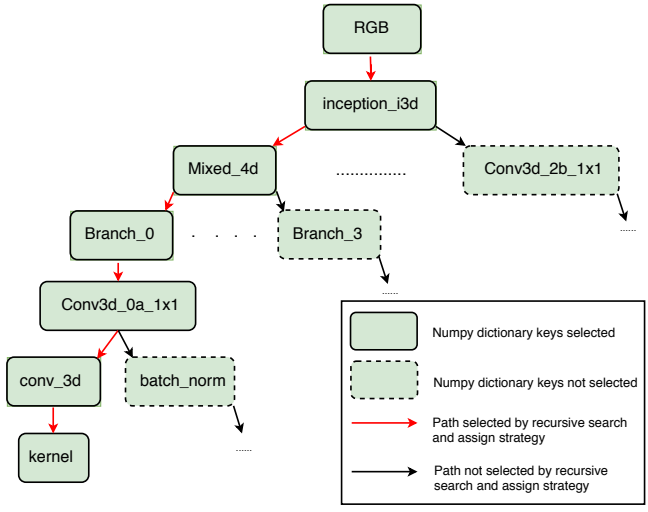

Fig. 6: The custom numpy dictionary format used to store weights of a model is illustrated here. Each block defines the set of keys available at a particular depth within the numpy final. The leaf block is associated with the final numpy array that is assigned to a tensor whose name can be reconstructured by following the names of each block traversed to reach the leaf block. When initializing a model from a pre-existing dictionary, a depth-first search and assign strategy is used to find and assign tensor values.

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 platform.

5.4 Metrics Module

TARP offers a **Metrics module** for testing and logging the activity recognition performance of a model as well as extracting features from a model. This **Metrics module** is incorporated in the **Testing Block** as a class which gets instantiated in test.py. During testing, the model output, correct label, and name of each video gets stored in this **Metrics** object. Depending on the layer from which the outputs, either features or predictions, are extracted, they can be too large in size to efficiently store in memory. Thus all outputs, labels, and names get stored in a temporary hdf5 file format and then deleted at the end of each testing session. HDF5 files are used due to their quick reading and writing capabilities. Just before getting deleted, any final video predictions are saved as numpy files if applicable. Additionally, during testing, the chosen metric and any other specified variable 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 tech-

nique which uses the softmax per-class predictions of a model and averages them across all frames of an input video. From these averaged predictions, the arg. maximum value is used as the overall class prediction for that video. Last frame prediction simply uses the softmax per-class prediction of the last frame in the video to generate the overall class prediction. Last frame classification is referenced originally in the ConvNet + LSTM model [7]. 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 videos in the UCF101 dataset in C3D.

Since the output predictions 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, it refers to clips extracted from one video that are used for testing which will all get averaged to produce a single video-level prediction. On the other hand, TSN would take multiple videos and refer to each as a single clip, thus a single batch containing different videos would each be logged separately. TARP is robust enough to handle both such cases.

As previously mentioned, TARP provides the ability to extract features from a model, as shown in Fig. 7. Features can be extracted for a multitude of reasons including to train a linear SVM which allows TARP to accomplish more than just activity recognition. Any of the state-of-the-art models that have been implemented, or a 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 a given model. When features are being extracted, the output of a model is not required to be

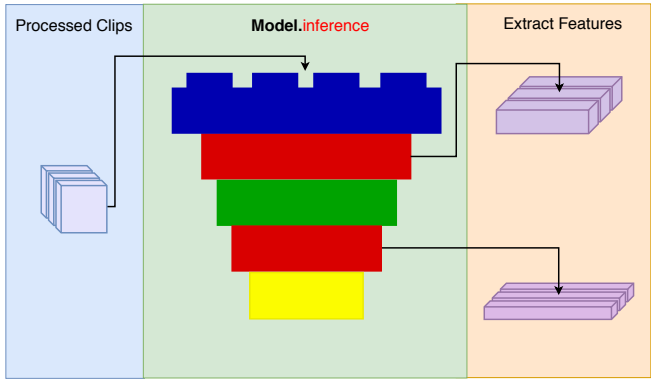

Fig. 7: The **Metrics module**utilizes a variety of metrics to measure the performance of a model. It also allows features to be extracted from any defined model using any appropriately formatted dataset. This figure conceptualizes the capability of features being extracted from various layers within a model.

of any standardized form. 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.

6 Benchmarks

In this section we will be providing numerical evaluations of TARP.

Table 1 show the performance on four SOTA models on the HMDB51 and UCF101 datasets. Results for C3D, I3D, TSN, and ResNet50+LSTM are compared between those generated in TARP and those given by the original authors of the models.

Table 1: Mean recognition accuracies are shown for various SOTA action recognition models across split 1 of HMDB51 and UCF101. Values marked with a (*) indicate that Accuracies are reported for each baseline model and preprocessing technique combination.

Base Model	Pretraining	HMDB51		UCF101	
		Our Acc	Author Acc	Our Acc	Author Acc
C3D	Sports-1M	–.–	50.30 *	–.–	82.30 *
I3D	Kinetics	–.–	74.30 *	–.–	95.1 *
TSN	ImageNet	–.–	54.40	–.–	85.50
ResNet50+LSTM	ImageNet	–.–	43.90	–.–	84.30

7 Future Endeavors

References

1. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al.: Tensorflow: A system for large-scale machine learning. In: OSDI. Volume 16. (2016) 265–283
2. Bradski, G.: The OpenCV Library. Dr. Dobb's Journal of Software Tools (2000)
3. Ji, S., Xu, W., Yang, M., Yu, K.: 3d convolutional neural networks for human action recognition. IEEE Transactions on Pattern Analysis and Machine Intelligence **35**(1) (Jan 2013) 221–231
4. Carreira, J., Zisserman, A.: Quo vadis, action recognition? a new model and the kinetics dataset. In: 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), IEEE (2017) 4724–4733
5. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. arXiv preprint arXiv:1512.03385 (2015)
6. Wang, L., Xiong, Y., Wang, Z., Qiao, Y., Lin, D., Tang, X., Van Gool, L.: Temporal segment networks: Towards good practices for deep action recognition. In: European Conference on Computer Vision, Springer (2016) 20–36
7. Donahue, J., Anne Hendricks, L., Guadarrama, S., Rohrbach, M., Venugopalan, S., Saenko, K., Darrell, T.: Long-term recurrent convolutional networks for visual recognition and description. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. (2015) 2625–2634