Chapter 1

Tube Proposal Network

1.1 Our implementation's architecture

In this chapter, we get involved with Tube Proposal Network(TPN), one of the basic elements of ActionNet. Before describing it, we present the whole structure of our model. We propose a network similar to [?]. Our architecture is consisted by the following basic elements:

- One 3D Convolutional Network, which is used for feature extraction. In our implementaion we use a 3D Resnet network which is taken from [?] and it is based on ResNet CNNs for Image Classification [?].
- Tube Proposal Network for proposing action tubes (based on the idea presented in [?]).
- A classifier for classifying video tubes.

The basic procedure ActionNet follows is:

1. Given a video, we seperate it into video segments. These video segments in some cases overlap temporally and in some others don't.
2. For each video segment, after performing spatiotemporal resizing, we feed its frames into ResNet34 in order to perform feature extraction. These activation maps are, next, fed into TPN for proposing sequences of bounding boxes. We name them Tubes of Interest (ToIs), like [?], and are likely to contain a person performing an action. N
3. After getting proposed ToIs for each video segment, using a linking algorithm, ActionNet finds final cancidate action tubes. These action tubes are given as input to a classifier in order to get their action class.

A diagram of ActionNet is shown at Figure 1.1.

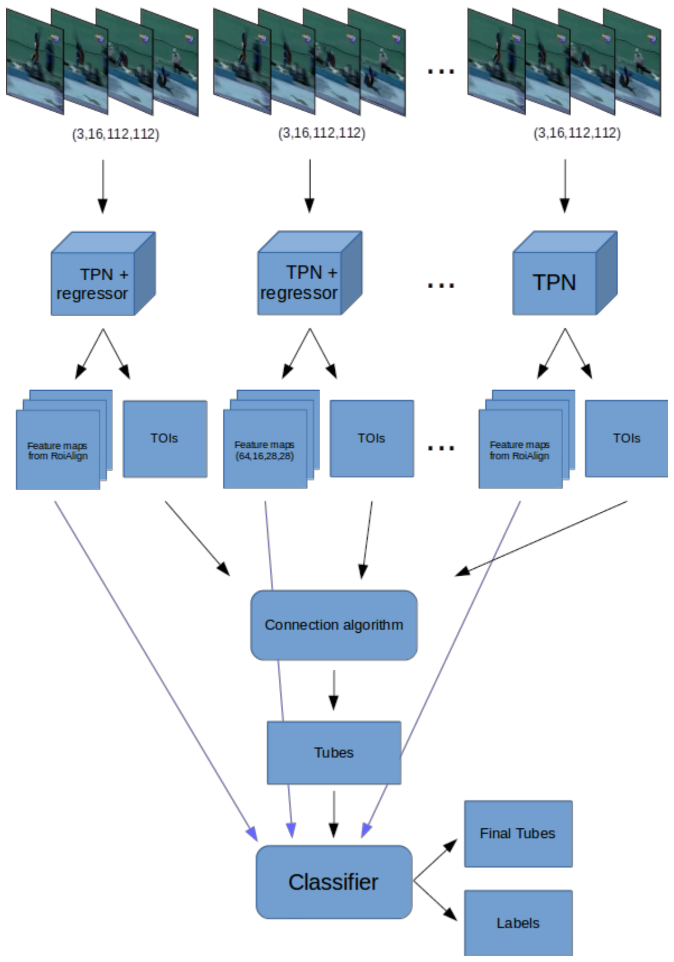

Figure 1.1: Structure of the whole network

1.2 Introduction to TPN

The main purpose of Tube Proposal Network (TPN) is to propose **Tube of Interest**(TOIs). These tubes are likely to contain an known action and are consisted of some 2D boxes (1 for each frame). TPN is inspired from RPN introduced by FasterRCNN ([**?**]), but instead of images, TPN is used in videos as show in [**?**]. In full correspondence with RPN, the structure of TPN is similar to RPN. The only difference, is that TPN uses 3D Convolutional Layers and 3D anchors instead of 2D.

We designed 2 main structures for TPN. Each approach has a different definition of the used 3D anchors. The rest structure of the TPN is mainly the same with some little differences in the regression layer.

1.3 Preparation for TPN

1.3.1 Preparing data

Before getting a video as input to extract its features and ToIs, this video has to be preprocessed. Preprocess procedure is the same for both approaches of TPN. Our architecture gets as input a sequnece of frames which has a fixed size in widht, height and duration. However, each video has different resolution. That's creates the need to resize each frame before. As mentioned in previous chapter, the first element of our network is a 3D RenNet taken from [**?**]. This network is designed to get images with dimensions (112,112). As a result, we resize each frame from datasets' videos into (112,112) frames. In order to keep aspect ratio, we pad each frame either left and right, either above and bellow depending which dimension is bigger. In figure 1.2 we can see the original frame and the resize and padded one. In full correspondance, we resize the groundtruth bounding boxes for each frame (figure 1.2b and 1.2d show that).

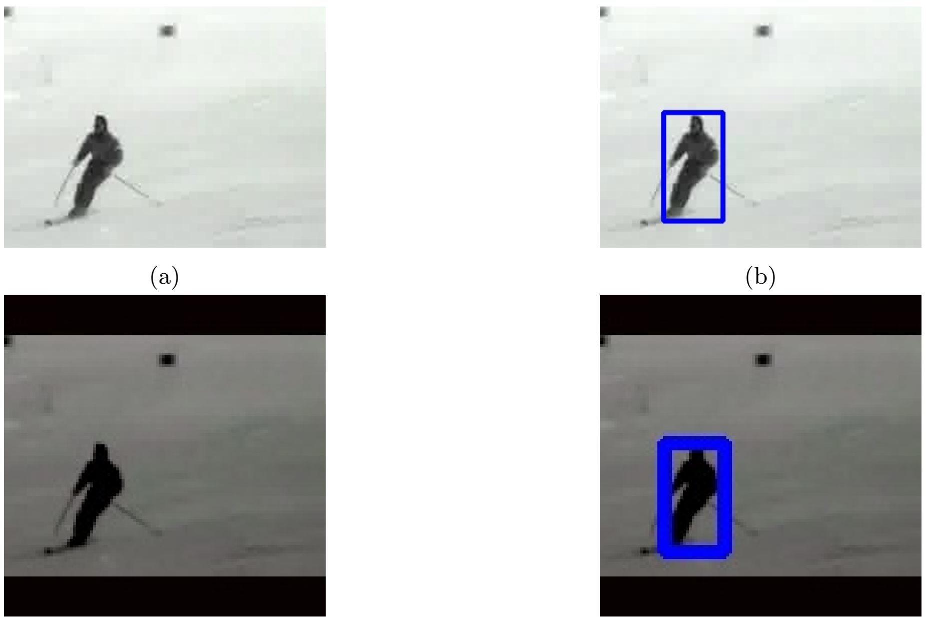
(a) (b) (c) (d)

Figure 1.2: At (a), (b) frame is its original size and at (c), (d) same frame after preprocessing part

1.3.2 3D ResNet

Before using Tube Proposal Network, we spatio-temporal features from the video. In order to do so, we extract the 3 first Layers of a pretrained 3D ResNet.

