

Initial Report on New SNN Based Olympic Counter Model Backbone

Submitted by: Mohammad Raghieb Noor

The issues with our initial model was problems with latency. The more the latency the more frames were missed in the buffer from processing. It was suspected that the latency could be brought down if the single stage object detector and classifier could be turned into a two stage object detector and classification model.

The model thus continued in three phases. In the first stage we spent time in building up our dataset from yolo annotated data by reading the txt files. The first phase was spent in extracting and processing data from existing databases. The second phase was spent in training classifiers based on Resnet50 V1 and VGG16 respectively and once that was done, the second phase proceed in training Siamese neural networks. Due to lack of accuracy in both the phases with 21 categories, we shifted to three categories instead of twenty one which was proven effective when training the model for classification alone.

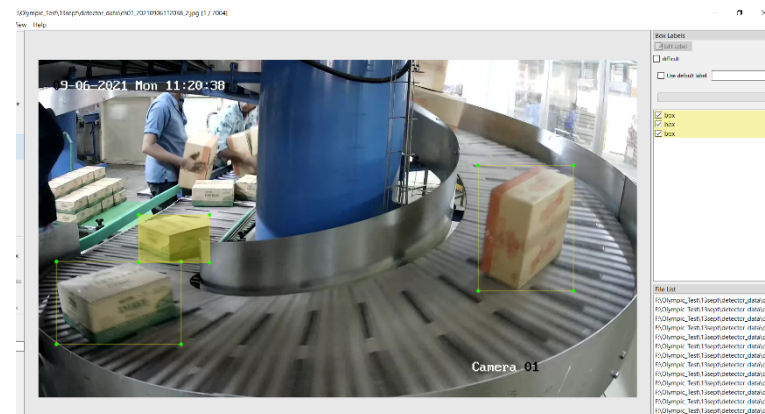
PHASE 1:

In the first stage of model we used the existing YOLO format data to create data for the object detector:

1)

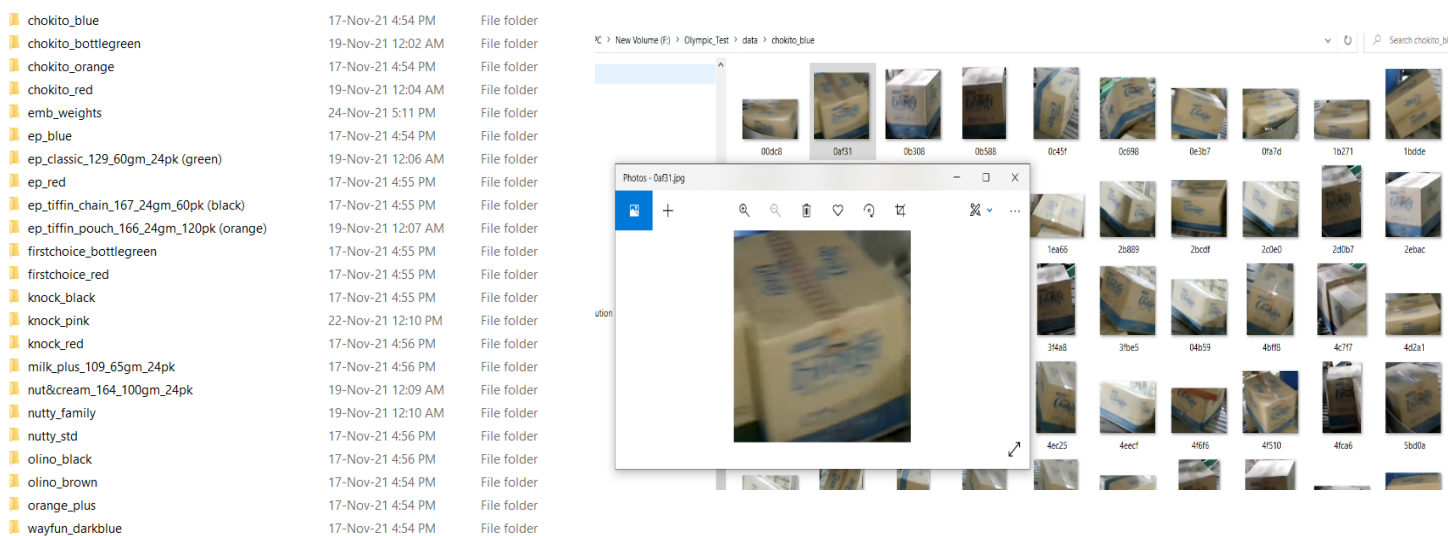


2)



