3D Object Detection

Description

This project aims to derive a 3D version (true 3D input “gray” images), from the original 2D (RGB) Mask R-CNN implementation.

Contents

Technical Details

Programming Language Used:

- Python

Libraries to be Used:

- Keras
- TensorFlow
- numpy
- sklearn
- os
- random
- distutils.version

Literatures to be used:

- *Mask R-CNN* (https://arxiv.org/pdf/1703.06870.pdf) • *Feature Pyramid Networks for Object Detection* (https://arxiv.org/pdf/1612.03144.pdf)
- *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*(https://papers.nips.cc/paper/5638-faster-r-cnntowards-real-time-object-detection-with-region-proposal-networks.pdf)
- *Weakly Supervised Region Proposal Network and Object Detection*(http://openaccess.thecvf.com/content_ECCV_2018/papers/Peng_Tang_Weakly_Supervised_Region_ECCV_2018_paper.pdf)
- *3D Object Proposals for Accurate Object Class Detection* (https://papers.nips.cc/paper/5644-3d-object-proposals-foraccurate-object-class-detection.pdf)

Blogs to be considered:

- *https://www.pyimagesearch.com/2018/11/19/mask-r-cnn-with-opencv/*
- *https://medium.freecodecamp.org/mask-r-cnn-explained-7f82bec890e3*
- *https://medium.com/@jonathan_hui/understanding-feature-pyramidnetworks-for-object-detection-fpn-45b227b9106c*
- *https://medium.com/@tanaykarmarkar/region-proposal-network-rpnbackbone-of-faster-r-cnn-4a744a38d7f9*

- *https://engineering.matterport.com/splash-of-color-instancesegmentation-with-mask-r-cnn-and-tensorflow-7c761e238b46* (***Most Important***)

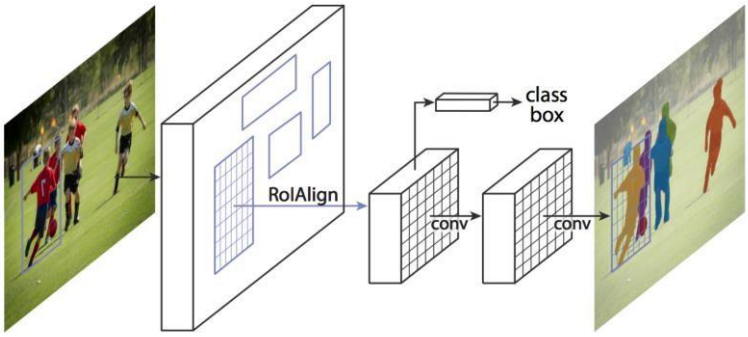
Project Objective:

- Develop a customized state of the art model for 3D Object Detection.
- Extend the Mask-RCNN implementation for 3D input tensors.(https://github.com/matterport/Mask_RCNN/)

Model Architecture

Overview:

Our basic Mask R-CNN model consists of two-stage frameworks. The first stage takes the image as input and generates proposals (areas where the desired object is probable). The second stage classifies the proposals and generates bounding boxes and masks.



Mask R-CNN framework. Source: https://arxiv.org/abs/1703.06870

The modules are as follows:

ResNet Backbone:

Function: *resnet_graph()*

The backbone is simply a standard convolutional neural network model, which acts as a feature extractor. The current model supports both *ResNet50* and *ResNet101* and these are great feature extractors. The early layers detect low level features mainly corners and edges and as we move forward via convolutions we have the later layers which can detect high-level features like the object itself. This is because for every convolution, we have weights associated with it, which try to tune the features better and better for every forward step and eventually the training leads us close to objects detection as a whole.

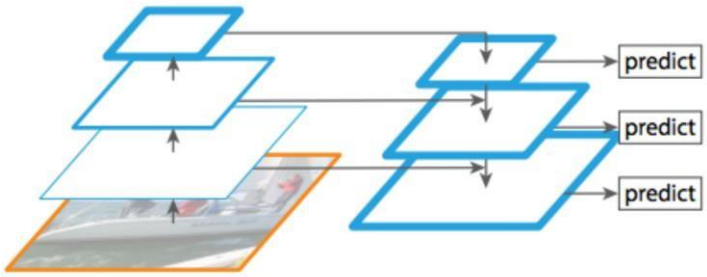
When we pass the input image through this backbone, we convert an RGB image (*x,y,3*) is converted to a feature map of shape (*f1,f2,channels*) where *x,y >> f1,f2* and *channels >>3*. This feature map gives us basically the *bottom-up levels* of FPN, the *first pyramid*.

Feature Pyramid Network (FPN):

Function: *MaskRCNN.build()*

We can still improve our ResNet backbone further. It basically represents the objects at multiple scales far better. The standard feature extraction did only consist of ResNet backbone, which for every forward convolution gives better and better features, and ultimately what we get is the *bottom-up layer feature pyramid*.