It is pretrained in Kinetics dataset [?] for sample duration = 16 and sample size = (112,122).

This network normally is used for classifying the whole video, so some of its layers use temporal stride = 2. We set their temporal stride equal to 1 because we don't want to miss any temporal information during the process. So, the output of the third layer is a feature maps with dimesions (256,16,7,7). We feed this feature map to TPN, which is described in following sections.

1.4 3D anchors as 6-dim vector

1.4.1 First Description

We started desinging our TPN inspired by [?]. We consider each anchor as a 3D bounding box written as $(x_1, y_1, t_1, x_2, y_2, t_2)$ where x_1, y_1, t_1 are the upper front left coordinates of the 3D and x_2, y_2, t_2 are the lower back left as shown in figure 1.3.

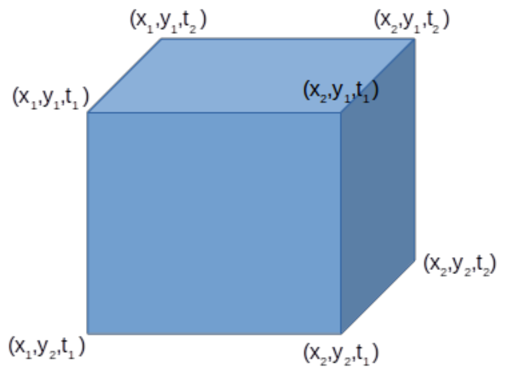

Figure 1.3: An example of the anchor $(x_1, y_1, t_1, x_2, y_2, t_2)$

The main advantage of this approach is that except from x-y dims, dimension of time is mutable. As a result, the proposed TOIs have no fixed time duration. This will help us deal with untrimmed videos, because proposed TOIs would exclude background frames. For this approach, we use **n = 4k = 60** anchors for each pixel in the feature map of TPN. We have k anchors for each sample duration(5 scales of 1, 2, 4, 8, 16, 3 aspect ratios of 1:1, 1:2, 2:1 and 4 durations of 16,12,8,4 frames). In [?], network's anchors are defined according to the dataset most common anchors. This, however, creates the need to redesign the network for each dataset. In our approach, we use the same anchors for both datasets, because we want our network not to be dataset-specific but to be able to generalize for several datasets. As sample duration, we chose 16 frames per video segment because our pre-trained ResNet is trained for video clips with that duration. So the structure of TPN is:

- 1 3D Convolutional Layer with kernel size = 3, stride = 3 and padding = 1
- 1 classification layer outputs *2n scores* whether there is an action or not for *n tubes*.
- 1 regression layer outputs *6n coordinates* $(x_1, y_1, t_1, x_2, y_2, t_2)$ for *n tubes*.

which is shown in figure 1.4

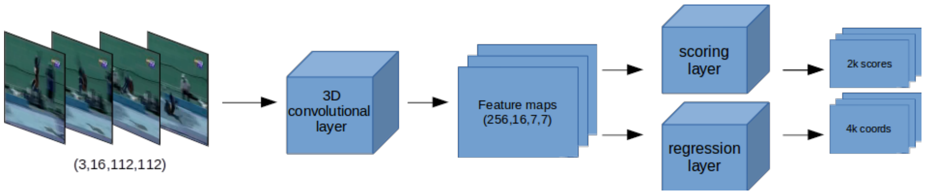

Figure 1.4: Structure of TPN

The output of TPN is the k-best scoring cuboid, in which it is likely to contain an action.

1.4.2 Training

As mentioned before, TPN extracts TOIs as 6-dim vectors. For that reason, we modify out groundtruth ROIs to groundtruth Tubes. We take for granted that the actor cannot move a lot during 16 frames, so that's why we use this kind of tubes. As shown in figure 1.5, these tubes are 3D boxes which include all the groundtruth rois, which are different for each frame.

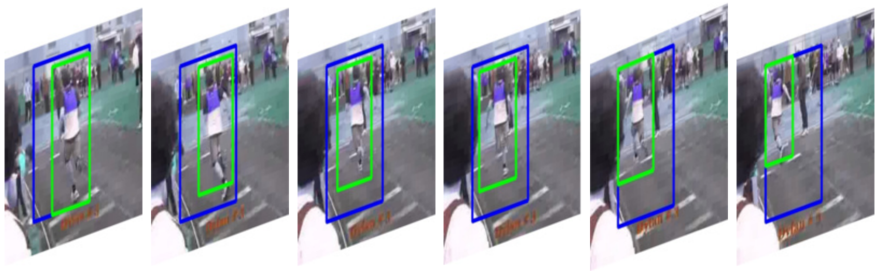

Figure 1.5: Groundtruth tube is coloured with blue and groundtruth rois with colour green

For training procedure, for each video, we randomly select a part of it which has duration 16 frames. We consider an anchor as foreground if its overlap score with a groundtruth action tube is bigger than 0.5. Otherwise, it is considered as background anchor. We use scoring layer in order to correctly classify those anchors and we use Cronss Entropy Loss as loss function. We have a lot of anchors for proposing an anction but few numbers of actions, so we choose 256 anchors in total for each batch. We set the maximum number of foreground anchors to be 25% of the 256 anchors and the rest are the background.

