Implementing a paper titled: "A Self-Adaptive Proposal Model for Temporal Action Detection based on Reinforcement Learning"

A report submitted for comprehensive evaluation

by

Tanay Agrawal

ID No: 2015B4A80567P

Under the guidance of

Dr. Surekha Bhanot

Professor

Department of Electrical and Electronics Engg./ Instrumentation

BITS Pilani, Pilani Campus.

BITS Pilani, Pilani Campus

First Semester 2018-19

Acknowledgement

The success and final outcome of this project required a lot of guidance and assistance from many people and I am extremely privileged to have got this all along the completion of my project. All that I have done is only due to such supervision and assistance and I would not forget to thank them.

I respect and thank Dr. Surekha Bhanot for providing me an opportunity to do the project work at BITS Pilani, Pilani Campus and giving me guidance which made me complete the project duly. I am extremely thankful to her for providing such a nice support, although she had busy schedule managing the corporate affairs.

Tanay Agrawal

Table of Contents

Introduction

The research being implemented is titled “A Self-Adative Proposal Model for Temporal Action Detection based on Reinforcement Learning”. It was written by Jingjia Huang, Nannan Li, Tao Zhang, Ge Li from the School of Electronics and Computer Engineering, Peking University, Shenzhen Graduate School. It was published in ArXiv on 22 June, 2017.

Due to the continuously booming of videos on the internet, video content analysis has attracted wide attention from both industry and academic eld in recently years. An important branch of video content analysis is action recognition, which usually aims at classifying the categories of manually trimmed video clips. Substantial progress has been reported for this task in recent times. However, most videos in real world are untrimmed and may contain multiple action instances with irrelevant background scenes or activities. This problem motivates the academic community to put attention to another challenging task - temporal action detection. This task aims to detect action instances in untrimmed video, including temporal boundaries and categories of instances. Methods proposed for this task can be used in many areas such as surveillance video analysis and intelligent home care. Temporal action detection can be regarded as a temporal version of object detection in image, since both of the tasks aim to determine the boundaries and categories of multiple instances (actions in time/ objects in space).

Most of modern approaches (Shou, Wang, and Chang 2016; Zhu and Newsam 2016; Xiong et al. 2017) usually solve the problem via a two-step pipeline: firstly generate a set of class independent action proposals, which are obtained via running a action/background classifier over a video at multiple temporal scales; then the proposals are classified by the pre-trained action detector, and post processing such as non-maximum suppression is applied. However, such extensive search for action localization is unsatisfying in terms of both accuracy and computational efficiency. Like the human detects the action through successively altering the span of attended region to narrow down the difference between the bounds of current window and that of true action region, the optimal algorithm should be the process of sequential, iterative observation and refinement consuming search steps as less as possible.

In the chosen paper, a class-specific action detection model called SAP that learns to continuously adjust the current region to cover the groundtruth more precisely in a self adapted way is proposed. This is achieved by applying a sequence of transformations to a temporal window that is initially placed at the beginning of the video and finally finds and covers action region as large as possible. The sequence of transformation is decided by an agent that analyzes the content of the current attended region and select the next best action according to a learned policy, which is trained via reinforcement learning based on Deep Q-learning algorithm (Mnih et al. 2015). Different from existing approaches that locate the action following a fixed path, our method generates various search trajectories for different action instances, depending on the video scenarios, the starting search position and the sequences of actions adopted. As a result, the trained agent will locate a single instance of an action in about 15 steps, which means that the model only processes 15 successive regions of an image to explore an uncover video segment, thus it is of great computational efficiency to compare with sliding window based approaches.

The model draws the inspiration from works that have used reinforcement learning to build active models for object localization in image (Caicedo and Lazebnik 2015; Jie et al. 2016; Bellver et al. 2016). However, we can not handle the video in a top-down way that is proved to perform effectively for image object localization, as the duration of the video is usually too long (from hundreds to thousands frames). We start the search from a position initially placed

at the beginning of the video, which will terminate until a instance of action has been found or the maximum transformation steps has been reached, and then a new search begins from the position on the right side away from current attended region. They have incorporate temporal pooling operation with feature extraction process to better represent the long video segment and design a ”jump” action to avoid the agent trapping itself in the region where no action occurs. We conducted a comprehensive experimental evaluation in the challenging THUMOS’14 dataset (Jiang et al. 2014), and the results demonstrate that SAP can achieve competitive performance in terms of precision and recall via a small number of action proposals.

Literature Review

Action Recognition. This task has been attended for a few years, and a large amount of research work have been done . In early years, researchers often tackle the problem based on hand-crafted visual features. Recently, impressed by the huge success of deep learning on image analysis task, some approaches have introduced deep models, especially Convolutional Neural Network (CNN), for better excavation the spatial-temporal information included in the video clip. Simonyan and Zisserman propose the two-stream network architecture with one branch processing RGB signal and the other one dealing with optical-flow signal. Tran et al. construct C3D model, which operates 3D convolution in spatio-temporal video volume directly and integrates appearance and motion cues for better feature representation. There have been also other efforts that attempt to combine frame-level CNN feature representation and long-range temporal structure to cope with input videos of long duration. Up to now, deep learning based approaches have achieved state-of-the-art performances.

Temporal Action Detection. Different from action recognition where actions are included in a trimmed video clip and the aim is to predict the category, temporal action detection needs to not only classify the action but also give out temporal localization. Most existing approaches address the problem via sliding window strategy for candidates generation and focus on feature representation and classifier construction. Shou et al. utilize a multi-stage CNN detection network for action localization, where background windows are first filtered out by a binary action/background classifier based on C3D feature, then an action detection network incorporated both classification loss and temporal localization loss is trained for candidate refinement. By the limitation of 16-frames input of C3D model, they select 16 frames in uniform from the whole video, which is inferior to temporal pooling operation utilized in our approach. Gao et al. decompose the input video into short video units, and pool features extracted from a set of contiguous units for representation of long video clip, and meanwhile employ a coordinate regression network to refine the temporal action boundaries. Our approach also includes location regression, whose regression offsets are calculated via the relative deviation rather than the absolute value, thus it will facilitate the model to converge more efficiently. Unlike the works mentioned above, Yeung et al. propose an attention based model that predicts the action position through a few of glimpses, which is trained via reinforcement learning. The difference between their work and ours is that our approach locates the action through continuously adjusting the span of current window not predicting the bounds directly.

Object Detection. Most of recent approaches for object detection are built upon the paradigm of "proposal + classification". Object proposals are usually either generated by methods relied on hand-crafted low-level visual cues, such as SelectiveSearch and Edgebox, or produced by fully convolutional network implemented on CNN features extracted from anchor boxes arranged uniformly on the image, such as Faster R-CNN. However, generating

too many proposals for a image with only one or two objects is unnecessary and computational inefficienct. Some works attempt to reduce the number of proposals with an active object detection strategy. Caicedo et al. learns optimal policy object image via Deep Q-Learning, where it starts from the whole image in a top-down way and adaptively adjusts the window scale and position to focus on the true region. Jie et al. propose an effective tree-structured reinforcement learning right expand of left uncovered end the refinement move left expand shrink move learns right to approach, which balance the exploration new jump objects and of covered ones, and can localize multiple objects in a single run. Inspired by, we design a reinforcement learning based approach for temporal action localization, which locates action instances within the long untrimmed video via the learned policy in a bottom-up way, and meanwhile utilizes a regression network to refine the predicted temporal window boundaries.

Proposed Model in the chosen paper: Self-Adaptive Proposal

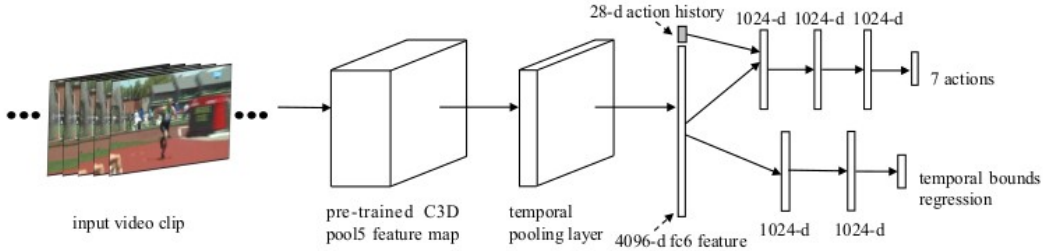


fig 1: Architecture of the proposed model

In this section, the action-proposal generation model is presented, which is self-adapted and will gradually adjust its predicted results according to the content of attended window and the history of executed actions to cover the true action region as accurate as possible in a few steps. We cast the problem of temporal action localization as a Markov Decision Process (MDP), in which the agent interacts with the environment and makes a sequence of decisions to achieve the settled goal. In our formulation, the environment is the input video clip, in which the agent has an observation of the current video segment, called temporal window, and restructures the position or span of the window, to achieve the goal of locating the action precisely. The agent receives positive or negative rewards after each decision made during the train phase to learn an effective policy. Besides, we construct a regression network to refine the final detection results to promote the accuracy of localization. The framework of our proposal generation model is illustrated above. In the following subsections, the set of actions A, the set of states S, and the reward function R(s, a) of MDP and the regression network are discussed in detail. To avoid confusion, the action performed by the actor in the video is called motion in this section, and in other sections the meaning of action is determined by the context.