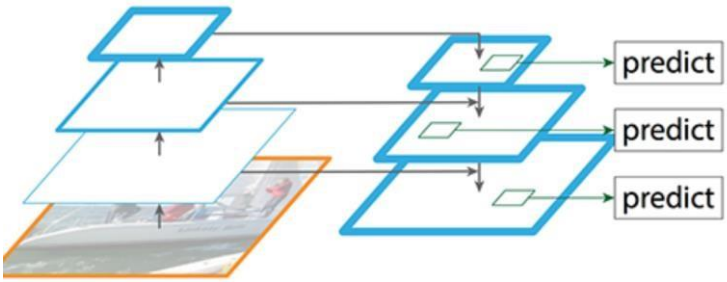
FPN improves the standard feature extraction by adding another pyramid (second) which takes in the high level features from the first pyramid (the bottom-up layer) and passes them to lower layers. This allows features at every level to have access to features at both high and low levels. This constitutes the *top-down feature map of the FPN*.



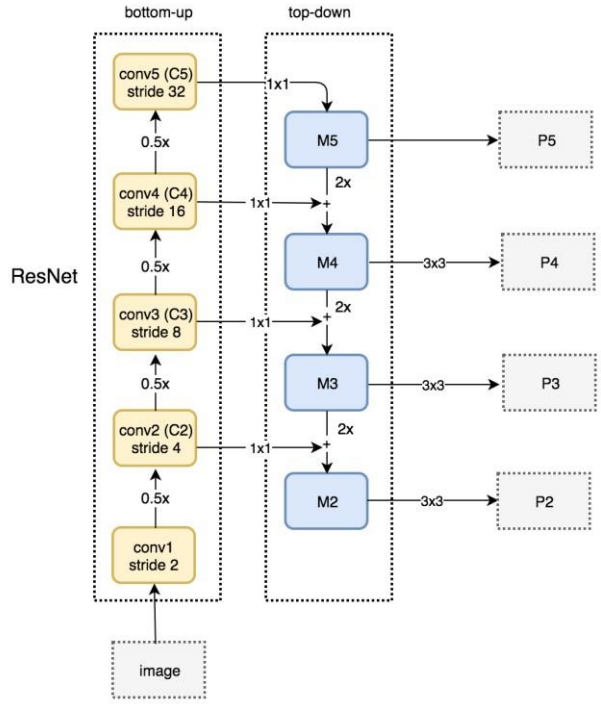
Source: Feature Pyramid Networks paper

Which one to pick basically depends on the size of the target object. If we talk about the bottom-up levels first, then we can say that as we go up, the *resolution* decreases but the *semantic value* increases. Now *semantically richest layer* is fed from bottom-up pyramid to the top most layer of the *top-down feature pyramid*. The reconstructed layers are semantically rich but the *location* of object is not yet precise.

To avoid this, *lateral connections* (skip connections in a way) are added between layers of both top-down and bottom-up feature pyramids.



Add skip connections (Source)



P layers are constructed by doing (*3*,*3*) convolutions to the merged *M* layers. Why stop at *P2*? Answer is *speed*! The *spacial dimensions* of *C1* are excessively large and as such, the calculations for *P1* might slow down the training and calculation process. All *P* layers must have same channels since we are using a same detector. Now these *P* levels are fed into the *RPN*, which we will discuss.

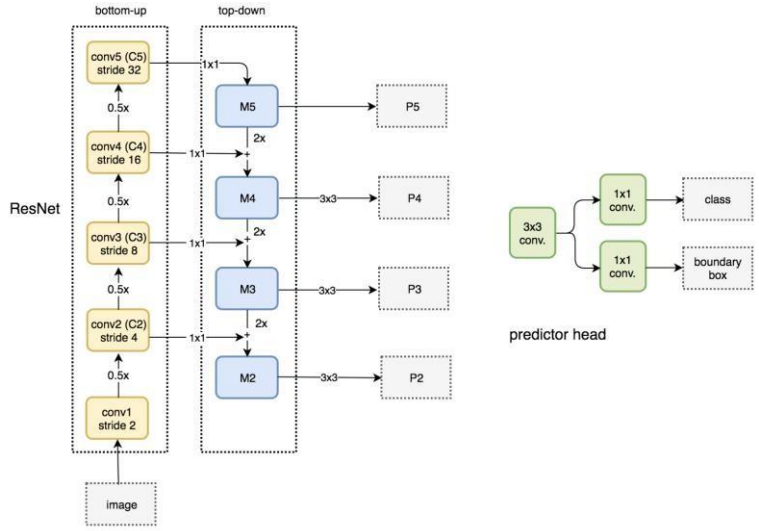
So finally, we can say that: ***Backbone = ResNet101/ResNet50 + FPN***.

Region Proposal Network (RPN):

A lightweight neural network scans the image in *sliding window fashion* and find areas that contains object. The regions that RPN scans over are called *anchors*. Regions of input image? Not quite! They are the regions of backbone feature map

that is generated by the FPN. By backbone feature map, we mean in the top-down level (depends if we are treating it as *M* layers or *P* layers. Here I mean *P* layers). This in turn allows RPN to reuse extracted features efficiently and avoid duplicate calculations.

This is how the RPN is integrated with FPN. After we feed top-down levels (I mean *P* layers) to convolutional layers, we call "those" layers, *RPN Head*.