Classifying correctly an anchor isn't enough for proposing an action tube. It is necessary to overlap as much as possible with the groundtruth action tubes. That's the reason we use a regression layer. This layer "moves" the cuboid closer to the area that it is believed that is closer to the action. For regression loss we use smooth-L1 loss as proposed in [**?**]. In order to calculate the regression targets, we use pytorch FasterRCNN implementation ([**?**]) for bounding box regression and we modified the code in order to extend it for 3 dimensions. So we have:

$$\begin{array}{ccc} t_x = (x - x_a)/w_a, & t_y = (y - y_a)/h_a, & t_z = (z - z_a)/d_a, \\ t_w = log(w/w_a), & t_h = log(h/h_a), & t_d = log(d/d_a), \\ t_x^* = (x^* - x_a)/w_a, & t_y^* = (y^* - y_a)/h_a, & t_z^* = (z^* - z_a)/d_a, \\ t_w^* = log(w^*/w_a), & t_h^* = log(h^*/h_a), & t_d^* = log(d^*/d_a), \end{array}$$

where *x, y, z, w, h, d* denote the 3D box's center coordinates and its width, height and duration. Variables $x, x_a,$ and x^* are for the predicted box, anchor box, and groundthruth box respectively (likewise for *y, z, w, h, d*). Of course, we calculate the regression loss only for the foreground anchors and not for the background, so at the most we will calculate 64 targets for each batch.

To sum up training procedure, we train 2 layers for our TPN, scoring and reggression layers. The training loss includes the training losses obtained by these layers and its formula is:

$$L = \sum_i L_{cls}(p_i, p_i^*) + \sum_i p_i^* L_{reg}(t_i, t_i^*)$$

where:

- L_{cls} is the Cross Entropy loss we use for classifying the anchors, with p_i is the predicted label, p_i^* is the groundtruth class and $p_i, p_i^* \in \{0, 1\}$
- L_{reg} is the smooth-L1 loss function, which multiply it with p_i^* in order to set active only when we have a positive anchor ($p_i^* = 1$) and to be deactivated for background anchors ($p_i^* = 0$).

1.4.3 Validation

Validation procedure is a bit similar to training procedure. We randomly select 16 frames from a validation video and we examine if there is at least 1 proposed TOI which overlaps ≥ 0.5 with each groundtruth action tube and we get recall score. In order to get good proposals, after getting classification scores and targets prediction from the corresponding layers, we use Non-Maximum Suppresion (NMS) algorithm. We set NMS threshold equal with 0.7, and we keep the first 150 cuboids with the biggest score.

1.4.4 Modified Intersection over Union(mIoU) TODO check again

During training, we get numerous anchors. We have to classify them as foreground anchors or background anchors. Foreground anchors are those which contain some action, and, respectively, background don't. As presented before, IoU for cuboids calculates the ratio between volume of overlap and volume of union. Intuitively, this criterion is good for evaluating 2 tubes if they overlap but it has one big drawback: it considers x-y dimesions to have same importance with time dimension, which we do not desire. That's becase firstly we care to be accurate in time dimension, and then we can fix x-y domain. As a result, we change the way we calculate the Intesection Over Union. We calculate seperately the IoU in x-y domain (IoU-xy) and in t-domain (IoU-t). Finally, we multiply them in order to get the final IoU. So the formula for 2 tubes $(x_1, y_1, t_1, x_2, y_2, t_2)$ and $(x'_1, y'_1, t'_1, x'_2, y'_2, t'_2)$ is:

$$IoU_{xy} = \frac{\text{Area of Overlap in x-y}}{\text{Area of Union in x-y}}$$

$$IoU_t = \frac{max(t_1, t'_1) - min(t_2, t'_2)}{min(t_1, t'_1) - max(t_2, t'_2)}$$

$$IoU = IoU_{xy} \cdot IoU_t$$

The above criterion help us balance the impact of time domain in IoU. For example, let us consider 2 anchors: a = (22, 41, 1, 34, 70, 5) and b = (20, 45, 2, 32, 72, 5). These 2 anchors in x-y domain have IoU score equal to 0.61. But they are not exactly overlaped in time dim. Using the first approach we get 0.5057 IoU score and using the second approach we get 0.4889. So, the second criterion would reject this anchor, because there is a difference in time duration.

In order to verify our idea, we train TPN using both IoU and mIoU criterion for tube-overlapping. At Table 1.1 we can see the performance in each case for both datasets, JHMDB and UCF. Recall threshold for this case is 0.5 and during validation, we use regular IoU for defining if 2 tubes overlap.

Dataset	**Criterion**	**Recall(0.5)**
JHMDB	IoU	0.70525
	mIoU	0.7052
UCF	IoU	0.4665
	mIoU	0.4829

Table 1.1: Recall results for both datasets using IoU and mIoU metrics

Table 1.1 shows that modified-IoU give us slightly better recall performance only in UCF dataset. Thats reasonable, because JHMDB dataset uses trimmed videos so time duration doesn't affect a lot. So, from now own, during training we use mIoU as overlapping scoring policy.

1.4.5 Improving TPN score

After first test, we came with the idea that in a video lasting 16 frames, in time domain, all kind of actions can be seperated in the following categories:

1. Action starts in the n-th frame and finishes after the 16th frame of the sampled video.
2. Action has already begun before the 1st frame of the video and ends in the n-th frame.
3. Action has already begun before the 1st frame of the video and finishes after the 16th video frame.
4. Action starts and ends in that 16 frames of the video.

On top of that, we noticed that most of actions, in our datasets, last more that 16 frame. So, we came with the idea to add 1 scoring layer and 1 reggression layer which will proposed ToIs with fixed duration equal with sample duration and they will take into account the spatial information produced by activation maps. The new structure of TPN is shown in figure 1.6. After getting proposals from both scores, we concat them with ration 1:1 between ToI extracted from those 2 subnetworks.

Our goal is to “compress” feauture maps in temporal dimension in order to propose action tubes according only to the spatial information. So, we came with 2 techniques for doing such thing:

1. Use 3D Convolutional Layers with kernel size = (sample duration, 1,1), stride =1 and no padding for scoring and regression. This kernel “looks” only in the temporal dimension of the activation maps and doesn’t consider any spatial dependencies.
2. Get the average values from temporal dimension and then use a 2D Convolutional Layer for scoring and regression.