C3D CNN Model, Temporal Pooling and fc6 layer:

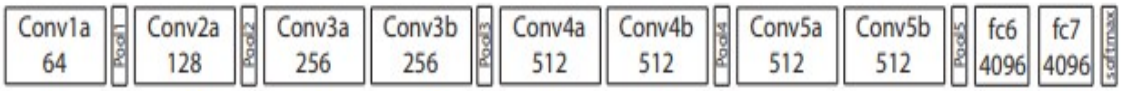


fig 2: Architure of C3D CNN Model

All of 3D convolution filters are 3 × 3 × 3 with stride 1 × 1 × 1. All 3D pooling layers are 2 × 2 × 2 with stride 2 × 2 × 2 except for pool1 which has kernel size of 1 × 2 × 2 and stride 1 × 2 × 2 with the intention of preserving the temporal information in the early phase. Each fully connected layer has 4096 output units. Only feature map from C3D network is used i.e. till pool 5 layer. Then, temporal pooling layer is added if the whole video is taken as input (discussed below). After that fc6 layer is added. These are used as pretrained models on Sports-1m dataset and fine tuned for UCF101.

MDP Formulation (upper network after fc6 layer in fig 1):

Actions. The set of actions A can be divided into two categories: one group for transformation on temporal window, such as "move left", "move right", "expand left", and the remaining one for terminating the search, "trigger", as shown in Fig. 2. The transformation group includes regular actions that comprises of translation and scale, and one irregular action. The regular actions vary the current window in terms of position or time span around the attended region, such as "move left", "expand left" or "shrink", which are adopted by the agent to increase the intersection with the groundtruth that has overlaps with the current window. The irregular action, namely "jump", translates the window to a new position away from the current site to avoid that the agent traps itself round the present location when there is no motion occurring nearby. The change caused by any regular actions at each time to the window equals to a value in proportion to the current window size. For instance, supposing that current window is denoted as $[x_l, x_r]$, where x_l and x_r stand for the left and right boundary respectively. The action "move left" translates the window to a new site of $[x_l', x_r']$ with $x_l - x_l' = x_r' - x_r = \alpha * (x_r - x_l)$, while for action "expand left" scales the window with the change of $x_l - x_l' = \alpha * (x_r - x_l)$ and $x_r' = x_r$. Here, $\alpha \in [0, 1]$ is a parameter that can give a trade-off between search speed and localization accuracy. In this paper, we set $\alpha = 0.2$. The action "jump" selects a new window randomly from the left or right side, which has the same size with the current window, being a distance away from the present site. The regular actions make the agent gradually adjust its position to cover the motion more accurately when it has found one; while the action "jump" let the agent explore unknown region that may contain the motion in a discontinuous and efficient way. The action "trigger" is employed by the agent whenever it considers that a motion has been localized by the current window, and stops the sequence of current search, and restarts a new search for the next motion with an initial window position away from current site.

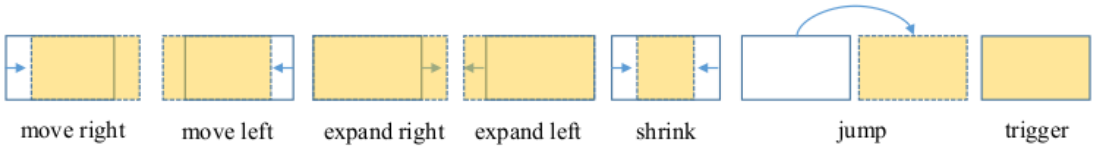


Fig 3: Actions of the MDP

State. The state of MDP is the concatenation of two components: the presentation of current window and the history of taken actions. To describe the motion within current window generally, the feature extracted from the C3D CNN model, which is pretrained on Sports-1M

and finetuned on UCF-101, is utilized as the presentation. Here, we choose the feature vector from fc6 layer (4096 dimension) in our problem, consideing its good abstract representation for the semantic information about the motion. The original C3D model can only accept 16 frames as input, however, the duration of temporal window is always far more than that number. To tackle with the problem, we design two different solutions: i.) uniformly select 16 frames from the whole duration ; ii.) fed all the frames into the C3D model and add a additional pooling layer (average pooling for our problem) between the "pool5" layer and the "fc6" layer, which condenses the dimension of extracted feature vector from "pool5" to the value specified by the C3D model. The history of the taken actions is a binary vector that tells which action has been adopted by the agent in the past. Each action in the history is represented by a 7-dimension binary vector where all the values are zero except the one corresponding to the taken action. In the experiment, we totally record 10 past actions as the history. The history of taken actions informs the agent the search path that has been passed through and the regions already attended, so as to stabilize search trajectories that might get stuck in repetitive cycles.

Reward Fuction. The reward function R(s, a) provides a feedback to the agent when it performs the action a at the current state s, which awards the agent for actions that will bring about the improvement of motion localization accuracy while gives the punishment for actions that leads to the decline of the accuracy. The quality of motion localization is evaluated via the simple yet indicative measurement, Intersection over Union (IoU) between current attended temporal window and the groundtruth. Supposing that w stands for the current window and g represents the groundtruth region of motion, then the IoU between w and g is defined as IoU(w, g) = span(w ∩ g) / span(w ∪ g). The reward function is proportional to the difference between IoUs of two successive states s and s 0 , where the agent moves to state s 0 from s by executing the action a. Specially, it is formulated as following:

(1)

$$r(s,a) = \max_{1 \le i \le n} \operatorname{sign}(\mathrm{IoU}(w', g_i) - \mathrm{IoU}(w, g_i)),$$

where w’ and w are attended windows corresponding to state s 0 and s respectively, n is the number of groundtruths within the input video. The reward function returns +1 or -1. Equation 1 indicates that the agent receives the reward +1 if the new window w 0 has more overlap with any of the groundtruth than the previous window w, while the reward -1 otherwise. Such binary reward value makes the agent clearly realize that at present state, which action drives the attended window towards the groundtruth, and thus accelerates the convergence rate of the model during training phase. In addition, such reward-function scheme facilitates better localization towards motion regions especially for the video with multiple motion instances, as there is no limitation on which motion should be focused on at each state. The "trigger" action has a different reward function scheme, as it leads to the termination of search and there is no next state. The reward of "trigger" is determined by a piecewise function of IoU threshold, which can be presented as following:

(2)

$$r_e(s) = \begin{cases} +\eta & \text{if } \mathrm{IoU}(w, g) \ge \tau \\ -\eta & \text{otherwise.} \end{cases}$$

In this equation, e represents the "trigger" action, η is the reward value and chosen as 3 in our experiment, τ is the IoU threshold, which controls the tradeoff between the localization accuracy and computational overhead. The large τ will encourage the agent to locate the motion more precisely, however it consumes more action steps to complete the search. In training phase, we do not stop the search process when the agent correctly performs the action "trigger" for the first time, and let it continuously explore uncover regions. Therefore, our model recognizes many termination states that have IoU with groundtruth more than τ . We utilize τ = 0.5 for our problem, and find that larger τ , such as 0.6 or 0.7, gives rise to negligible promotion on recall value, which is validated by the experiments.

Deep Q-learning. The goal of the agent is to maximize the sum of discounted rewards that are received through continuously transforming the current attended window during a sequence of interactions with the environment (an episode). In other words, the agent needs to learn a policy π(s) that specifies an optimal action at current state s in the view of maximizing the long-term benefit. Due to the lack of state transition probability and the model free environment, we utilize reinforcement learning, specially Deep Q-learning, to estimate the optimal value for each state-action pair. In this paper, we follow the deep Q-learning framework proposed by Mnih et al. that estimates the action-value function via a neural network. The architecture of our Deep Q-Network (DQN) is illustrated as the up branch of Fig. 1. The C3D CNN model is just used for feature extraction, and we do not train the whole pipeline for the full feature hierarchy learning, due to the good generalization of CNN model pretrained on large dataset and short of sufficient motion detection data for jointly training both two networks. During training phase, the agent operates multiple episodes with randomly initialized positions for each video clip. We train separate DQN for each motion category and follow the epsilon-greedy policy. Specially, the agent randomly selects an action from the whole action set with probability epsilon at current state, while greedily chooses the optimal action according to the learned policy with probability 1-epsilon. During the whole training epochs, epsilon is annealed linearly from 1.0 to 0.1, which gradually shifts from exploration to exploitation. We also incorporate the replay-memory scheme to collect various transition experiences from the past episodes, from which each record may be repeatedly used for model updates, in favor of breaking short-term correlations between states. A minibatch (e.g. 200 records) is randomly sampled from replay-memory as training samples to update the model at each time.

Regression Network (lower network after fc6 layer in fig 1):

Inspired by Fast R-CNN where a regression network is incorporated to revise the position deviation between the predicted result and the groundtruth, we also introduce a regression model to refine the motion proposals. As shown in the down branch of Fig. 1, the regression channel accepts 4096-dimension feature vector as input and gives out two coordinate offsets on both starting and end moment. Unlike spatial bounding box regression, in which coordinate scaling is needed due to various camera-projection perspectives, we directly utilize original temporal coordinate (i.e. frame number) for offsets calculation leveraging the advantage of unified frame rate among video clips in our experiment. The regression biases are represented as the ratio of position deviation relative to the predicted span, which are defined as following:

$$o_s = (s_p - s_g)/(e_p - s_p), \quad o_e = (e_p - e_g)/(e_p - s_p), \tag{3}$$

where s p and e p are frame indexes for predicted starting and end moment, while s g and e g are frame indexes for the matched groundtruth. The loss for temporal coordinate regression, L reg , is defined as following:

(4) $$L_{reg} = \frac{1}{N_{end}} \sum_{i=1}^{N_{end}} (|o_{s,i}| + |o_{e,i}|),$$

where N end is the number of actions that correctly perform "trigger" in a minibatch. In other words, we only regress the position of temporal window whose IoU with groundtruth is more than 0.5. We utilize L 1 norm to make the loss be insensitive to outliers.