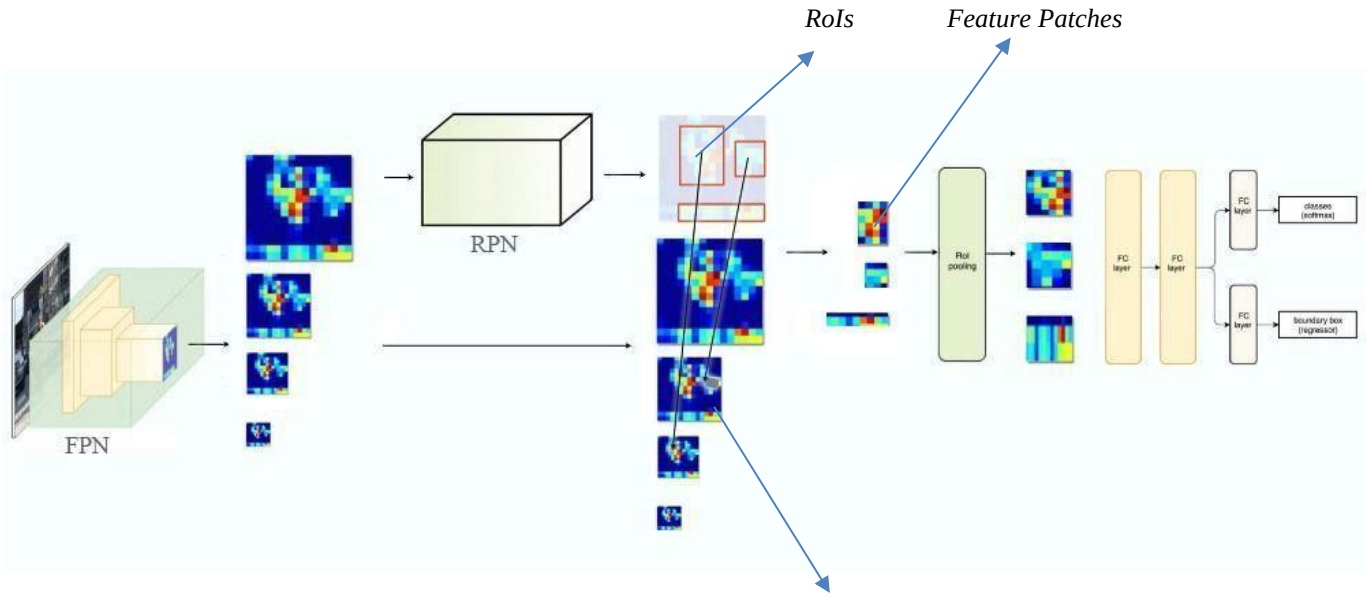
As we can see in the above image, that the RPN generates two outputs *for each anchor*:

1. **Anchor classes:** It is either foreground (*likely* an object in the box) class or the background class (no object) -- classifier.
2. **Bounding Box Refinement:** It lists foreground/positive anchors, which might not be centred as per the true object. So RPN estimates the deltas of the coordinates and dimensions of each positive anchor (normalized and in %) to fit the object better --- regressor.

Now, using RPN predictions, we pick top (say top k) anchors based on the scores (which are merely representative of the likeliness of containing an object). If too many boxes are there overlapping, we select the one having maximum score (foreground score) and discard the rest i.e. *Non Max Suppression*. The bounding boxes are also refined here dimension wise.

Class: *ProposalLayer*

This gives us the final proposals of RPN – *Region of Interests* (*RoI*) which are passed onto the next stage.



Selection of feature layer/levels based on RoIs

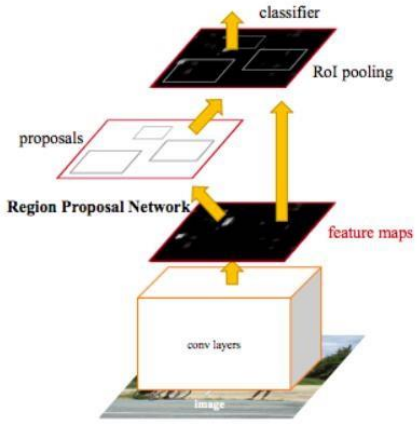
The formula to pick the feature layers of *P* based on *RoIs* is given as:

$$k = \lfloor k_0 + \log_2(\sqrt{wh}/224) \rfloor.$$

where

$k_0 = 4$

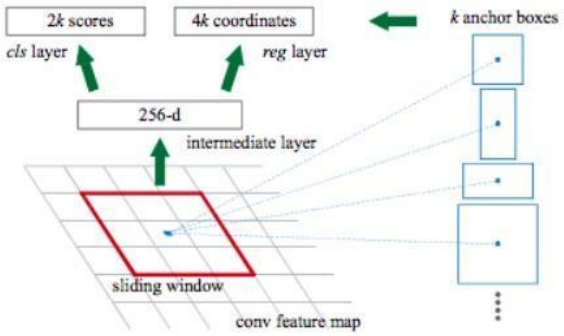
k is the P_k layer in the FPN used to generate the feature patch.



Credit: Original Research paper

This is how original RPN paper shows the proposals generation.

This below image is of RPN only:



Credits: Original Paper

For k, anchor boxes, we have 2k scores (foreground and background for each anchor box) and 4k coordinates (*w*,*h*,*x*,*y* deltas for each). This holds for 2D dataset only.