Training and Validation procedures remain the same. The only big difference is that now we have from 2 difference system proposed TOIs. So, we first concate them and, then, we follow the same procedure. For training loss, we have 2 different cross-entropy losses and 2 different smooth-L1 losses, each for every layer correspondly. So training loss is, now, defined as :

$$\begin{gathered} L = \sum_i L_{cls}(p_i, p_i^*) + \sum_i L_{cls}(p_{fixed,i}, p_{fixed,i}^*) + \\ \sum_i p_i^* L_{reg}(t_i, t_i^*) + \sum_i p_{fixed,i}^* L_{reg}(t_{fixed,i}, t_{fixed,i}^*) \end{gathered} \tag{1.1}$$

where:

- L_{cls} is the Cross Entropy loss we use for classifying the anchors, with p_i is the predicted label, p_i^* is the groundtruth class and $p_i, p_i^* \in \{0, 1\}$

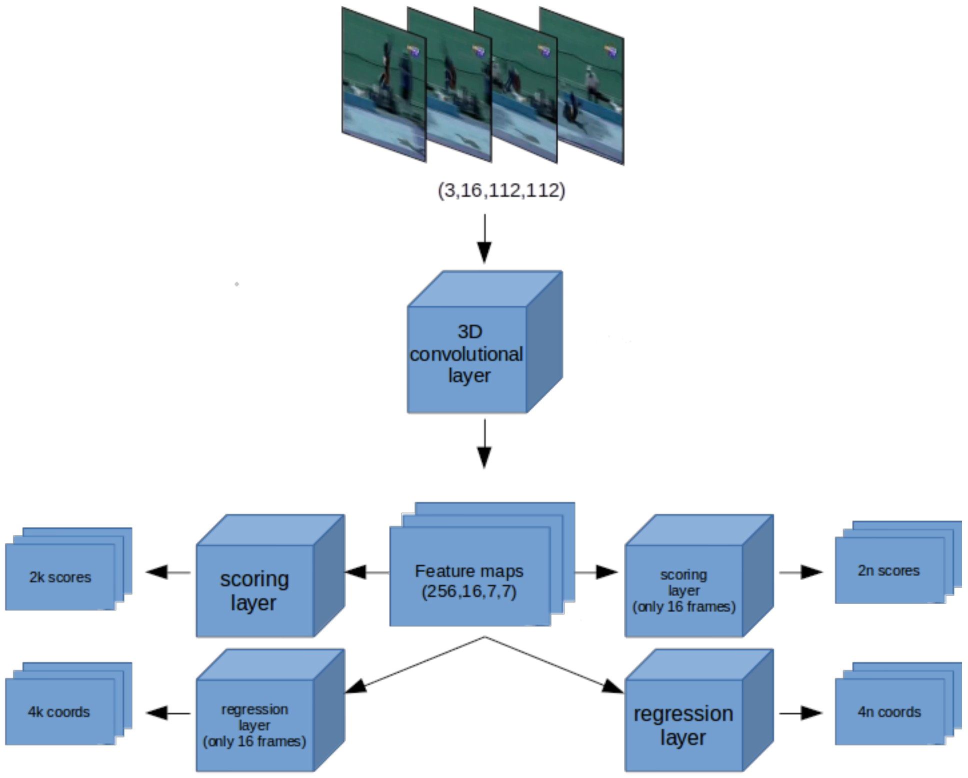

Figure 1.6: TPN structure after adding 2 new layers, where k = 5n.

- L_{reg} is the smooth-L1 loss function, which multiply it with p_i^* in order to set active only when we have a positive anchor ($p_i^* = 1$) and to be deactivated for background anchors ($p_i^* = 0$).
- p_i are the anchors from scoring layers with mutable time duration and p_i^* are their corresponding groundtruth label.
- $p_{fixed,i}$ are the anchors from scoring layers fixed time duration = 16 and $p^*_{fixed,i}$ are their corresponding groundtruth label.

We train our TPN Network using both techniques and their recall performance is show in Table 1.2.

As we can see from the previous results, the new layers increased recall performance significantly. On top of that, Table 1.2 shows that getting the average values from time dimension gives us the best results.

1.4.6 Adding regressor

The output of TPN is the α-highest scoring anchors moved according to their regression prediction. After that, we have to translate the anchor into

Dataset	Fix-time anchors	Type	Recall(0.5)
JHMDB	No	-	0.7052
	Yes	Kernel	0.6978
		Mean	0.7463
UCF	No	-	0.4829
	Yes	Kernel	0.4716
		Mean	0.4885

Table 1.2: Recall results after adding fixed time duration anchors

tubes. In order to do so, we add a regressor system which gets as input TOIs' feature maps and returns a sequence of 2D boxes, each for every frame. The only problem is that the regressor needs a fixed input size of featuremaps. This problem is already solven by R-CNNs which use roi pooling and roi align in order to get fixed size feature maps from ROIs with changing sizes. In our situation, we extend roi align operation, presented by Mask R-CNN, and we call it **3D Roi Align**.

3D Roi Align 3D Roi align is a modification of roi align presented by Mask R-CNN ([**?**]). The main difference between those two is that Mask R-CNN's roi align uses bilinear interpolation for extracting ROI's features and ours 3D roi align uses trilinear interpolation for the same reason. Again, the 3rd dimension is time. So, we have as input a feature map extracted from ResNet34 with dimensions (64,16,28,28) and a tensor containing the proposed TOIs. For each TOI whose actiovation map whose size is (64,16,7,7), we get as output a feature map with size (64, 16, 7, 7).

Regression procedure

At first, for each proposed ToI, we get its corresponding activation maps using 3D Roi Align. These features are given as input to a regressor. This regressor returns $16 \cdot 4$ predicted transforms $(\delta_x, \delta_y, \delta_w, \delta_h)$, 4 for each frame, where δ_x, δ_y specify the coordinates of proposal's center and δ_w, δ_h its width and height, as specified in [**?**]. We keep only the predicted translations, for the frames that are $\geq t_1$ and $< t_2$ and for the other frames, we set a zero-ed 2D box. After that, we modify each anchor from a cuboid written like $(x_1, y_1, t_1, x_2, y_2, t_2)$ to a sequence of 2D boxes, like:
$(0, 0, 0, 0, ..., x_{T_1}, y_{T_1}, x'_{T_1}, y'_{T_1}, ..., x_i, y_i, x'_i, ..., x_{T_2}, y_{T_2}, x'_{T_2}, y'_{T_2}, 0, 0, 0, 0, ...)$,
where:

- $T_1 \leq i \leq T_2$, for $T_1 < t_1 + 1, T_2 < t_2$ and $T_1, T_2 \in \mathbb{Z}$
- $x_i = x_1, y_i = y_1, x'_i = x_2, y'_i = y_2$.