Methodology

For implementing this research paper, I have first understood all the concepts that are being used and also those from which the paper is inspired. Then, some small projects have been done for understanding the topics better. Finally, the paper has been implemented.

The topics studied are Convolutional Neural Networks (also C3D model), Reinforcement learning (Deep Q Networks and Markov Decision Process), Fast R-CNN, optimisation. The learnings have been discussed below. The material shown is from the notes made and may miss a few things, but is exhaustive for this particular project.

Learning

Convolutional Neural Network:

- These use a filter to slide over the image.
- The filter has the same depth as the image.
- When sliding, filter takes dot product for each instance.
- For each convolution, there can be multiple filters as each filter has a different purpose. So, we get ‘n’ activation maps if there are ‘n’ filters.
- If N, F are the sizes of input and filter and S is the stride size, then the activation map is of size ((N-F)/S) + 1.
- Padding can be used to change the activation map size.

Different popular architectures of CNNs were studied: AlexNet, VGGNet, GoogLeNet, ResNet, Network in network and few others in less detail like FractalNet, DenseNet, SqueezeNet. These are not being discussed as they are not relevant to this project.

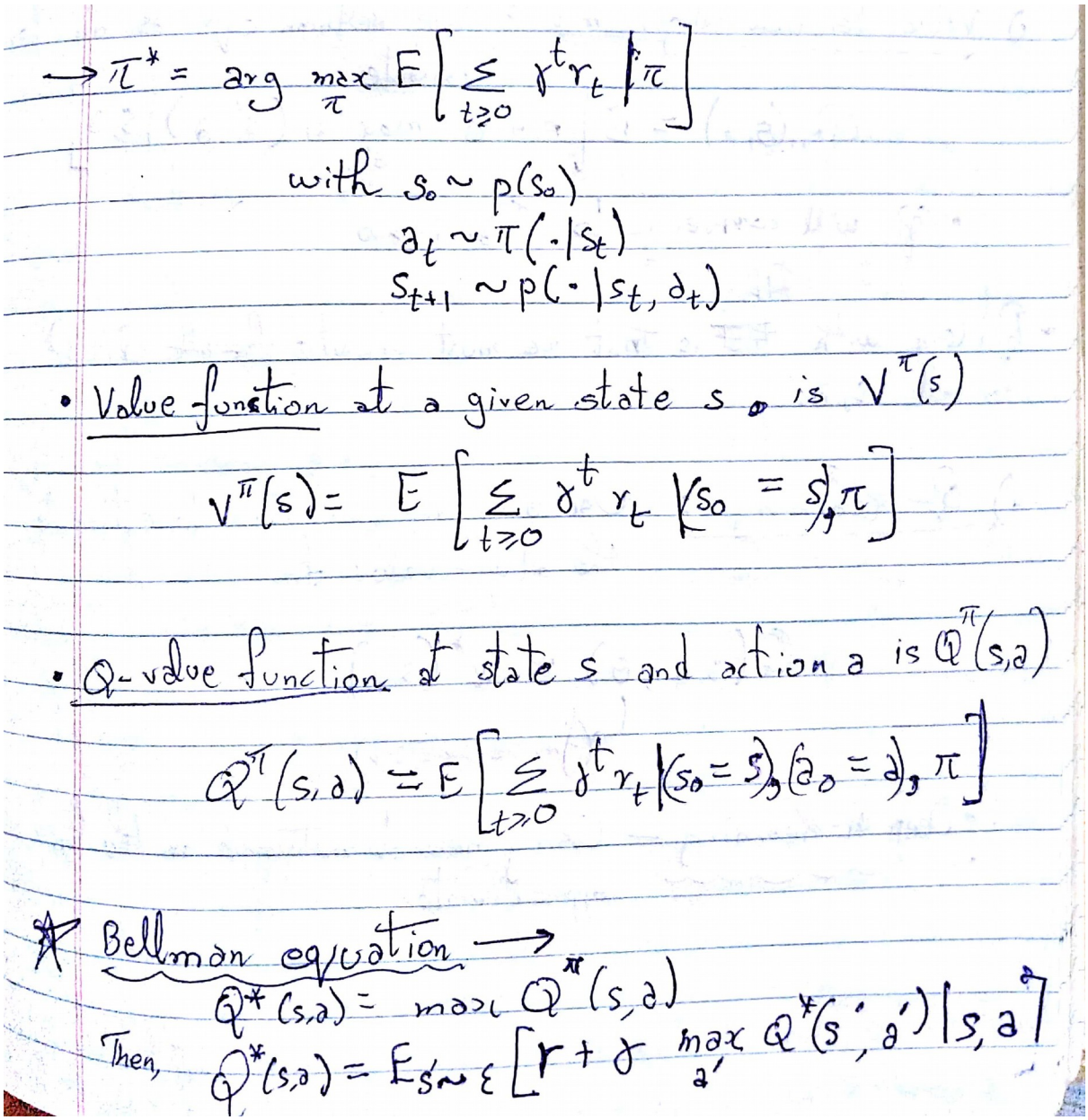
C3D CNN model is used in the research paer. This is similar to a 2d CNN, the difference is that the pooling layer is 3d in place of 2d (an extra temporal deminsion) and traverses over a video (2d image with an additional temporal dimension). The C3D model only has 16 frames as input but videos are longer in the dataset (THUMOS 14), so what has been done is that, another pooling layer has been used after the 3d CNN to fit this.

Reinforcement Learning (RL):

Markov Decision Process is the mathematical formulation of the RL problem. It depends on set of possible states (S), set of possible actions (A), distribution of reward given S and A (R), transition probability i.e. distribution over next state given S and A (P), discount factor (gamma).

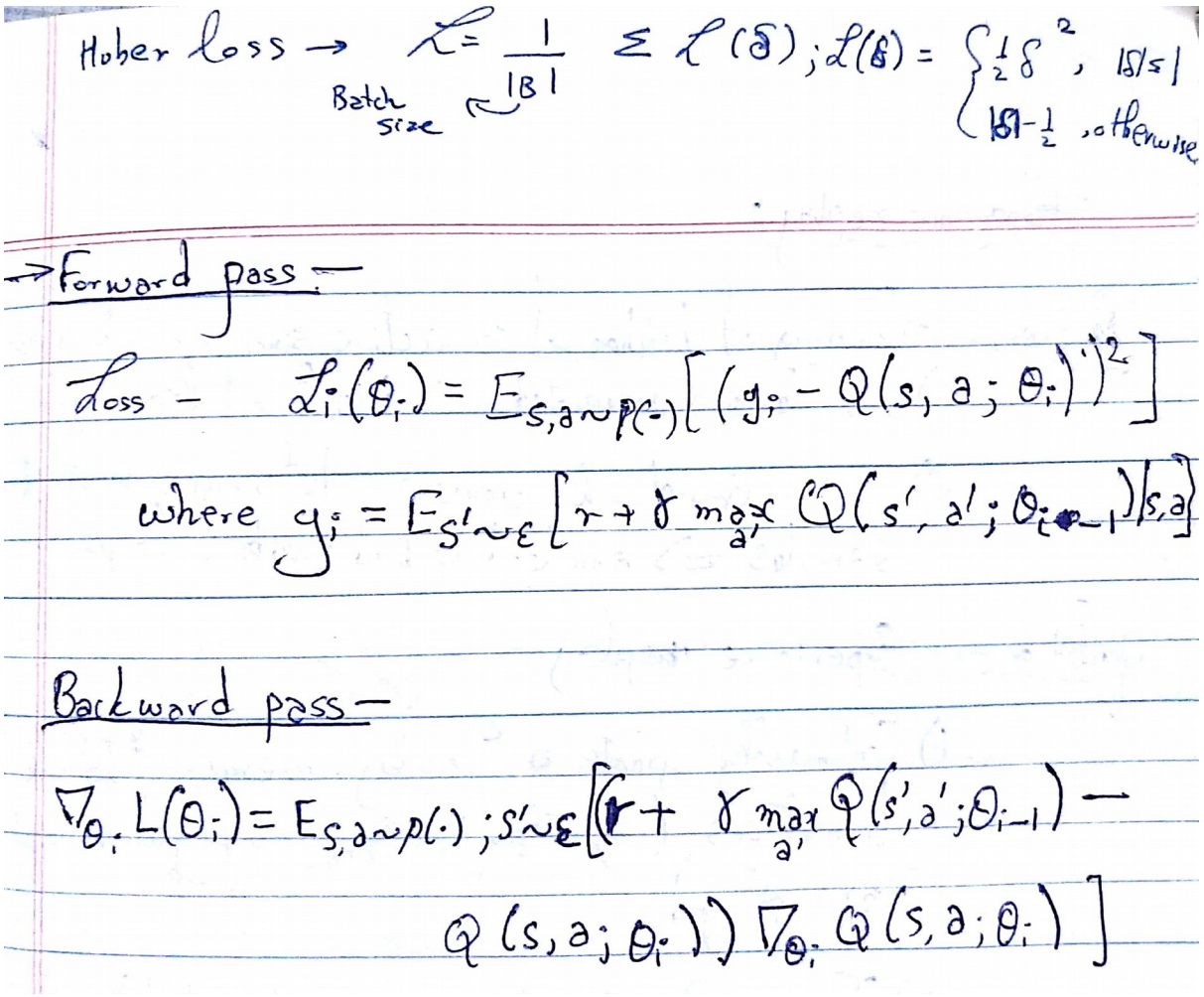
A policy pi is defined from S to A that specifies what action to take in each state.