The labelling of anchors is done based on *Intersection over Union* (*IoU*) with the ground truth box and IoU>0.7 approximately. Now we need to train our RPN algorithm and for that, our RPN loss function will be:

$$L(\{p_i\},\{t_i\}) = (1/N_{cls})\times\Sigma L_{cls}(p_i,p_i^*) + (\lambda/N_{reg})\times\Sigma p_i^* L_{reg}(t_i,t_i^*)$$

Loss Function

i → Index of anchor, p → probability of being an object or not, t →vector of 4 parameterized coordinates of predicted bounding box, * represents ground truth box. L for cls represents Log Loss over two classes.

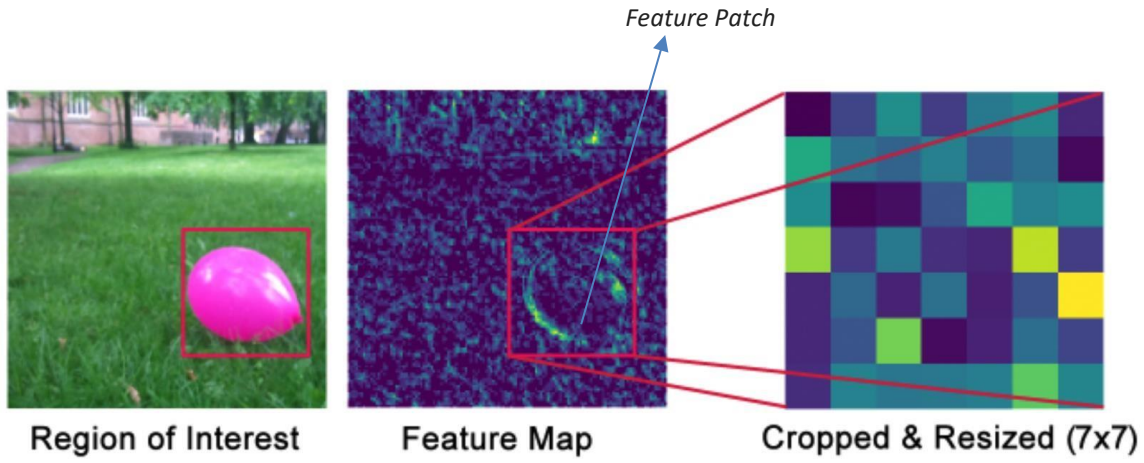
$$L_{reg}(t_i,t_i^*) = R(t_i - t_i^*)$$

p* with regression term in the loss function ensures that if and only if object is identified as yes, then only regression will count, otherwise p* will be zero, so the regression term will become zero in the loss function.

N_{cls} and N_{reg} are the normalisation. Default λ is 10 and is done to scale classifier and regressor on the same level.

ROI Classifier & Bounding Box Regressor:

Therefore, before we continue with the feeding of the *feature patches* generated via *RoIs* and *feature layer selection*, we do notice they differ in size. Since, the Mask-RCNN model has classification task, it requires all input sizes to be equal and our feature batches differ in sizes because of different sizes of *region proposals (RoIs)* after the bounding box refinement step in the RPN. In order to rectify this problem, *RoI Pooling* comes into play.



The feature map here is from a low-level layer, for illustration, to make it easier to understand.

Now, how to resize the feature patches? *Feature patches* are cropped parts of the feature maps selected based on the size of the *RoIs*.

RoI Pooling: Creating a *feature patch* and resizing it to a fixed size. The method to perform *RoI pooling* as per the current model is called *RoIAlign*, in which feature maps are sampled at different points and then *bilinear interpolation* is applied.

Class: *PyramidROIAlign*

Now these *refined feature patches* (although many literatures still say RoI) are fed into the fully connected convolutional deep networks which constitute the *Region-CNN* and it generates two outputs for *each feature path (or RoI?)*:

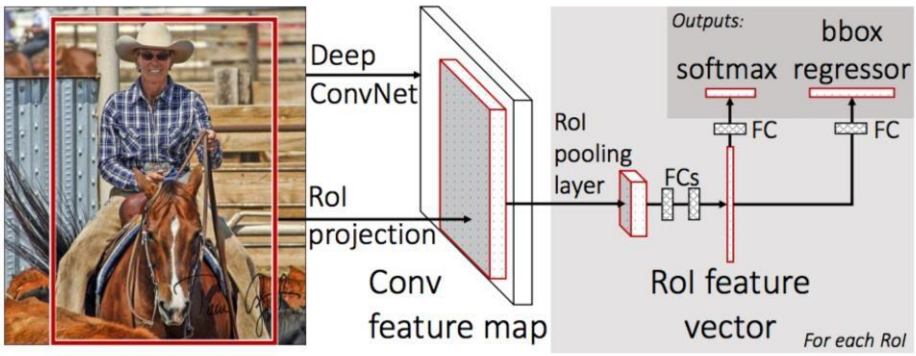


Illustration of stage 2. Source: Fast R-CNN (https://arxiv.org/abs/1504.08083)

1. **Class:** The class of the object in the RoI. Unlike the RPN it classifies the regions into the specific classes. If the class is a background class, then the RoI associated with it is discarded.
2. **Bounding Box Refinement:** Refines the location and the size of the bounding box to encapsulate the target object.