We had sampled **1716** annotated frames were used as **train dataset** and **583** annotated frames were used as **test dataset** like this in order to trial on our detector data.

Once this step was done, we proceeded towards extracting box data from our yolo database for our classification tasks. As we can see below, the boxes have been extracted directly from our yolo based data.



PHASE 2:

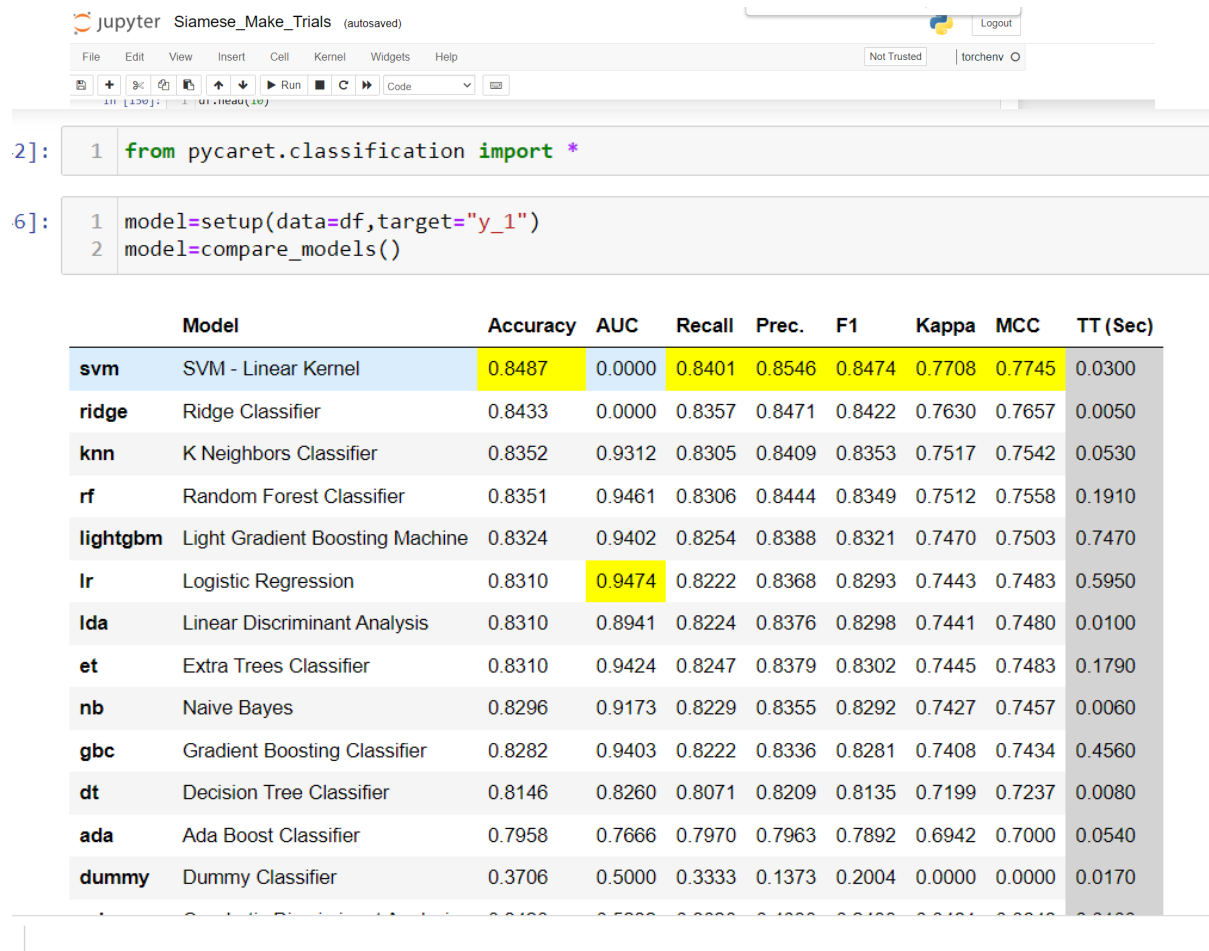
In the first phase of the new model, an SSD MobileNetV2 (300x300) from Tensorflow Object Detection API was used to create our preliminary object detection. We reached an accuracy of 90% in localization of the boxes. After that we shifted towards training the classifier models which were based on VGG16 and ResNet50 V1. We used data from all 21 categories, we used 200-300 images per class, however in the cases of training the models, our accuracy was very bad, even when we trained for a long time and reached about roughly 80-85 percent accuracy in training, the output results were not good at all. (Training results can be reproduced by going through the the notebook:



After all this was done, we moved unto making Siamese Neural Network (SNN) based classifiers. In the first stage, of training the SNN, we found that the loss was not reducing for 21 classes when. The loss didn't rarely budged to fall under 0.90, which, in the case of a Siamese Neural Network, is very high. However, in the case of the model with 3 categories, we found significant improvement in reduction of Loss and reached a very low loss amounting to approx. 0.04. Training results can be reproduced by going through notebook attached with the file (**Siamese_Make_Trials.IPYNB**). The structure of our Embedder Model looks like this:

Once this was done, we moved onto making a classifier. The classifier basically takes the embedded vectors that we get as output from our SNN and then fit it as input to another ML model that gives us an output of which category the embedded vector belongs to. In making a classifier, I followed the basic SNN convention of training a simpler machine learning model. But while making this, I turned towards comparing performance of basic machine learning models on a rotation of 10 epochs on our embedder data that we extracted from the data base.

Embedder Data:



The screenshot shows a Jupyter Notebook titled "Siamese_Make_Trials (autosaved)". It contains two code cells. The first cell (index 2) imports all functions from `pycaret.classification`. The second cell (index 6) sets up the data and compares models. Below the code cells is a table showing the performance of various machine learning models on the embedder data.

	Model	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC	TT (Sec)
svm	SVM - Linear Kernel	0.8487	0.0000	0.8401	0.8546	0.8474	0.7708	0.7745	0.0300
ridge	Ridge Classifier	0.8433	0.0000	0.8357	0.8471	0.8422	0.7630	0.7657	0.0050
knn	K Neighbors Classifier	0.8352	0.9312	0.8305	0.8409	0.8353	0.7517	0.7542	0.0530
rf	Random Forest Classifier	0.8351	0.9461	0.8306	0.8444	0.8349	0.7512	0.7558	0.1910
lightgbm	Light Gradient Boosting Machine	0.8324	0.9402	0.8254	0.8388	0.8321	0.7470	0.7503	0.7470
lr	Logistic Regression	0.8310	0.9474	0.8222	0.8368	0.8293	0.7443	0.7483	0.5950
lda	Linear Discriminant Analysis	0.8310	0.8941	0.8224	0.8376	0.8298	0.7441	0.7480	0.0100
et	Extra Trees Classifier	0.8310	0.9424	0.8247	0.8379	0.8302	0.7445	0.7483	0.1790
nb	Naive Bayes	0.8296	0.9173	0.8229	0.8355	0.8292	0.7427	0.7457	0.0060
gbc	Gradient Boosting Classifier	0.8282	0.9403	0.8222	0.8336	0.8281	0.7408	0.7434	0.4560
dt	Decision Tree Classifier	0.8146	0.8260	0.8071	0.8209	0.8135	0.7199	0.7237	0.0080
ada	Ada Boost Classifier	0.7958	0.7666	0.7970	0.7963	0.7892	0.6942	0.7000	0.0540
dummy	Dummy Classifier	0.3706	0.5000	0.3333	0.1373	0.2004	0.0000	0.0000	0.0170

Machine Learning Model Comparison:

From the images above we can see the type of data we used to train the classifier, the embedded vectors were of size 108 and in the image the 109th column was basically the labels, named: "y_1".