Objective: To find optimal policy (pi*) that maximmizes discounted reward.



Intution for Bellman equation: If the optimal S,A values for the next time step Q* are know, then the optimal strategu is to take the action that maximizes the expected value of r + gamma * (Q*(s',a')).

Q learning is used for solving for optimal policy. A function approximator (neural networks in deep Q learning) is used to estimate Q*.



Another useful conceppt is Experience Replay.

Problem – Learning of batches of consecutive samples is bad as
1) Samples are correlated, so learning is inefficient.
2) As current paramerters determine next training samples, there can be feedback loops and training might get stuck.

Solution – 1) Continually update a 'replay memory table' of transitions (s_t , a_t, r_t, s_{t+1}) as game episodes are played.
2) Train Q network on random batches from this table.

A small sub-project was made on reinforcement learning:

Cartpole problem: Balancing a pole fixed at one end to a cart that can move in one dimension. Deep Q networks were used for this. The aim was to better understand reinforcement learning. Results for this project have not been ventured into as they will be found out in the implementation of the research paper.

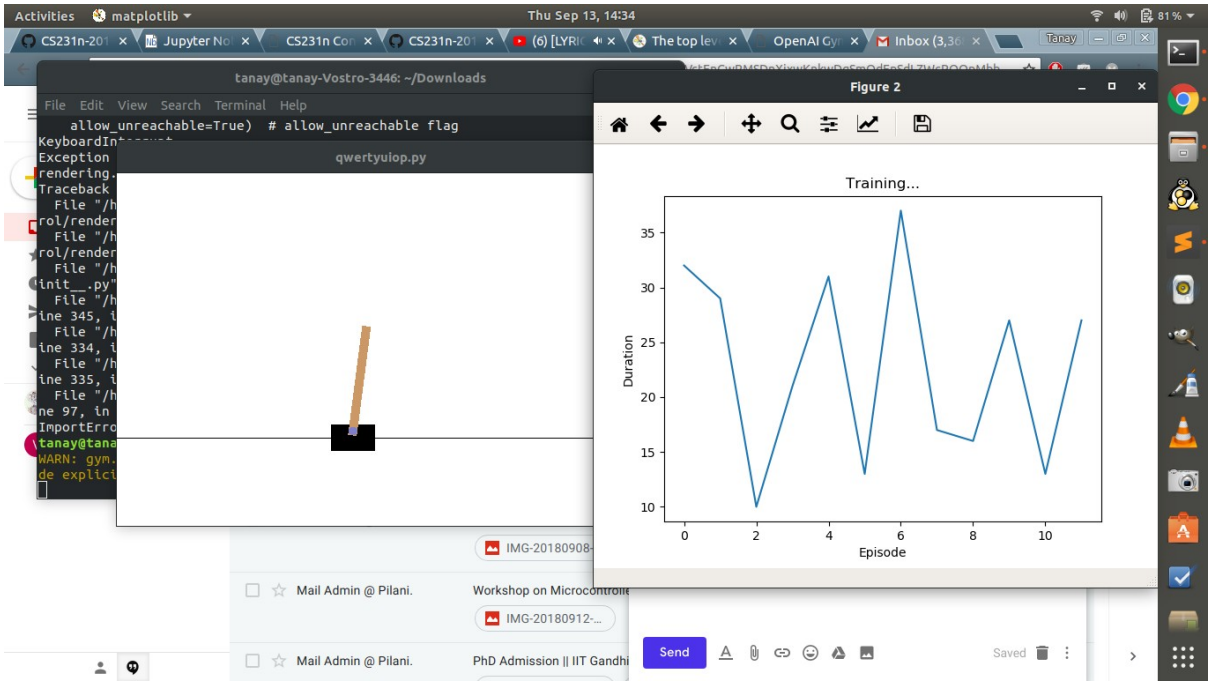


Fig 4: Training of Cartpole problem

In this, experience replay is used.

Gym is a library which provides the environment for the MDP for this problem.

For the forward pass, Huber loss (defined above) is used. It is less sensitive to outliers than L2 loss function.

Architecture:
3frames input > convolution layer (16 filters, kernel size = 5, stride = 2) > batch normalisation > convolution layer (32 filters, kernel size =5, size = 2) > batch normalisation > convolution layer (32 filters, kernel size =5, size = 2) > batch normalisation > 448 d FC layer with 2 outputs (as there are 2 Q values, for moving left or right).

Another technique used is that, the model initially learns only from the replay memory. Then, once it starts learning, it uses the learnt pattern with an exponentially increasing probability, otherwise it keeps learning from the replay memory. This is used to prevent the function to converge to a local minimum i.e. cover all possible patterns of motion.

The pole is assumed to be balanced if it is less than 15 degrees from the vertical axis. Otherwise it moves on to the next episode.

In the figure above, the left windows shows the animation of the cartpole problem and the cart eventually learns to balance the stick on it.

The right window shows the time for which the the pole remains on the cart. It decays while oscillating. It oscillates because of the randomness of the state action pairs selected from replay memory and optimal policy.

Fast R-CNN:

These are same as CNN with an additional concept of 'region of interest' that gives boxes where objects may appear. This idea is used in this paper for the regression network.

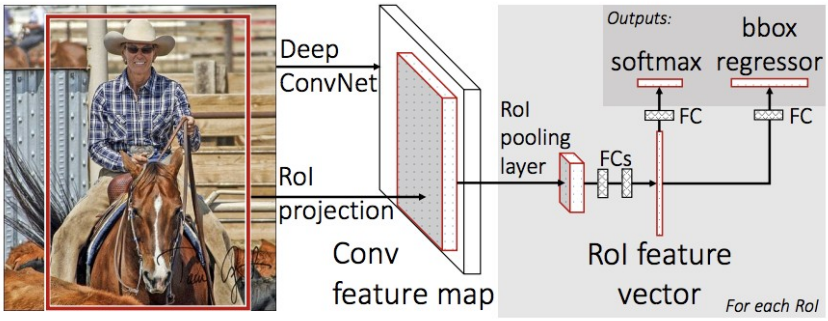


Fig 5: Fast RCNN approach and architecture

Implimentation of the research paper(Code):

The code is written in lua using torch7. The files and folders in the codebase are:

Read_Input_Cmd.lua : This file has functions for commands to be put in the termonal for running the code

Reinforcement.lua : This file is for defining DQN network

Metrics.lua : This file implements all the functions in the code that are not available as functions in torch7

Mask_and_Actions.lua : This file has all the functions for performing the actions of the MDP and some functions for the interval of the mask (proposed temporal extent of the action)

Interface.lua : This has functions for getting data from folder, converting data for suitable input to C3D and forward pass for C3D and fc6 layer

Interface_PyramidPooling.lua : This has functions for getting data from folder, converting data for suitable input to C3D and forward pass for C3D, Pooling layer and fc6 layer

Training_SAP.lua : Training model

Training_DQNRGN.lua : Training model without pooling layer

Validate_SAP.lua : Validation for the trained model

Validate_PoolingDQN.lua : Validation for the model without regression layer

Validate_DQNRGN.lua : Validation for the model without pooling layer

Validate_DQN.lua : Validation for the model without regression and pooling layer

Log : Folder for log files

Track : Folder for tracking progress of validation files

Thumos : Folder for processed dataset. Thumos dataset for temporal action detection contains videos and files giving ground truth frames. The datasett has been converted to images saved in temporal order in an organised manner as shown below:

```
.
├── dataset          # put your datasets on the project folder and named it with dataset name
│   ├── class        # action class name (CleanAndJerk, BaseballPitch, HighJump...)
│   │   ├──
│   │   │   ├── clip_idx             # index of videos
│   │   │   │   ├── frame_idx.jpg       # video images(from 1 to total sampled image number)
│   │   │   │   ...
│   │   │   │   ├── frame_idx.jpg
│   │   │   │   ├── FrameNum.txt        # total sampled image number
│   │   │   │   ├── gt.txt              # groundtruth interval
│   │   │   │   ├── NumOfGt.txt         # number of groundtruth in the clip
│   │   │   │   └──
│   │   │   ...
│   │   │   ├── clip_idx
│   │   │   ├── ClipNum.txt          # number of clips in this category
│   │   │   └──
│   │   └──
│   ├── class
│   ...
│   ├── class
│   ├── trainlist.t7
│   ├── validatelist.t7
└──
```

The trainlist.t7 and validatelist.t7 are files that give the location of training and validation datasets in the Thumos folder. Their structure is as follows:

```
{
  1 --class
  {
     1 --clip_idx
     ...
  }
  ...
  21 --class
  {
    1 --clip_idx
    ...
  }
}
```

- ClipNum.txt : a single number indicates the number of clips in this category
- FrameNum.txt : a single number indicates the total sampled image number from the clip
- NumOfGt.txt : a single number indicates the number of groundtruth segments in the clip
- gt.txt records the index of first frame and last frame of the groundtruth.It should be arranged as follow:

<start_1> <end_1>
.
.
.
<start_n><end_n>

Note: Testing is done using the validate files.