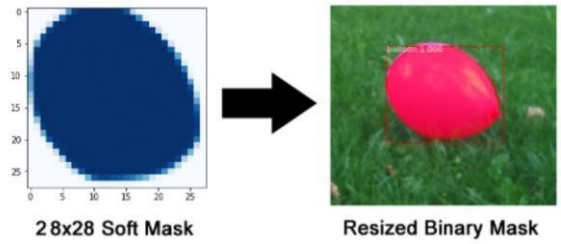
Function: *fpn_classifier_graph()*

This whole framework model is ***Faster R-CNN + FPN + RPN***.

Segmentation Masks:

Mask R-CNN = Faster R-CNN + Mask Network

The mask network (CNN network) inputs the positive regions selected by the *RoI classifier* and generates the *masks* for them. The generated masks are generally of low resolutions. They are originally the *soft* masks as in having float values and thus hold more details than the binary masks. As a final output, we rescale the soft masks to binary masks and resize them to the size of the *RoI bounding box*. This gives us final *segmentation masks* per object.



Function: *build_fpn_mask_graph()*

Approach to the problem

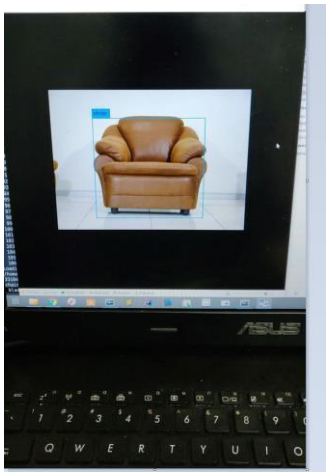
- **Original implementation:** The original implementation is that of the Mask RCNN as given in the matterport repository. I would first run the Mask RCNN code (again) as given in the repository for the sample training data of 2D images. The model already has pre-trained weights tested on the COCO dataset. We can do *transfer learning* and use these weights on our custom dataset, note the final loss and IoU.

  I ran the model on the balloon dataset on google collaboratory, which they already gave in the repository. Actually, the model was trained on COCO dataset and they used the same weights and retrained on the balloon dataset – *transfer learning*.

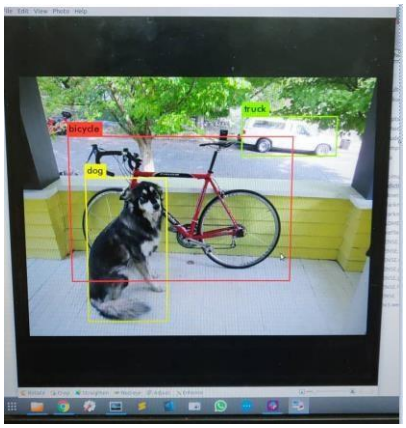
  The following were the results:

Proposals from RPN Final Detections and Masks

Then, I used the same weights trained on COCO dataset and used it to predict bounding box in a custom image without any retraining. The following was the result:



Luckily, the COCO dataset readily contained this class so I did not need to retrain.



I will try to train it from scratch on a new dataset and share the same with you soon.

- **Extend to 3D the core of the implementation:** The source code of the Mask-RCNN as per the matterport repository rests on the ResNet101 backbone (as written in the ReadMe). However, the source code they have used is adopted

from here (*https://github.com/fchollet/deep-learningmodels/blob/master/resnet50.py*) which is *ResNet50*. The code for the 2D images is already provided in the repository. I shall extend the same model to 3D images.

- **Transform RGB integer values (data type is float) to one gray value (float):** This one seems straightforward enough. It is asking to convert 0-255 integral-float values to 0-1 float values. Simple division of pixel values by 255.0 should suffice. In this case, it may not be needed because integral-float values are energy deposit in the cell and we need not re-normalize it. Tests and validation on such images should of course be carried out.
- **Extend to 3D the Feature Pyramid Network (FPN):** The Feature Pyramid Network is one of the most essential block of Mask RCNN. The model for 2D images is provided. I am extending it to 3D images. Then tests and validation of 3D images are carried out on this module. Basic gist is that, FPN generates featured layers that is fed into the RPN to generate ROI which then in turn generate feature patches.
- **Extend to 3D the Region Proposal Network (RPN):** I will have to extend the code involved in RPN module to 3D as well. RPN model generates ROIs which interact with top-down feature layers (feature-map collectively) to generate feature patches which are fed into the M-RCNN code.
- **Extend 3D to the remaining modules:** The remaining modules in the model.py file need also be extended to 3D and we will perform this very task and then test these modules against training and validation data and at last, test data.
- **Build synthetic events or tools to define bounding boxes and masks on real data to feed the training process:** It is indeed very important to define the bounding boxes and segmentation masks correctly and we need good tools for that.
- **Extend to other tools like visualization, evaluation:** This is also a very important task to achieve and evaluation and visualization are very essential part of training a model.
- **Training on simple cases:** First, we will test our extended model on simple dataset like 3D HGCAL images. We will also validate on this data.
- **Test on more complex cases:** Of course, otherwise our model is not at all generalized.
- **Optimize the efficiency of the whole process:** This of course is a necessity because a model needs to be efficient.