Training In order to train our Regressor, we follow about the same steps followed previously for previous TPN's training procedure. This means that we randomly pick 16 ToI from those proposed by TPN's scoring layer. From those 16 tubes, 4 are foreground tubes, which meands 25% of the total number of the tubes as happened previously. We extract their corresponding features using 3D Roi Algin and calculate their targets like we did for regression layer. We feed Regressor Network with these features and compare the predicted targets with the expected. Again, we use smooth-L1 loss for loss function, calculated only for foreground ToIs. So, we add another parameter in training loss formula which is now defines as:

$$\begin{aligned} L = \sum_i L_{cls}(p_i, p_i^*) + \sum_i L_{cls}(p_{fixed,i}, p^*_{fixed,i}) + \\ \sum_i p_i^* L_{reg}(t_i, t_i^*) + \sum_i p^*_{fixed,i} L_{reg}(t_{fixed,i}, t^*_{fixed,i}) + \\ \sum_i q_i^* L_{reg}(c_i, c_i^*) + \end{aligned} \tag{1.2}$$

where except the previously defined parameters, we set c_i as the regression targets for picked tubes q_i. These tubes are the ones randomly selected from the proposed ToIs and q_i^* are their corresponding groundtruth action tubes, which are the closest to each q_i tube. Again we use q_i^* as a factor because we consider a tube as background when it doesn't overlaps with any groundtruth action tube more that 0.5 .

First regression Network

The architecture of reggression network is show in Figure 1.7, and it is described below:

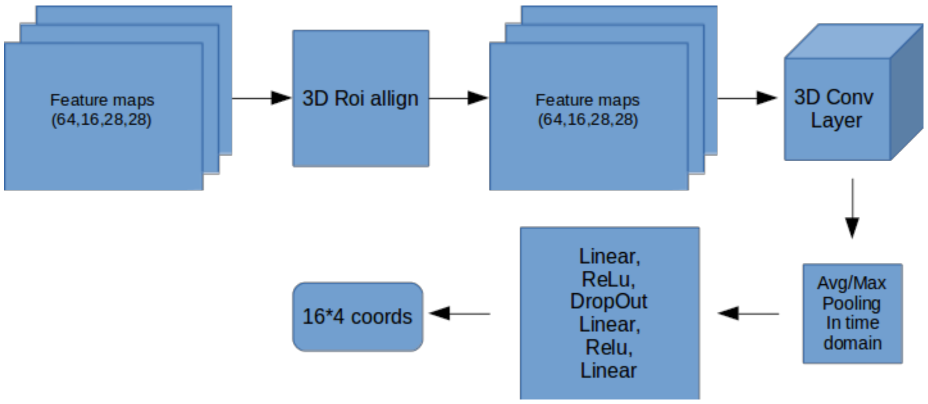
Feature maps (64,16,28,28)
3D Roi allign
Feature maps (64,16,28,28)
3D Conv Layer
Avg/Max Pooling In time domain
Linear, ReLu, DropOut Linear, Relu, Linear
16*4 coords

Figure 1.7: Structure of Regressor

1. Regressor is consisted, at first, with a 3D convolutional layer with kernel = 1, stride = 1 and no padding. This layer gets as input ToI's normalized activation map extracted by 3D Roi Align.

2. After that, we calculate the average value in time domain, so from a feature map with dimensions (64,16,7,7), we get as output a feature map (64,7,7).
3. These feature maps are given as input to a Linear layer followerd by a Relu Layer, a Dropout Layer, another Linear Layer and Relu Layer and a final Linear.

We use Recall metric In order to assess the performance of regressor. We calculate 3 recall performaces:

Cuboid Recall, which is the recall perfomance for proposed cuboids. We interested in this metric, because, we want to know how good are our proposals before modifying them into sequences of boxes.

Single frame Recall, which is the recall perfomance for the proposed ToI against the groundtruth tubes.

Follow-up Single Frame Recall, which is the recall performance for only the cuboids that were over the overlap threshold between proposed cuboids and groundtruth cuboids. We uses this metric in order to know how many of our proposed cuboids end up in being good proposals.

Dataset	**Pooling**	**Cuboid**	**Singl. Fr.**	**Follow-up S.F.**
JHMDB	avg	0.8545	0.7649	0.7183
	max	0.8396	0.7761	0.5783
UCF	avg	0.5319	0.4694	0.5754
	max	0.5190	0.5021	0.5972

Table 1.3: Recall results after convertying cuboids into sequences of frames

As the above results show, we get lower recall perfomance in frame-level. On top of that, when we translate a cuboid into a sequence of boxes, we miss 20-40% of our proposals. This means that we don't modify good enough our cuboids, algouth we get only 10% decrease. Probably, we get such score from cuboids, that even though didn't overlap well (according to overlap threshold), achieve to become a good proposal in frame-level and in temporal level.

1.4.7 Changing Regressor - from 3D to 2d

After getting first recall results, we experiment using another architecture for the regressor network, in ordet to solve the modification problem, introduced in previous section. Instead of having a 3D Convolutional Layer, we will use a 2D Convolutional Layer in order to treat the whole time dimension as one during convolution operation. So, as shown in Figure 1.8, the 2^{nd} Regression Network is about the same with first one, with 2 big differences:

1. We performing a pooling operation at the feature maps extracted by 3D Roi Align operation, after we are normalized.

2. Instead of a 3D Convolutional Layer, we have a 2D Convolutional Layer with kernel size = 1, stride = 1 and no padding.