Training and validation/testing code have been attached below. The training_SAP code is as follows:

For following the code structure, one can look at if conditions and loops. They are explained properly using comments and tthe process is easy to understand. It'll be easier to copy the code in a text editor and save the file as *.lua for reading.

```
require 'Read_Input_Cmd'
require 'Reinforcement3'
require 'Interface_PyramidPooling'
require 'Mask_and_Actions'
require 'Metrics'
```

```
require 'optim'

--read input para
local cmd = torch.CmdLine()
opt = func_read_training_cmd(cmd, arg)

-- create log file
local log_file = io.open(opt.log_log, 'w')
if not log_file then
                print("open log file error")
                error("open log file error")
end

-- read training clip id from files
local training_file = './Thumos/trainlist.t7'
print(training_file)
local clip_table = torch.load(training_file)
local tt = clip_table[opt.class]
if tt == nil then
                error('no trainlist file')
end

local training_clip_table={}
training_clip_table = tt
--for i=1,10 do
--                table.insert(training_clip_table, tt[#tt-10+i])
--end

--set training parameters
local max_epochs = opt.epochs
local batch_size = opt.batch_size
local max_steps = 25

--DQN training trick parameters
local experience_replay_buffer_size = opt.replay_buffer
local gamma = 0.90 --discount factor
local epsilon = 1 -- greedy policy
local trigger_thd = 0.5 -- threshold for terminal

local count_train = torch.Tensor(1):fill(0)
local train_period = torch.floor(opt.batch_size/100)


-- number_of_actions and history_action_buffer_size are globle variables in Reinforcement
local history_vector_size = number_of_actions * history_action_buffer_size
local input_vector_size = history_vector_size + C3D_size
-- define the last action as trigger
local trigger_action = number_of_actions
local jump_action = number_of_actions-1
local act_alpha = opt.alpha

--init replay memory
local replay_memory = {}
```

```
--init reward
local reward = 0
--if opt.model_name is '0', then init DQN model
--else load a saved DQN
local dqn = {}
dqn= func_get_dqn(opt.model_name, log_file)
--load RGN
local rgn = {}
rgn = func_get_rgn('0', log_file)


local fc6 = torch.load('fc6.t7')
fc6:evaluate()

-- set gpu
opt.gpu = func_set_gpu(opt.gpu, log_file)

if opt.gpu >=0 then
                dqn = dqn:cuda()
                fc6 = fc6:cuda()
                rgn = rgn:cuda()
end

local params, gradParams = dqn:getParameters()
local params_rgn, gradParams_rgn = rgn:getParameters()

-- define loss function and trainer
local criterion = nn.SmoothL1Criterion()
if opt.gpu >=0 then crirerion = criterion:cuda() end

--defines loss function and trainer for rgn
local criterion_rgn = nn.AbsCriterion()
if opt.gpu >=0 then crirerion_rgn = criterion_rgn:cuda() end

-- training with optim
-- optim paras for optim
local optimState = {learningRate = opt.lr, maxIteration = 1, learningRateDecay = 0.00005, evalCounter = 0}
local optimState_rgn = {learningRate = 1e-4, maxIteration = 1,learningRateDecay = 0.00009, evalCounter = 0}
local logger = optim.Logger(opt.log_err)
logger:setNames{'Training_error', 'epoch'}
local rgn_err = opt.log_err .. '_rgn'
local logger_rgn=optim.Logger(rgn_err)
logger_rgn:setNames{'Training_error'}


--read dataset
local gt_table = func_get_data_set_info(opt.data_path, opt.class, 1)
print(gt_table)

-- get average length
local count = 0
local len = 0
for i,v in pairs(tt) do
                local tmp_table = gt_table[v]
```

```
                for j,u in pairs(tmp_table) do
                        count = count +1
                        len = len + u[2]-u[1]
                end
end
local avg_len = torch.floor(len/count)
if avg_len < 16 then avg=16 end
print(avg_len)
local max_gt_length = 128 -- max length to split gt
--gt_table = func_modify_gt(gt_table, max_gt_length)

-- load C3D model
local C3D_m = torch.load('c3d.t7');
C3D_m:evaluate()

for i = 1, max_epochs
do
                log_file:write('It is the ' .. i .. ' epoch\n')
                print('It is the ' .. i .. ' epoch')

                for j, v in pairs(training_clip_table)
                do
                        local masked = false
                        local masked_segs={}
                        local not_finished = true
                        local tmp_gt = gt_table[v]
                        local total_frms = tmp_gt[1][3]
                        local gt_num = table.getn(tmp_gt)
                        local available_objects = torch.Tensor(gt_num):fill(1)
                        print('load images')
                        local clip_img = func_load_clip(opt.data_path, opt.class, 1, v,total_frms)

                        log_file:write('\tIt is the ' .. j .. ' clip, clip_id = ' ..
                                                                v .. ' total_frms = '.. total_frms .. '\n')
                        print('\tIt is the '.. j .. ' clip, clip_id = ' ..
                                                                v .. ' total_frms = '.. total_frms)

                        local lp_t = torch.round(total_frms * 10/max_steps/avg_len)-- loop times
                        if lp_t <= 1 then lp_t = 2 end
                        for k = 1, lp_t
                        do
                                log_file:write('\t\tIt is the ' .. k .. ' gt, from '.. '\n')
                                print('\t\tIt is the ' .. k .. ' gt, from '.. '\n')

                                -- init mask, return beg index and end index of mask
                                -- in hjj_mask_and_action
                                local cur_mask = func_mask_random_init(total_frms, masked_segs, avg_len)
                                local old_mask = cur_mask

                                -- iou_table record the iou of each gt and cur_mask
                                -- reset iou_table in the beginning of each loop
                                local iou_table = torch.Tensor(gt_num):fill(0)

                                local old_iou = 0
```

```
		local new_iou = 0
		local overlap = 0

		local new_dist = max_dist


		-- check if available objects left
		if torch.nonzero(available_objects):numel() == 0 then
			not_finished = false
		end

		-- calculate iou for cur_mask and gt
		old_iou, new_iou, iou_table, index = func_follow_iou(cur_mask,

	tmp_gt, available_objects, iou_table)
		overlap = func_calculate_overlapping(tmp_gt[index], cur_mask) -- intersec/cur_mask



		local now_target_gt = tmp_gt[index]

		-- init history action buffer
		local history_vector = torch.Tensor(history_vector_size):fill(0)
		print('\t\t\tInit mask: ' .. cur_mask[1] .. '\t' .. cur_mask[2] .. '\t' .. total_frms..'\n')
		-- get C3D

		--local C3D_vector = func_get_C3D(opt.data_path, opt.class, 1,
		--                                                                        v, cur_mask[1],
cur_mask[2], C3D_m, {}, 27, fc6)
		local C3D_vector = func_get_C3D(clip_img[{ {cur_mask[1], cur_mask[2]},{} }], C3D_m,{},27,fc6)

		local input_vector = torch.cat(C3D_vector, history_vector, 1)
		if opt.gpu >=0 then input_vector = input_vector:cuda() end

		local bingo = false -- it is a right trigger action or not
		local action = 0 -- init action
		local step_count = 0 -- reset step_count
		reward = 0 -- re-init reward
		--while (not bingo) and (step_count < max_steps) and not_finished
		while (step_count < max_steps) and not_finished
		do
			log_file:write('\t\t\tStep: ' .. step_count .. ' ---> Action= ' .. action ..
						' ; Mask= [' .. cur_mask[1] .. ' , ' .. cur_mask[2] ..
						' ]; GT = [' .. now_target_gt[1] .. ' , ' .. now_target_gt[2] ..
						' ]; Reward= ' .. reward .. ' ; iou = ' .. new_iou .. '; overlap = '
						.. overlap .. '\n')
			print('\t\t\tStep: ' .. step_count .. ' ---> Action= ' .. action ..
						' ; Mask= [' .. cur_mask[1] .. ' , ' .. cur_mask[2] ..
						' ]; GT = [' .. now_target_gt[1] .. ' , ' .. now_target_gt[2] ..
						' ]; Reward= ' .. reward .. ' ; iou = ' .. new_iou .. '; overlapt = '
						.. overlap .. '\n')
			-- run DQN
			local action_output = dqn:forward(input_vector)
			print(action_output)
```

```
                local tmp_flag = 0
                local trigger_memory = {}

                -- It is checking for last non-trigger action, which may actually lead to an
                -- terminal state; we force it to be terminal action in case actual IoU
                -- is higher than 0.5, to train faster the agent;
                local tmp_v = 0
                tmp_v, action = torch.max(action_output,1)
                action = action[1]-- from tensor to numeric type

                if (cur_mask[2]-cur_mask[1]+1) >= max_gt_length*2 and (action == 4 or action == 5) then
                        -- forbid expand than max_gt_length
                        -- choose a random action
                        action = torch.random(torch.Generator(),1,3)
                elseif (cur_mask[2]-cur_mask[1]) <= 16 and action == 3 then
                        action = torch.random(torch.Generator(),1,4)
                        if action == 3 then action = 5 end
                end
                if action == trigger_action then
                        tmp_flag = 1
                elseif i < max_epochs and new_iou > trigger_thd then
                        --action = trigger_action
                        tmp_flag = 2
                elseif i < max_epochs and new_iou == 0 then
                        action = jump_action
                elseif torch.uniform(torch.Generator()) < epsilon then -- greedy policy
                        action = torch.random(torch.Generator(),1,number_of_actions)
                end

                local localize_reg = torch.Tensor(2):fill(0)

                if action == trigger_action then -- estemated as trigger
                        old_iou, new_iou, iou_table, index = func_follow_iou(cur_mask,
        tmp_gt, available_objects, iou_table)
                        overlap = func_calculate_overlapping(tmp_gt[index], cur_mask)

                        now_target_gt = tmp_gt[index]

                        reward = func_get_reward_trigger(new_iou)

                        if reward > 0 then
                                localize_reg[1] = (now_target_gt[1]-cur_mask[1])/(cur_mask[2]-
cur_mask[1]+1)
                                localize_reg[2] = (now_target_gt[2]-cur_mask[2])/(cur_mask[2]-
cur_mask[1]+1)
                        end

                        step_count = step_count+1
                        bingo = true

                        log_file:write('\t\t\tStep: ' .. step_count .. ' ---> Action= ' .. action ..
                                ' ; Mask= [' .. cur_mask[1] .. ' , ' .. cur_mask[2] ..
                                ' ]; GT = [' .. now_target_gt[1] .. ' , ' .. now_target_gt[2] ..
```

```
                                                                ' ]; Reward= ' .. reward .. ' ; iou = ' .. new_iou .. '; overlap = '
                                                                .. overlap .. '; self = '.. tmp_flag .. '\n')
                                                        print('\t\t\tStep: ' .. step_count .. ' ---> Action= ' .. action ..
                                                                ' ; Mask= [' .. cur_mask[1] .. ' , ' .. cur_mask[2] ..
                                                                ' ]; GT = [' .. now_target_gt[1] .. ' , ' .. now_target_gt[2] ..
                                                                ' ]; Reward= ' .. reward .. ' ; iou = ' .. new_iou .. '; overlap = '
                                                                .. overlap .. '; self = '.. tmp_flag ..'\n')
                                                        trigger_memory[1] = {input_vector, number_of_actions, reward, input_vector,
localize_reg}
                                                        action = torch.random(torch.Generator(),1,number_of_actions-1)
                                                        --*****************************
                                                        --step_count = max_steps -- to jump out of the loop
                                                        --*****************************
                                                elseif tmp_flag == 2 then
                                                        -- forced trigger
                                                        old_iou, new_iou, iou_table, index = func_follow_iou(cur_mask,
                                                                                                                                tmp_gt, available_objects, iou_table)
                                                        overlap = func_calculate_overlapping(tmp_gt[index], cur_mask)

                                                        now_target_gt = tmp_gt[index]

                                                        reward = func_get_reward_trigger(new_iou)

                                                        if reward > 0 then
                                                                localize_reg[1] = (now_target_gt[1]-cur_mask[1])/(cur_mask[2]-
cur_mask[1]+1)
                                                                localize_reg[2] = (now_target_gt[2]-cur_mask[2])/(cur_mask[2]-
cur_mask[1]+1)
                                                        end

                                                        step_count = step_count+1
                                                        
                                                        log_file:write('\t\t\tStep: ' .. step_count .. ' ---> Action= ' .. number_of_actions ..
                                                                ' ; Mask= [' .. cur_mask[1] .. ' , ' .. cur_mask[2] ..
                                                                ' ]; GT = [' .. now_target_gt[1] .. ' , ' .. now_target_gt[2] ..
                                                                ' ]; Reward= ' .. reward .. ' ; iou = ' .. new_iou .. '; overlap = '
                                                                .. overlap .. '; self = '.. tmp_flag .. '\n')
                                                        print('\t\t\tStep: ' .. step_count .. ' ---> Action= ' .. number_of_actions ..
                                                                ' ; Mask= [' .. cur_mask[1] .. ' , ' .. cur_mask[2] ..
                                                                ' ]; GT = [' .. now_target_gt[1] .. ' , ' .. now_target_gt[2] ..
                                                                ' ]; Reward= ' .. reward .. ' ; iou = ' .. new_iou .. '; overlap = '
                                                                .. overlap .. '; self = '.. tmp_flag ..'\n')
                                                        
                                                        -- add to memory