Schedule of Deliverables

Part I:

Summary of tasks:

- Create a *blog* showcasing progress on a daily basis.
- Run the original 2D implementation on the system.
- Properly set up the coding environment.
- Report consisting of brief and concise plan for the upcoming phases focusing mainly on *development of 3D modules*.
- Start converting 2D modules to 3D *independently* (the development strategy for validating new 3D module is discussed in next phase). ● Getting idea of how to create a custom *dataset.py* file in 3D.

In this phase in depth discussions of plans and their implementations will be done with the mentor prior to the actual implementation. This will be done via mailing lists/slack channels/skype. I will basically get to know the work done in the organization and the way it should be carried out. I will constantly keep the community active with the possible discussions focussing on the reasonable implementations to benefit the community as a whole. I will read about the Mask-RCNN and about the implementation in general and in more depth. The theory is as important as the implementation and without the correct foundations required, the coding will be nothing but a disaster. Before taking any step, I will *rectify* it with my mentor in advance before realizing it is too late.

As an initiative to show my progress, I will also create a *blog* showing progress on a daily basis. Reporting the small *report of progress* to my mentor on *weekly basis* is a good plan to avoid any confusion. I will also create a personal repository

on my github account for the same along with forking the original repository (already done).

As far as the implementation goes, during this period I will run the original matterport Mask RCNN implementation again on the 2D datasets of different complexity so that the image of the flow of code as per the module becomes vivid. I will compare the performances with some other models if necessary.

At the *end* of this phase, I will have a *final report*, which shows a brief coding plan for the next upcoming coding phases, which should mainly involve the conversion to 3D images. This should hold true for any phase. *I should (must!) have the results of the current phases as well as the brief plan for the next phase*. So finally, at the end of this phase I will have:

- Implementation results of the original 2D model on different datasets of different complexities.
- Blog, which tells the work, I would have accomplished in total.
- Brief, crisp and cogent plan for the next coding phase that will focus on the conversion of model to 3D images (only development).
- Most importantly, my coding environment should be properly setup by then!

Coding Schedule:

Part 2:

Summary of tasks:

- Choose (or design) a suitable development strategy to debug new 3D modified modules (discussed ahead).
- Following this very strategy, convert *at least* 80% modules (up to RPN) for 3D images *without any compilation errors.*
- Make sure that modified modules at least give us some *sensible* outputs on simple test 3D images.
- Update *utils.py* and create a custom dataset loader for 3D images.
- Brief plan for the next phase.

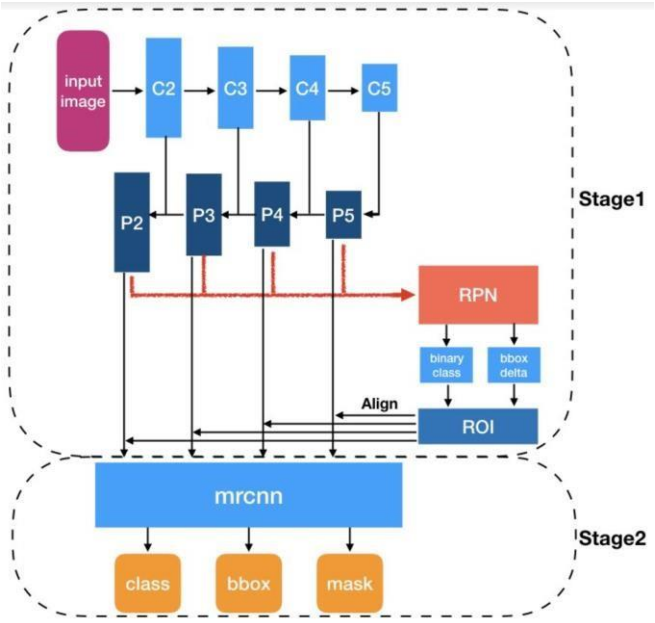
This phase will be a ***development phase***. This period will focus on the conversion task of at least some of the modules done right. The aim of this phase will be to

convert at least 80% of the modules for 3D images *without any compilation errors*. It is indeed very difficult to eliminate all the compilation errors first because I have never seen a code which does not give a compilation error at the start. Specifically at least ResNet backbone, FPN and RPN modules should be updated for the 3D images and these modules should compile without any errors. I should also make sure to test these modules for simple input images of a very simple dataset first. I must make sure that the outputs of FPN modules (feature maps) and RPN module (RoIs) make sense and their dimensions consistent. Note that this is not the only work, but the *minimum work* that has to be accomplished.

If the time permits, I will also update the Mask-RCNN network and the RoI pooling (and also anchor selection) to 3D images as well. They should also compile (just compile) without any errors. Then test the outputs on very small set of images and see if it makes sense (not even makes sense but merely outputs something). Lastly if there is still time (which is not likely), I will update the whole model and make sure it at least compiles (forget execution for now!).

Development Strategy:

We need to have a suitable development strategy to debug new 3D module. See the below image for the clear picture of how data flows in the model:



We will follow following points:

- **Data loader:** First, we need to know how to load data in 3D. The best way to proceed according to me is to copy the code written in *coco.py* (already provided with a 2D dataset) and update each function according to the 3D dataset.