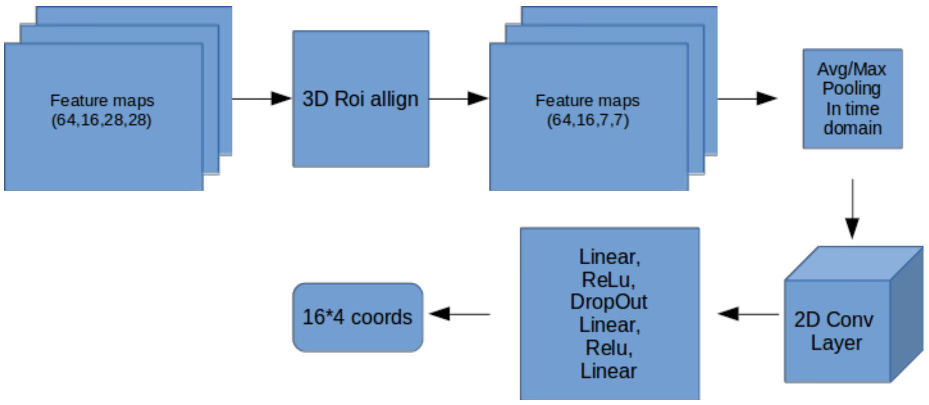
Feature maps (64,16,28,28)
3D Roi allign
Feature maps (64,16,7,7)
Avg/Max Pooling In time domain
2D Conv Layer
Linear, ReLu, DropOut Linear, Relu, Linear
16*4 coords

Figure 1.8: Structure of Regressor

On top of that, we tried to determine which feature map is the most suitable for getting best-scorig recall performance. This feature map will be given as input to Roi Algin operation. At Table 1.4, we can see the recall performance for different feature maps and different pooling methods.

Dataset	**Pooling**	**F. Map**	**Recall**	**Recall SR**	**Recall SRF**
JHMDB	mean	64	0.6828	0.5112	0.7610
		128	0.8694	0.7799	0.6756
		256	0.8396	0.7687	0.7029
	max	64	0.8582	0.7985	0.5914
		128	0.8358	0.7724	0.8118
		256	0.8657	0.8022	0.7996
UCF	mean	64	0.5055	0.4286	0.5889
		128	0.5335	0.4894	0.5893
		256	0.5304	0.4990	0.6012
	max	64	0.5186	0.4990	0.5708
		128	0.5260	0.4693	0.5513
		256	0.5176	0.4878	0.6399

Table 1.4: Recall performance using 3 different feature maps as Regressor's input and 2 pooling methods

As we noticed from the above results, again, our system has difficulty in translating cuboids into 2D sequence of ROIs. So, that makes us rethink the way we designed our TPN.

1.5 3D anchors as 4k-dim vector

In this approach, we set 3D anchors as 4k coordinates (k = 16 frames = sample duration). So a typical anchor is written as $(x_1, y_1, x'_1, y'_1, x_2, y_2, ...)$ where x_1, y_1, x'_1, y'_1 are the coordinates for the 1st frame, x_2, y_2, x'_2, y'_2 are the coordinates for the 2nd frame etc, as presented in [**?**]. In figure 1.9 we can an example of this type of anchor.

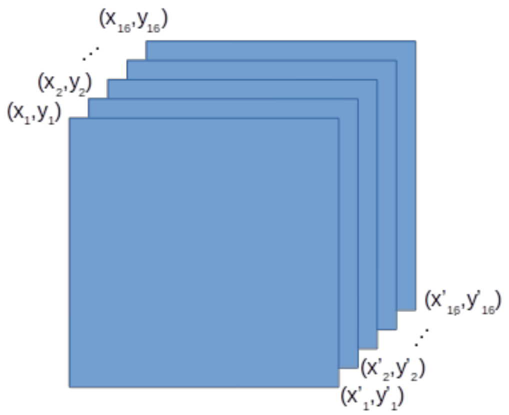

Figure 1.9: An example of the anchor $(x_1, y_1, x'_1, y'_1, x_2, y_2, ...)$

The main advantage of this approach is that we don't need to translate the 3D anchors into 2D boxes, which caused many problems at the previous approach. However, it has a big drawback, which is the fact that this type of anchors has fixed time duration. In order to deal with this problem, we set anchors with different time durations, which are 16, 12, 8 and 4. Anchors with duration < sample duration (16 frames) can be written as 4k vector with zeroed coordinateds in the frames bigger that the time duration. For example, an anchor with 2 frames duration, starting from the 2nd frame and ending at the 3rd can be written as (0, 0, 0, 0, $x_1, y_1, x'_1, y'_1, x_2, y_2, x'_2, y'_2$, 0, 0, 0, 0) if sample duration is 4 frames.

This new approach led us to change the structure of TPN. The new one can is presented in figure 1.10. As we can see, we added scoring and regression layers for each duration. So, TPN follows the next step in order to propose action tubes:

1. At first, we get the feature map, extracted by ResNet, as input to a 3D Convolutional Layer with kernel size = 1, stride = 1 and no padding.

2. From Convolutional Layer, we get as output an activation map with dimensions (256,16,7,7). For reducing time dimension, we use 4 pooling layer, one for each sample duration with kernel sizes *(16,1,1), (12,1,1,), (8,1,1) and (4,1,1)* and stride = 1, for sample durations 16, 12, 8 and 4 respectively. So, we get activation maps with dimensions *(256,1,7,7),*

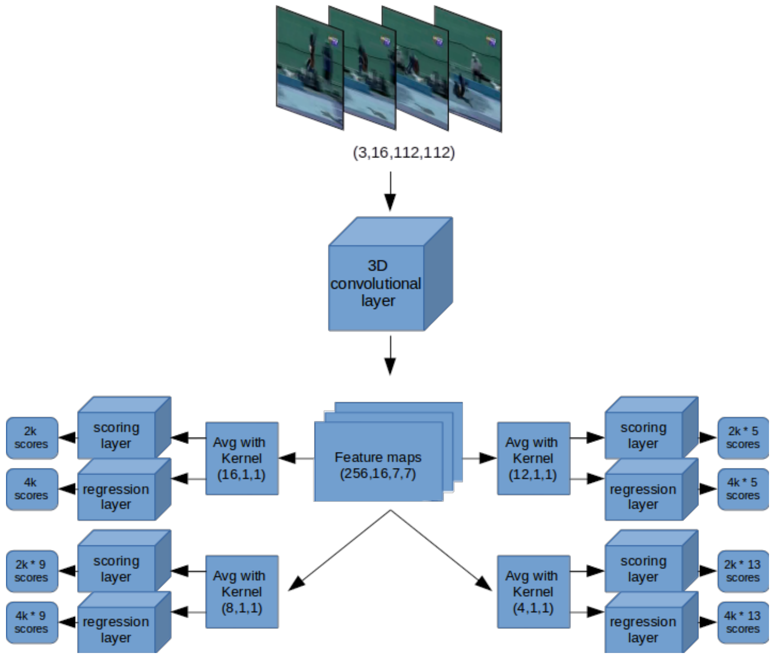

Figure 1.10: The structure of TPN according to new approach

(256,5,7,7), (256,9,7,7) and (256,13,7,7), in which second dimension is the number of possible time variations. For example, in $(256, 5, 7, 7)$ feature map, which is related with anchors with duration 12 frames, we can have 5 possible cases, from frame 0 to frame 11, frame 1 to frame 12 etc.