                                                        trigger_memory[1] = {input_vector, number_of_actions, reward, input_vector,
localize_reg}
                                                end
                                                if action == jump_action then
                                                        -- encourage jump action if it is a iou==0 state
                                                        if new_iou <= 0.05 then
                                                                reward = func_get_reward_movement(0, 1,0,0)*0.5 -- half reward
                                                        else
                                                                reward = func_get_reward_movement(1,0,0,0)*(5)
```

```
                    end
                    cur_mask = func_take_advance_action(cur_mask, action, total_frms, act_alpha, tmp_gt)

                    old_iou, new_iou, iou_table, index = func_follow_iou(cur_mask,

    tmp_gt, available_objects, iou_table)
                    overlap = func_calculate_overlapping(tmp_gt[index], cur_mask)
                    now_target_gt = tmp_gt[index]
                    old_iou = new_iou

                    history_vector = func_update_history_vector(history_vector, action)
                    step_count = step_count + 1
                elseif action ~= trigger_action then -- take action
                -- 1 move forward; 2 move back; 3 narrow; 4 left_expand; 5 right_expand
                    cur_mask = func_take_advance_action(cur_mask, action, total_frms, act_alpha,tmp_gt)

                    old_iou, new_iou, iou_table, index = func_follow_iou(cur_mask,

    tmp_gt, available_objects, iou_table)
                    overlap = func_calculate_overlapping(tmp_gt[index], cur_mask)
                    now_target_gt = tmp_gt[index]
                    reward = func_get_reward_movement(old_iou, new_iou,0,0)
                    old_iou = new_iou

                    history_vector = func_update_history_vector(history_vector, action)
                    step_count = step_count + 1
                    -- log wiil be written at the beginning of the next loop
                end
                --local C3D_vector = func_get_C3D(opt.data_path, opt.class, 1,
                --                                                        v, cur_mask[1],
cur_mask[2], C3D_m, {}, 27, fc6)
                local C3D_vector = func_get_C3D(clip_img[{ {cur_mask[1], cur_mask[2]},{} }], C3D_m,{},27,fc6)
                local new_input_vector = torch.cat(C3D_vector, history_vector, 1)
                if opt.gpu >=0 then new_input_vector = new_input_vector:cuda() end

                count_train[1] = count_train[1]+1
                -- experience replay
                local tmp_experience  = {input_vector, action, reward, new_input_vector, localize_reg}
                if table.getn(replay_memory) < experience_replay_buffer_size then
                    table.insert(replay_memory, tmp_experience)
                    if #trigger_memory > 0 then
                        if #trigger_memory == 1 then
                            table.insert(replay_memory, trigger_memory[1])
                        else
                            error('wrong trigger memory')
                        end
                    end
                    input_vector = new_input_vector
                else
                    -- replay_memory is a stack
                    table.remove(replay_memory, 1)
                    table.insert(replay_memory, tmp_experience)
                    if #trigger_memory > 0 then
                        table.remove(replay_memory, 1)
```

```
                table.insert(replay_memory, trigger_memory[1])
            end

            local tmp_mod = torch.fmod(count_train,train_period)
            tmp_mod = tmp_mod[1]
            if tmp_mod == 0 then

                local minibatch = func_sample(replay_memory, batch_size) -- in Hjj_Reinforcement
                local memory = {}
                -- construct training set
                local training_set = {data=torch.Tensor(batch_size, input_vector_size), label=torch.Tensor(batch_size, number_of_actions)}
                function training_set:size() return batch_size end
                setmetatable(training_set, {__index = function(t,i)
                    return {t.data[i], t.label[i]} end})

                if opt.gpu >= 0 then
                    training_set.data = training_set.data:cuda()
                    training_set.label = training_set.label:cuda()
                end

                -- construct training set for rgn
                local count_batch = 0
                for l,memory in pairs(minibatch)
                do
                    local tmp_action = memory[2]
                    local tmp_reward = memory[3]
                    if tmp_action == trigger_action and tmp_reward > 0 then
                        count_batch = count_batch+1
                    end
                end
                local training_set_rgn = {}
                if count_batch > 0 then
                    training_set_rgn = {data=torch.Tensor(count_batch, input_vector_size), label=torch.Tensor(count_batch, 2)}
                    function training_set_rgn:size() return count_batch end
                    setmetatable(training_set_rgn, {__index = function(t,i)
                        return {t.data[i], t.label[i]} end})
                end
                if opt.gpu >= 0 then
                    training_set_rgn.data = training_set_rgn.data:cuda()
                    training_set_rgn.label = training_set_rgn.label:cuda()
                end

                log_file:write('\t\t\t\t Doing memory replay...\n')
                print('\t\t\t\t Doing memory replay...\n')
                local counter_batch = 1
                for l, memory in pairs(minibatch)
```

```
                                                    do
                                                        local tmp_input_vector = memory[1]
                                                        local tmp_action = memory[2]
                                                        local tmp_reward = memory[3]
                                                        local tmp_new_input_vector = memory[4]
                                                        if tmp_action == trigger_action and tmp_reward > 0 then
                                                            if (count_batch-counter_batch < 0) then error('rgn
data set error') end
                                                            training_set_rgn.data[counter_batch] =
tmp_input_vector
                                                            training_set_rgn.label[counter_batch] = memory[5]
                                                            counter_batch = counter_batch + 1
                                                        end
                                                        local old_action_output = dqn:forward(tmp_input_vector)
                                                        local new_action_output = dqn:forward(tmp_new_input_vector)
                                                        local tmp_v = 0
                                                        local tmp_index = 0
                                                        local y = old_action_output:clone()
                                                        tmp_v, tmp_index = torch.max(new_action_output, 1)
                                                        tmp_v = tmp_v[1]
                                                        tmp_index = tmp_index[1]
                                                        local update_reward = 0
                                                        if (tmp_action == trigger_action) or (tmp_action == jump_action)
then
                                                            update_reward = tmp_reward
                                                        else
                                                            update_reward = tmp_reward + gamma * tmp_v
                                                        end
                                                        y[tmp_action] = update_reward
                                                        training_set.data[l] = tmp_input_vector
                                                        training_set.label[l] = y
                                                    end
                                                    -- training
                                                    log_file:write('\t\t\t\t Training...\n')
                                                    print('\t\t\t\t Training...\n')
                                                    local function feval(x)
                        if x ~= params then
                            params:copy(x)
                        end
                                                        gradParams:zero()
                                                        --print(params:sum())

                                                        local outputs = dqn:forward(training_set.data)
                        --print(outputs:sum())
                                                        local loss = criterion:forward(outputs, training_set.label)
                                                        local dloss_doutputs = criterion:backward(outputs,
training_set.label)
                                                        --print(dloss_doutputs:sum())
                        --io.read()
                        dqn:backward(training_set.data, dloss_doutputs)
                                                        logger:add{loss*100, i}
                                                        return loss, gradParams
                                                    end
                                                    optim.sgd(feval, params, optimState)
```

```
                                                    --training rgn
                                                    if count_batch > 0 then
                                                        log_file:write('\t\t\t\t Training RGN...\n')
                                                        print('\t\t\t\t Training RGN...\n')
                                                        local function feval_rgn(x)
                                if x ~= params_rgn then
                                    params_rgn:copy(x)
                                end
                                                        gradParams_rgn:zero()
                                                        --print(params:sum())

                                                        local outputs = rgn:forward(training_set_rgn.data)
                                --print(outputs:sum())
                                                        local loss = criterion_rgn:forward(outputs,
training_set_rgn.label)
                                                        local dloss_doutputs =
criterion_rgn:backward(outputs, training_set_rgn.label)
                                                        --print(dloss_doutputs:sum())
                                --io.read()
                                rgn:backward(training_set_rgn.data, dloss_doutputs)
                                                        logger_rgn:add{loss}
                                                        return loss, gradParams_rgn
                                                    end
                                                    optim.sgd(feval_rgn, params_rgn, optimState_rgn)

                                                    local tmp_a = torch.Tensor(1)
                                                    local tmp_b = 1000
                                                    tmp_a[1] = optimState_rgn.evalCounter
                                                    local tmp_mod = torch.fmod(tmp_a,tmp_b)
                                                    tmp_mod = tmp_mod[1]
                                                        -- save enviroments
                                                        if tmp_mod == 0 or optimState_rgn.evalCounter == 1
then
                                                            local mdl_name={}
                                                            if opt.gpu >= 0 then
                                                                mdl_name =
'./model/rgn_pooling_'.. opt.name .. opt.class .. '_'.. optimState_rgn.evalCounter
                                                            else
                                                                mdl_name = './model/c_'..
opt.name .. opt.class .. '_'.. i
                                                            end
                                                            torch.save(mdl_name, {rgn = rgn})
                                                        end
                                                end
                                            end -- mod
                                            input_vector = new_input_vector
                                        end -- if memory replay

                                        if action == trigger_action then
                                            bingo = true
                                            masked = true
                                            if reward == 3 then
```

```
                                                table.insert(masked_segs, {cur_mask[1]+torch.floor((cur_mask[2]-
cur_mask[1]+1)*0.1),
                                                                                cur_mask[2]-torch.floor((cur_mask[2]-
cur_mask[1]+1)*0.1)})
                                        end
                                    else
                                        masked = false
                                    end
                                end -- while (not bingo) and (step_count < max_steps) and not_finished
                                -- available_objects[index] = 0
                            end -- gts loop

                        end -- clips loop
                        if epsilon > 0.1 then
                            epsilon = epsilon - 0.1
                        end
                        -- save enviroments
                        if table.getn(replay_memory) >= experience_replay_buffer_size then
                            local mdl_name={}
                            if opt.gpu >= 0 then
                                mdl_name = './model/g_'.. opt.name .. opt.class .. '_'.. i
                            else
                                mdl_name = './model/c_'.. opt.name .. opt.class .. '_'.. i
                            end
                            torch.save(mdl_name, {dqn = dqn, gpu = opt.gpu})
                        end

end -- epochs loop

log_file:close()
```