- **2D slicing from 3D:** If we are moving step by step to convert module in 3D, we need to convert each module in 3D and leaving rest modules as it is and make sure that the model still compiles. To do this, we can follow this code of keras:

```
... # some layers, probably a Flatten() included
fc_out = keras.layers.Dense(64)(previous_layer)
fc_out_reshaped = keras.layers.Reshape([32, 2])(fc_out)
... # now you can do 1D conv, or RNN, or whatever you want
```

- Convert the ResNet backbone to 3D and pass the 2D slice of the output to FPN.
- Similarly, convert FPN to 3D and pass 2D slices of *P* layers to RPN.
- We need to keep moving forward step by step till the MRCNN and keep on compiling the model at every step of conversion and test it with sample images (mix of 2D/3D model).
- **Remember:** Instead of testing with actual images, it is always better to create a *random 4D matrix* of values 0-255 (*width, height, depth, channels*) and feed it as input. This can be easily done using *numpy*.
- Lastly keep the copy of both 2D models and 3D models and try to link those two while debugging since the inputs of 2D models will be projections of 3D models (will have to try this part).

Therefore, at the end of this phase I will have:

- Model with at least 80% (up to RPN) modules extended to 3D images and which compile without any errors.
- Test outputs on very minimalistic inputs and see if the outputs actually give us something.
- Brief plan for the next phase (clear and concise) – execution strategies to get reasonable metric.
- Same things to be incorporated in the blogs as well.
- Choose 2-3 datasets (simple datasets, not too complex) to test the model in the later stages.

Part 3:

Summary of tasks:

- Conversion of all the modules to 3D images using the development strategy given in the previous phase. (This will take half of this phase).
- Conversion of *configuration* files and *dataset* files as per 3D images as well and we should update the configuration class files as per our dataset and *pre-process* the input images if necessary.
- Compile and run the model on 2-3 simple datasets chosen in the previous phase.
- Train the model on these datasets and get training *IoU* of at least *0.6* and validation IoU of *0.5*.

Brief steps of training a neural network model:

- Try to train on small and simpler datasets first.
- If it does not work, try to debug the model, layer by layer until the model is correctly configured.
- Try for the complete dataset.
- Always eliminate *underfitting* before eliminating *overfitting*.
- Repeat the debugging steps until both are eliminated. ● For details, see the following rules.

- Using these rules, push the *training IoU* to *at least 0.7* on the simpler dataset (on the 2-3 sets we chose in previous phase) first. *Validation IoU* metric should be *at least 0.6-0.7* by now. This is the minimum accomplishment to be expected at the end of this phase.

This phase is a ***development and execution*** phase. In previous phase we would have completed at least the conversion to 3D images of 80% of the modules (up to RPN at least). If the conversion of the whole model (every module) is already finished in the previous phase, it will be a very good advantage. I am not specifying the detailed codes for conversion to 3D of every module here; I will do so in these phases, where I will specify in report in depth about the plan to be carried out in the next phase.

Nevertheless, if the conversion work is pending, I would take about 2 *weeks* to convert the whole model for 3D images and test each modules on simple images just to see if the output actually gives us *something* which makes little bit of sense.

At most, by 13-14th July, we should have our model extended to 3D images and it should be able to *compile without any errors*. According to me, the anchor selection is one of the part where the conversion to 3D images is not easy, so it would be better if the majority of the conversion part were completed in the

previous phase itself. Further, we would also have a model by now in which each of the module compiles and gives *some output* for each of the simple test image.

Now comes the *implementation* part. We would choose any one of the three datasets we selected at the end of the previous phase (better to select the *simplest*) and try to train our newly modified model on that dataset. Instead of straight away training on the whole dataset, we will follow these rules presented in Annex section (please click on the link).

At the end of this phase, I will have:

- M-RCNN model which runs successfully on 2-3 simple datasets of 3D images giving training IoU of at least 0.6 and validation IoU of at least 0.5.

- A report showing the work accomplished and incorporated in the blog and the brief and concise plan for the next phase.

- Choose 1-2 datasets of medium and high complexity (very high similarity among them, imbalanced and very large number of classes) to train the model in the next phase.

- The debugging steps I have presented in the Annex may be used in all the coding phases.

Part 4:

Summary of tasks:

- Boost the training and validation *IoU* of the 2-3 simple datasets to at least *0.75-0.85*.
- The training and validation IoU of the complex datasets are expected to be around *0.6*.
- At the end, the final report has to be submitted.

This phase is a ***debugging*** phase. In this phase, we will train, validate and test our model for the two datasets we chose at the end of the previous phase for medium to high complexity. This is going to be short because I have already explained the *debugging steps* in the previous phase evaluation. The aim of this phase is to boost

the *training* and *validation IoU* of the 2-3 simple datasets which we selected before to at least 0.75-0.85 and the training and validation IoU of the 2 complex datasets should be at least 0.6.

In order to do this we will follow the same steps 1-16 of the "training the model" in Annex but for these two complex datasets.