3. Again, like in previous approach, for each pixel of the activate map we correspond $\mathbf{n = k = 15}$ anchors (5 scale of 1, 2, 4, 8, 16, 3 aspect rations of 1:1, 1:2, 2:1). Of course, we have 4 different activate maps, with 1, 5, 9 and 13 different cases and a 7×7 shape in each filter. So, in total we have $28 \cdot 15 \cdot 49 = 20580$ different anchors. Respectively, we have 20580 different regression targets.

1.5.1 Training

Training procedure stays almost the same like previous approach's. So, again, we randomly choose a video segment and we consider anchors with overlap bigger than 0.8 with any groundtruth tube, alongside with background anchors whose overlap is bigger that 0.1 and smaller than 0.3.

As Table 1.5, it is obvious that we get better recall performances compared to previous' approach. Additionally, we can see that 3Dmax pooling performs better than 3D avg pooling. The difference between max pooling and avg pooling is about 10%, which is big enough to make us choose max pooling operation

Dataset	Pooling	Recall(0.5)	Recall(0.4)	Recall(0.3)
JHMDB	mean	0.6866	0.7687	0.8582
	max	0.8134	0.8694	0.9216
UCF	avg	0.5435	0.6326	0.7075
	max	0.6418	0.7255	0.7898

Table 1.5: Recall results using 2nd approach for anchors

before getting anchors' scores and regression targets.

1.5.2 Adding regressor

Even though, our TPN outputs frame-level boxes, we need to improve these predictions in order to overlap with the groundtruth boxes as well as possible. So, in full correspondance with the previous approach, we added an regressor for trying to get better recall results.

3D Roi align In this approach, we know already the 2D coordinates. So, we can use the method proposed in [**?**]. They extend RoiAlign operator by splitting the tube into T 2D boxes. Then, they use classic RoiAlign to extract a region from each of the temporal slices in the feature map. After that, they concatenate the region in time domain so they get a $T \times R \times R$ feature map, where R is the output resolution of RoiAlign, which is 7 in our situation.

As a first approach, we use a 3D convolutional layer followed by 2 linear layer. Our regressors follows the following steps:

1. At first, use 3D RoiAlign in order to extract the feature maps of the proposed action tubes. We normalize them, and give them as input to the 3D convolutional layer.

2. The output of the 3D Convolutional Layer is fed into 2 Linear layers with ReLu faction between them and finally we get $sampleduration \times 4$ regression targets. We keep only the proposed targets, that there is a corresponding 2D box.

We train our regressor using the same loss function as previous approach's formula which is:

$$L = \sum_i L_{cls}(p_i, p_i^*) + \sum_i L_{cls}(p_{fixed,i}, p_{fixed,i}^*) +$$
$$\sum_i p_i^* L_{reg}(t_i, t_i^*) + \sum_i p_{fixed,i}^* L_{reg}(t_{fixed,i}, t_{fixed,i}^*) +$$
$$\sum_i q_i^* L_{reg}(c_i, c_i^*) +$$

We want again to find the best matching feature maps, so we train our regressor for feature maps $(64, 8, 7, 7)$ and $(128, 8, 7, 7)$. We didn't experiment using $(256, 8, 7, 7)$ feature map because we got OutOfMemory error during training, despite several modifications we did in the implementation code.

Dataset	**Feat. Map**	**Recall(0.5)**	**Recall(0.4)**	**Recall(0.3)**
JHMDB	64	0.7985	0.903	0.9552
	128	0.7836	0.8881	0.944
UCF	64	0.5794	0.7206	0.8134
	128	0.5622	0.7204	0.799

Table 1.6: Recall performance when using a 3D Convolutional Layer in Regressor's architecture

According to Table 1.6, we got the best results when we use $(64, 16, 7, 7)$ feature map. This is the expected result, because these feature maps are closer to the actual pixels of the actor, than $(128, 16, 7, 7)$ feature maps, in which because of $3 \times 3 \times 3$ kernels, which combine spation-temporal information from neighbour pixels. However, as we can see, we got worst recall performance than when we didn't use any regressor if we compare results from Tables 1.5 and 1.6.

1.5.3 From 3D to 2D

Following the steps we used before, we design an architecture that uses instead of a 3D Convolutional Layer, a 2D. Unlike we did before, in this case, we don't use any pooling operation before feeding the first 2D Convolutional Layer. On the contrary, we manipulate our feature maps like not being spatio-temporal but, only spatial. So, our steps are:

1. At first, we use, again ,3D RoiAlign in order to extract the feature maps of the proposed action tubes and normalize them. Let us consider a feature map extracted from ResNet which has dimensions $(64, sampleduration, 7, 7)$ and after applying RoiAlign and normalization, we get a $(k, 64, sampleduration, 7, 7)$ feature map, where k is the number of proposed action tubes for this video segment.

2. We slice the proposed action tubes into T 2D boxes, so the dimensions of the Tensor, which contains the coordinates of action tubes, from $(k, 4 \cdot sampleduration)$ become $(k, sampleduration, 4)$. We reshape the Tensor into $(k \cdot sampleduration, 4)$, in which, first k coordinates refer to the first frame, the second k coordinates refer to the second frame and so on.

3. Respectively, we reshape extracted feature maps from $(k, 64, sampleduration, 7, 7)$ to $(k \cdot sampleduration, 64, 7, 7)$. So, now we deal with 2D feature maps, for which as we said before, we consider that contain only spatial information. So, we use 3 Linear Layers in order to get 4 regression targets. We keep only those we have a corresponding bounding box.