The validate_SAP code is as follows:

For following the code structure, one can look at if conditions and loops. They are explained properly using comments and tthe process is easy to understand. It'll be easier to copy the code in a text editor and save the file as *.lua for reading.

```
require 'Read_Input_Cmd'
require 'Reinforcement3'
require 'Mask_and_Actions'
require 'Metrics'
require 'Interface_PyramidPooling'

local cmd = torch.CmdLine()
opt = func_read_validate_rgn_cmd(cmd, arg)

-- create log file
local log_file = io.open(opt.log_log, 'w')
if not log_file then
        print("open log file error")
        error("open log file error")
end
```

```
local name  = './data_output/track'.. opt.name .. '.txt'
local track_file = io.open(name, 'w')
if not track_file then
        print("open track file error")
        error("open track file error")
end

name  = './data_output/gt'.. opt.name .. '.txt'
local gt_file = io.open(name, 'w')
if not gt_file then
        print("open gt file error")
        error("open gt file error")
end

local training_file = './' .. opt.data_path .. '/trainlist.t7'
local clip_table = torch.load(training_file)
local tt = clip_table[opt.class]
if tt == nil then
        error('no trainlist file')
end

-- read validate clips from files
local validate_file = './' .. opt.data_path .. '/validatelist.t7'

print(validate_file)
local clip_table = torch.load(validate_file)
local validate_clip_table = clip_table[opt.class]
if validate_clip_table == nil then
        error('no trainlist file')
end

-- action parameters
local max_steps = opt.max_steps
local trigger_thd = 0.65 -- threshold for terminal
local trigger_action = number_of_actions
local act_alpha = opt.alpha
local max_trigger = 11
local mask_rate = 0.05 -- 1-2*mask_rate of current mask will not be used anymore

-- number_of_actions and history_action_buffer_size are globle variables in Reinforcement
local history_vector_size = number_of_actions * history_action_buffer_size
local input_vector_size = history_vector_size + C3D_size

-- load dqn
if opt.model_name == '0' then
        error('model needed')
end
local dqn={}
local ldldl
dqn= func_get_dqn(opt.model_name, log_file)
dqn:evaluate()
local fc6 = torch.load('fc6.t7')
fc6:evaluate()
local rgn = func_get_rgn(opt.rgn_name, log_file)
```

```
rgn:evaluate()

-- set gpu
opt.gpu = func_set_gpu(opt.gpu, log_file)
if opt.gpu >=0 then
        dqn = dqn:cuda()
        rgn = rgn:cuda()
        fc6 = fc6:cuda()
end

local gt_table = func_get_data_set_info(opt.data_path, opt.class, 1)

-- get average length
local count = 0
local len = 0
for i,v in pairs(tt) do
        local tmp_table = gt_table[v]
        for j,u in pairs(tmp_table) do
                count = count +1
                len = len + u[2]-u[1]
        end
end
local avg_len = torch.floor(len/count)
if avg_len < 16 then avg=16 end
print(avg_len)

local max_gt_length = 128 -- max length to split gt

-- load C3D model
local C3D_m = torch.load('c3d.t7');
C3D_m:evaluate()

-- used to visualize the action sequence
local iou_record_table = {}
local gt_ind_record_table = {}
local mask_record_table= {}
local max_steps = 15 -- prevent from being trapped in local pit

for i,v in pairs(validate_clip_table)
do
        --if i>1 then break end

                local tmp_gt = gt_table[v]

                for c=1,#tmp_gt do
                        gt_file:write(i .. '\t' .. tmp_gt[c][1] .. '\t' .. tmp_gt[c][2] .. '\n')
                end

                local total_frms = tmp_gt[1][3]
                print('load images')
                local clip_img = func_load_clip(opt.data_path, opt.class, 1, v,total_frms)
                local masked_segs={}
                local gt_num = table.getn(tmp_gt)
                local steps = 0
```

```
                log_file:write('\tIt is the ' .. i .. ' clip, clip_id = ' ..
                                                                v .. ' total_frms = '.. total_frms .. '\n')
                print('\tIt is the '.. i .. ' clip, clip_id = ' ..
                                                                v .. ' total_frms = '.. total_frms)
                local lp=1
                local left_frm = total_frms
                local knocked = 0
                local start_wall = 0
                local last_f = 1

                while (left_frm > 16) and (knocked < 5)
                do
                        local iou = 0
                        local index = 0
                        --local mask = func_mask_random_init(total_frms, masked_segs)
                        -- continue from the end of last mask
                        local mask = {last_f, last_f+avg_len}
                        if last_f - 16 > 0 then
                                mask[1] = mask[1] - 16
                                mask[2] = mask[2] - 16
                         end

                        if mask[2] >= total_frms then
                                mask[2] = total_frms
                                knocked = knocked + 1
                        end
                        if mask[1] == 1 then start_wall = start_wall + 1 end

                        local history_vector = torch.Tensor(history_vector_size):fill(0)
                        local bingo = false
                        local action = 0
                        local last_action = 0
                        local step_count = 0
                        local output_record = 0
                        while (left_frm > 16) and (knocked < 5) and (not bingo) and (step_count < max_steps)
                        do
                                iou, index = func_find_max_iou(mask, tmp_gt)
                                if not(action == 0) then
                                        track_file:write(i .. '\t' .. lp .. '\t' .. steps .. '\t' ..
                                                                iou .. '\t' .. mask[1] .. '\t' .. mask[2] .. '\t' ..
action ..'\t'.. output_record[trigger_action] .. '\t' .. v .. '\n')
                                end
                                print('\t\tstep ' .. steps .. '\t; beg = ' .. mask[1] .. '\t ;end = ' .. mask[2]
                                                        .. ' ; iou ' .. iou .. '\t' .. action ..'\t'.. action-action .. '\t' ..
v .. '\n')

                                --local C3D_vector = func_get_C3D(opt.data_path, opt.class, 1,
                                --                                      v, mask[1],
mask[2], C3D_m, {}, 27, fc6)
                                local C3D_vector = func_get_C3D(clip_img[{ {mask[1], mask[2]},{} }], C3D_m,{},27,fc6)
                                local input_vector = torch.cat(C3D_vector, history_vector, 1)

                                if opt.gpu >=0 then input_vector = input_vector:cuda() end
```

```
                local action_output = dqn:forward(input_vector)
                local tmp_v = 0
                output_record = action_output:clone()
                tmp_v, action = torch.max(action_output,1)
                action = action[1]-- from tensor to numeric type
                -- give a very small number for getting the second max action
                action_output[action] = -111111111

                print('\t\t\tAction = ' .. action .. '\n')


                if action == 3 and mask[2]-mask[1] <= 16 then
                        tmp_v, action = torch.max(action_output,1)
                        action = action[1]-- from tensor to numeric type
                elseif (action == 4 or action == 5) and mask[2]-mask[1]+1 >= 2*max_gt_length then
                        tmp_v, action = torch.max(action_output,1)
                        action = action[1]-- from tensor to numeric type
                        if action == 4 or action == 5 then
                                action_output[action] = -111111111
                                tmp_v, action = torch.max(action_output,1)
                                action = action[1]-- from tensor to numeric type
                        end

                end
                if action == trigger_action then
                        print('############### BOOM! ############'.. mask[1] ..
                                ' - ' .. mask[2] .. ' ; ' .. total_frms .. '\n')

                        -- modify with rgn
                        local localize_reg = rgn:forward(input_vector)
                        local new_1 = mask[1] + torch.floor(localize_reg[1]*(mask[2]-mask[1]+1))
                        local new_2 = mask[2] + torch.floor(localize_reg[2]*(mask[2]-mask[1]+1))
                        print('Move frome ( '.. mask[1] .. ', '.. mask[2] ..' ) to ( '.. new_1 .. ', '
                                .. new_2 .. ' )\n')
                        if new_1 <= 0 then new_1 = 1 end
                        if new_2 >= total_frms then new_2 = total_frms end
                        if (new_2 - new_1) > 0 then
                                mask[1] = new_1
                                mask[2] = new_2
                        end
                        track_file:write(i .. '\t' .. lp .. '\t' .. steps .. '\t' ..
                                                iou .. '\t' .. mask[1] .. '\t' .. mask[2] .. '\t' .. action .. '\
t' .. output_record[trigger_action] .. '\t' .. v ..'\n')
                        bingo = true
                        -- not to mask all the area
                        if last_f < mask[2] then
                                last_f = mask[2]
                                left_frm = total_frms - last_f
                        end
                        if mask[2] == total_frms then
                                knocked = knocked + 1
                        end
                        mask[1] = mask[1]+torch.floor((mask[2]-mask[1])*mask_rate)
```

```
                                                mask[2] = mask[2]-torch.floor((mask[2]-mask[1])*mask_rate)
                                                table.insert(masked_segs, mask)
                                        else
                                                local rand = 0
                                                mask = func_take_advance_action_forward(mask, action, total_frms, act_alpha*(1+rand))
                                                if mask[1] == 1 then
                                                        start_wall = start_wall + 1
                                                else
                                                        start_wall = 0
                                                end
                                                if last_f < mask[2] then
                                                        last_f = mask[2]
                                                        left_frm = total_frms - last_f
                                                elseif last_f - mask[2] >= 64 or start_wall >= 5 then
                                                        bingo = true
                                                        print('~~~~~~go back too much!!!!')
                                                end
                                                if mask[2] == total_frms then
                                                        knocked = knocked + 1
                                                end
                                        end
                                        history_vector = func_update_history_vector(history_vector, action)
                                        steps = steps + 1
                                        step_count = step_count + 1
                                end             --while
                                lp = lp+1
                        end -- while

end --for clips

gt_file:close()
track_file:close()
```