After that we have also added a comparison of different traditional machine learning model performance on the data with several metrics for accuracy, and efficiency(TT).

PHASE 3:

On the third phase we built the script to join together all the components.

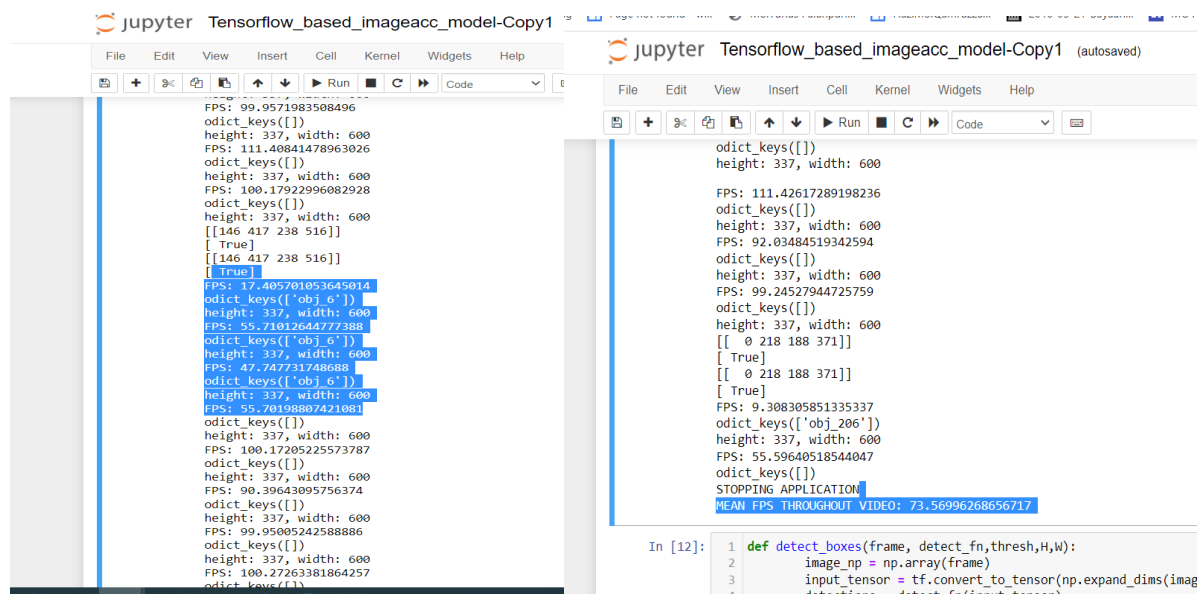
- In this phase we made sure that our **classification took place only once**, that is, when the **detected boxes are mid way into the frame**, as that is where the **view of boxes** is the **clearest**. The code reference can be seen in the tracker section of the notebook (**SNN_RnD_Project_Script**), the function to look for:
box_recognize(self,image,model,svm,label_map)
- The implementation of this required some modification of our existing tracker, notable changes can be found in the **tracker.update** function.
- The problem of one box jumping over another was solved in the **update_track** function in **Track Class** in the **track_update** function in the **Tracker Class** function in and functions. Main loop script can be consulted for further reference.
- The rest of the model followed counting logic from the previous model.
- The notebook used in this phase:

Results, Conclusion and Recommendations:

Results:

For the purpose of a prototype, the first problem that we were trying to tackle was the FPS we get per frame in order to reduce latency. Since there are several components, we need to describe the FPS of not just the detection and classification algorithm, but the frames in which the classifications occur. So we set a `time.time()` (`var_name = start`) from the start of each frame and put another `time.time()` (`var_name=end`) and calculated the frame per second by the formula $1/(end-start)$.