Again, we experiment using 64, 128 and 256 feature maps (in this case, there is no memory problem). The results of our experiments are shown at Table 1.7.

Dataset	**Feat. Map**	**Recall(0.5)**	**Recall(0.4)**	**Recall(0.3)**
JHMDB	64	0.8358	0.9216	0.9739
	128	0.8172	0.9142	0.9627
	256	0.7724	0.8731	0.9328
UCF	64	0.6368	0.7346	0.7737
	128	0.6363	0.7133	0.7822
	256	0.6363	0.7295	0.7822

Table 1.7: Recall performance when using a 2D Convolutional Layer instead of 3D in Regressor's model

As we can see, we get improve recall performance up 3% for JHMDB dataset and about the same perfomance for UCF dataset. Again, we get best performance if we choose $(64, 16, 7, 7)$ feature maps.

1.5.4 Changing sample duration

After trying all the previous versions, we noticed that we get about the same recall performances with some small improvements. So, we thought that we could try to reduce the sample duration. This idea is based on the fact that reducing sample duration, means that anchor dimensions will reduce, so the number of candidate anchors. That's because, now we have smaller number of cases, so smaller number of parameters alongside with small number of dimensions for regression targets. We train our TPN for sample duration = 8 frames 4 frames. We use, of course, TPN's second architecture, because as shown before, we get better recall performance.

Without Regressor

At first, we train TPN, again without regressor. We do so, in order to compare recall performance for all sample durations, without using any regressor. The results are shown in Table 1.8. For all cases, we use max pooling before scoring and regression layers, and we didn't experiment at all with avg pooling. Of course, for sample duration = 16, we used the calculated one in Table 1.5.

According to Table 1.8, we notice that we get best performance for sample duration = 8 for both datasets. For dataset JHMDB sample duration equal with 8 is gets far better results from the others aprroaches, followed by approach with sample duration = 4. For UCF dataset, although sample duration equal with 8 gives us best performances sample duration equal with 4 gives us about the same. The difference between those 2 duration is less that 1%.

Dataset	Sample dur	Recall(0.5)	Recall(0.4)	Recall(0.3)
JHMDB	16	0.8134	0.8694	0.9216
	8	0.9515	0.9888	1.0000
	4	0.8843	0.9627	0.9888
UCF	16	0.6418	0.7255	0.7898
	8	0.7942	0.8877	0.9324
	4	0.7879	0.8924	0.9462

Table 1.8: Recall results when reducing sample duration to 4 and 8 frames per video segment

With Regressor

Following the idea of reducing sample duration for getting better recall performance, we trained TPN with a regressor. We trained for both approaches, which means both 3D and 2D Convolutional Layer approaches were trained. Recall performances are presented at Table 1.9.

Dataset	Sample dur	Type	Recall(0.5)	Recall(0.4)	Recall(0.3)
UCF	8	2D	0.8078	0.8870	0.9419
		3D	0.8193	0.8930	0.9487
	4	2D	0.7785	0.8914	0.9457
		3D	0.7449	0.8605	0.9362
JHDMBD	8	2D	0.9366	0.9851	0.9925
		3D	0.8918	0.9776	0.9963
	4	2D	0.9552	0.9963	1.0000
		3D	0.9142	0.9701	0.9888

Table 1.9: Recall results when a regressor and sample duration equal with 4 or 8 frames per video segment

According to 1.9, it is clear that using a 2D Convolutional Layer as presented above results in better recall performance that using a 3D. Futhermore, we notice that the addition of a regressor causes both improvements and deteriorations in recall performances. For dataset UCF, approach with sample duration = 8 improves by about 1-2% recall performance but for sample duration = 4 it reduce it by 1-3%. On the other hand, for dataset JHMDB, now, sample duration = 4 gets better results by adding a regressor and sample duration = 8 gets worse. So, after considering both results from Tables 1.8 and 1.9, we think that the best approach is using sample duration equal with 8, with the adddition of a regressor, which uses a 2D Convolutional Layer. We know that this appoarch gets worse performance at JHMDB but it gives us the best results in UCF. But, since JHMDB's results are high enough, we are most interested in improving UCF's results. That's the reason, we will use the aformentioned approach in the rest chapters.

1.6 General comments

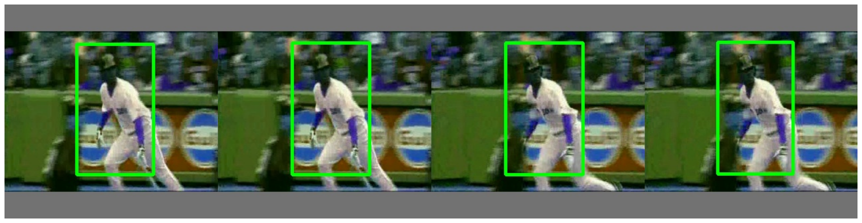
Figure 1.11: Structure of the whole network

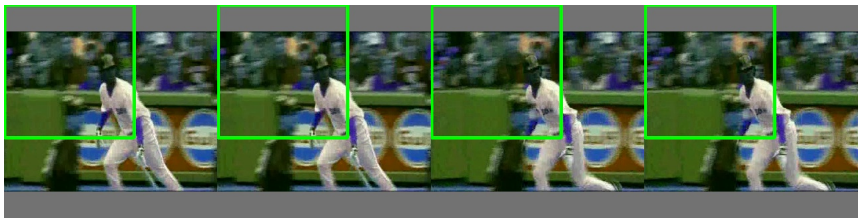
Figure 1.12: Structure of the whole network

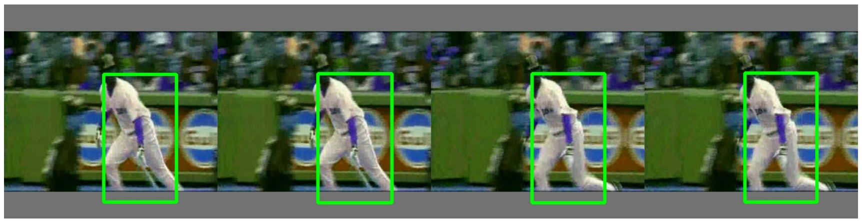
Figure 1.13: Structure of the whole network