Rest of the training and validation files have slight changes in each, but the basic structure is the same.

Results Obtained

The hyperparameters taken are same as given in the paper except for dropout probability which was changed to 0.3 as it gave less validation error for most classes. The results obtained are comparable to the research paper and even better in some cases. They are given in the next section.

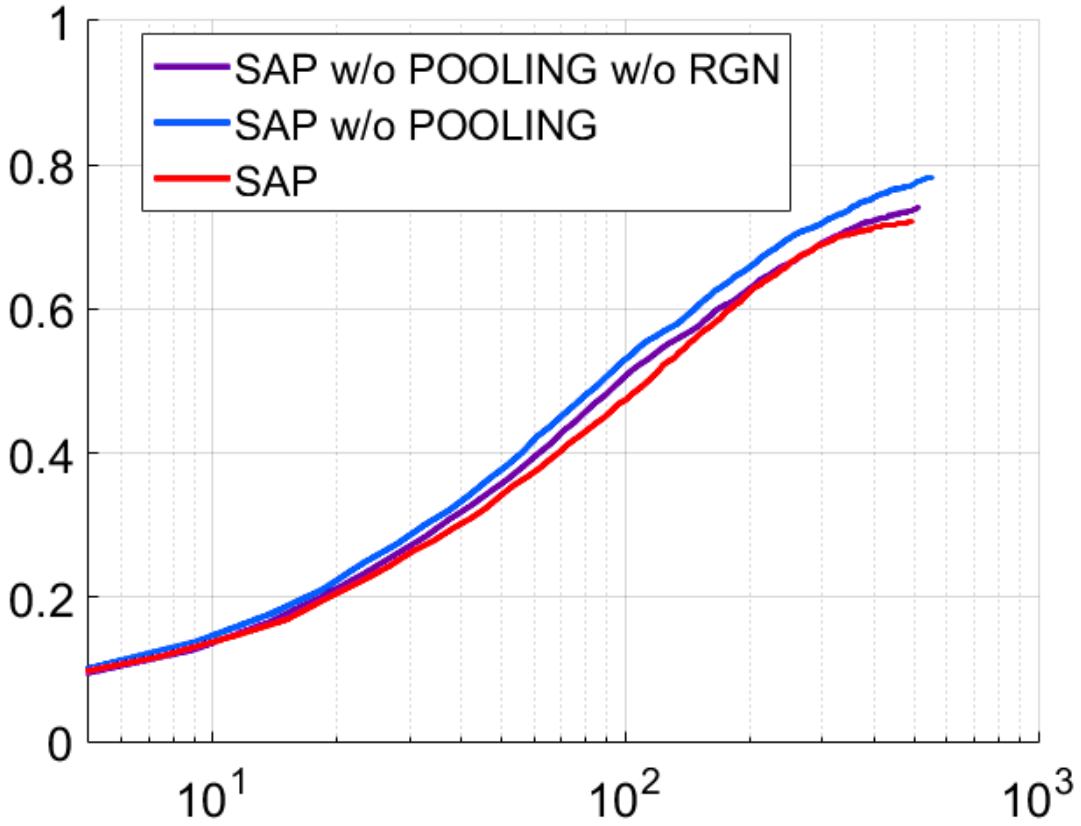


Fig6: Graph of Recall v/s Average number of proposals at IoU=0.5

The mAP% of 27.7 is obtained at NA=67 approximately (refer to the table below) and droupout =0.2 for SAP. For dropout = 0.3, mAP obtained is 27.9 which is better than what the paper states. MAP stands for Mean Average Precision.

Model@Proposal Number	mAP(%)
DAPs+TSN @50	6.99
DAPs+TSN @100	9.16
SCNN-prop+TSN @100	12.4
SCNN-prop+TSN @50	16.9
SAP w/o POOLING w/o RGN@50	22.3
SAP w/o POOLING@50	24.6
SAP@50	**26.4**
Oneata *et al.*@NA	14.4
Yeung *et al.*@NA	17.1
SCNN @NA	19.0
CDC @NA	23.3
SAP@NA	**27.7**

Our Model@67, dropout=0.3 27.9

Table: Temporal-action detection results evaluation: mAP calculated with a fixed number of average proposals on THUMOS'14. @NA means the proposal number is not specified for the methods.

The two metrics clash as recall vs number of proposals graph shows that SAP without pooling is better than SAP, but SAP gets higher mAP. This maybe because the SAP model produces low number of false positives. Besides, the results also illustrate that localization regression is of benefit to the detection task without exception.

The run-time property of our method is dependent on the DQN's performance. For a well trained DQN agent, it will concentrate on the ground truth in a couple of steps once it perceives the action segment. Meanwhile, it can also accelerate the exploring process over the video with "jump" action. Besides, the selection of scalar α is also an important factor that will influence the run-time performance. A large α will make the agent take a brief glance over the video in most of the case, but will also result in coarse proposals. As a trade off,we set the $\alpha = 0.2$ during the training and testing phase. During the test, we load the video first and then record the run-time of our model used to scan the video. On GTX 1060, the average run-time over all testing videos in THUMOS'14 is around 20 FPS, including the feature extraction.

Conclusion

The paper was implemented and slightly improved by tuning hyperparameters of the model (dropout probability). They have introduced an active action proposal model called SAP that learns to adaptively adjust the span of attended current window to cover the true action regions in a few steps. They build our model based on deep reinforcement learning and lean an optimal policy to direct the agent to act. In order to precisely locate the action, they design a regression network to revise the offsets between predicted bound results and the groundtruth. Experiment results on THUMOS'14 dataset validate that the proposed approach can achieve comparable performance with most of modern action-detection methods with much fewer action proposals.