In order for a starting run, we set the frames skipped to 10. This lead us to the following results:



The image shows two screenshots of a Jupyter Notebook interface. The left screenshot shows a code cell with multiple lines of output, including FPS values and object detection results. The right screenshot shows a code cell with a single line of output, indicating the mean FPS throughout the video.

```
FPS: 99.9571983508496
odict_keys([])
height: 337, width: 600
FPS: 111.40841478963026
odict_keys([])
height: 337, width: 600
FPS: 100.17922996082928
odict_keys([])
height: 337, width: 600
[[146 417 238 516]]
[ True]
[[146 417 238 516]]
[ True]
FPS: 17.405701053645014
odict_keys(['obj_6'])
height: 337, width: 600
FPS: 55.71012644777388
odict_keys(['obj_6'])
height: 337, width: 600
FPS: 47.747731748688
odict_keys(['obj_6'])
height: 337, width: 600
FPS: 55.70198807421083
odict_keys([])
height: 337, width: 600
FPS: 100.17205225573787
odict_keys([])
height: 337, width: 600
FPS: 90.39643095756374
odict_keys([])
height: 337, width: 600
FPS: 99.95005242588886
odict_keys([])
height: 337, width: 600
FPS: 100.27263381864257
odict_keys([])
```

```
odict_keys([])
height: 337, width: 600
FPS: 111.42617289198236
odict_keys([])
height: 337, width: 600
FPS: 92.03484519342594
odict_keys([])
height: 337, width: 600
FPS: 99.24527944725759
odict_keys([])
height: 337, width: 600
[[ 0 218 188 371]]
[ True]
[[ 0 218 188 371]]
[ True]
FPS: 9.308305851335337
odict_keys(['obj_206'])
height: 337, width: 600
FPS: 55.59640518544047
odict_keys([])
STOPPING APPLICATION
MEAN FPS THROUGHOUT VIDEO: 73.56996268656717
```

From the above data, it can be inferred that the classification takes the longest, and the frames in which **classification** is required, our model takes upto about **1/17 second (17 FPS)** and the frames in which just **object detection and tracking** is done, our model takes about **1/55 second (55 FPS)** as for the frames in which **just tracking is done**, our model takes about **1/100 second (100-110 FPS)**. Thus according to current results and meagre existing data, we can see that this model pipelines provides us much less latency than our existing model by bringing our average FPS to about **73.569 frames/sec.**

Conclusion:

- The problems we faced in this model was that the SNN worked much more prematurely than it did in the training phase.
- The object detection was creating problems due to its preliminary detection results and training data.
- The model is still at its infancy stages and does require further improvements.

- We can see improvements in latency which was the initial target, but further development is required in the classification and detection models.

Recommendation:

- One of the reasons I speculate the SNN is not working properly is because during training, we used Tensorflow pipeline which uses RGB image channelling whereas the video was extracted with CV2 as documentation on Tensorflow or PIL video frame extractors was not found. Now the problem with using CV2 here is CV2 uses BGR channeling this changes our input matrix structure during testing from that of which was used in training. In order to solve this, we can opt for greyscaled image inputs while training our Siamese, as this creates single channelled images, and both TF and CV2 are supposed to provide similar matrix inputs for the Siamese as there is only one channel unlike the colored three channelled RGB or BGR format
- Another problem was with the detection of the boxes. I believe this can be solved by simply opting for a better SSD model such as SSD Mobile Net V2 (600x600)
- Another recommendation would be to load all our models with cv2.dnn module, that way, we can use blobs to counteract the RGB – BGR issues and hope to achieve results we received during training as we talked about above in the conclusions.
- **We can also go beyond regular convention of opting for the SVM according to Siamese papers, we can also fit more dense layers to our existing Siamese model, and then train just the dense layers on the existing data, that way we wont even need to pass data to the SVM and definitely reach better inference speed during classification and also theoretically increase accuracy compared to the SVM results we got on training.**

References

Reference for SSD MobileNet V2 (300x300) and SSD MobileNet V2 (600x600):

- https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf2_detection_zoo.md

Reference for whole TFOD Object Detection Training Module:

- <https://github.com/nicknochnack/TFODCourse>