After this phase, I will have:

- Working model M-RCNN for 3D images which performs well on simple datasets (*0.8 IoU*) and reasonable on complex datasets with still the scope of improvement

- *Final report* illustrating all the work done till date and marking the success of GSoC. If the time remains, I will create a report consisting of brief plan of the upcoming *optional phase*.

- Blog --- completed for the above 3 phases.

- Submit this work to the organization in the final week.

Optional Part:

The optional phase will consist of the following works:

- *Segmentation* of 3D images on simple and complex datasets (categorical accuracy of at least 75% on simple and 60% on complex datasets).
- Train and adapt parameters and improve the validation metric on complex datasets to 0.8 IoU.
- Build 3D visualization tools that display the weights and biases after each step and create histograms on that.
- Changing the ResNet backbone to *ResNet150* and see if it makes any difference and why.
- Change the FPN and RPN algorithm using the physics knowledge (the detector)
- Add a regression module (to evaluate the energy for instance)

After this phase, I will have:

- Blog illustrating everything.
- M-RCNN model optimized for 3D images segmentation.
- M-RCNN model optimized for complex datasets.
- Working Regression Module to calculate energy instance of a cell.
- Modified FPN-RPN algorithm as per the detector.

Annex

Training, Validating, Evaluating, Debugging and Testing a Neural network Model:

Note: *This section is solely for reference and should be in no way considered as the main part of this proposal.*

1. Make sure the dataset is in proper format. Use *VGG Image Annotator* to set the outputs in proper format. Divide the dataset into training, validation and test set. Start with the ratios 80:10:10 then proceed to 90:5:5 and finally to 98:1:1. Make sure the model we are training, we should update the *configuration class files* as per our dataset and *pre-process* the input images if necessary.

2. **Training without Validation:** We have to make sure that the model first trains correctly rather than validating it. Doing training and validation side by side at the beginning is a misstep. To start the training, please see the below points.

3. **The Golden Rule:** First rule is to check whether our model is *structured properly*, meaning our model works if at all. To check that, we train it on a *single data point* (single image). Now if our model is correct and sensible no matter how inefficient it is, it should at least overfit that single data point without taking much time. If *not*, proceed to step 5 else proceed to step 4.

4. Train the model for 5 images and then proceed to 10 images without any validation. In both these cases, the model should *overfit*. It should train these small datasets.
If *no*: We have to try to debug the network. Please see the below point for that.
If *yes*: Skip to point 7.

5. **Debugging a Neural Network:** If these steps don't work after point 6, then scroll down and follow more steps to debugging – point 8 or so.

- **Is the input data not making sense?** Careless mistakes like replacing height with width and repeating same batches repeatedly will make the network model to *not learn*.

- **Try passing some random value:** If the error behaves the same way, then it is likely that the model is turning these random values to garbage at some point. Now this problem is *very difficult* to eliminate. We need to debug layer by layer or module by module. Please see the below point 6 for that.

- **Check data loader/generator:** We need to check that this code is actually loading the inputs to the network ResNet. Try printing the input to the ResNet to see if it is making any sense.

- **Is the setup of modules okay?** Please check the module code once again to see that they are actually connected with each other and are making sense.

- **Buggy Training Algorithm:** Is there in any calculations operations like division by 0 or logarithm of negative numbers? Eliminate it. Are the dataset pixel values integers instead of float32? Please change it.

- **Check the initial loss:** If it is very large, then this probably means that the *weight initialization* is bad. Change the weight initialization. If unsure, set it to *Xavier* or *He Initialization*.

- **Is backpropagation working?** To check if it is, implement the *gradient checking algorithm*. We can also output gradients using *keras backend* and see if it matches.

- **Try to change hyperparameters:** Try *grid search algorithm* or manually tune it.

- **Did you standardize the features?** Centred towards zero mean and unit variance. Please fix if not.

- Check the *learning rate*. Start high and go low. Follow exponential decrease with each epoch or use *KerasReduceLRonPlateau* callback to monitor training loss.

- **Try switching the optimizers:** Adam is fast but it's generalization error is higher than SGD with momentum. We will have to trial and error with these optimizers and generally either of these two can do the needful. So start with these two first. Then move to RMSProp.

- *Shuffle* the dataset. Remember, shuffle outputs in same order as the inputs.
  Use *train_test_split* with *shuffle=True* or *zip* function.

- *Batch Normalization* makes the training really fast. Please try to add it after each layer.

- Again, check the consistency of dimensions for each module. Keras *model.summary()* should help in this regard.

6. **Debugging Tips:**

- If there is *memory crash* due to large data, we can use tensorflow *TfRecords* or even read the data in batches if using Keras by creating a separate function/loader.

- Try to print images being read or output after each module using *print(x.eval(sess..))* if in tensorflow.

- Try printing out the *statistics* of the tensors.

- Wrapping tensorflow in keras is a friendly idea using *engineered_features* in Keras.

- Always try to print outputs at each layer using *keras.layers.lambda*.

- Add *tf.print* and *tf.summary* wherever possible to keep track.

- Use Tensorflow debugger (*https://github.com/Createdd/Writing/blob/master/2018/articles/DebugTFBasics.md#5-use-the-tensorflow-debugger*).

- Use keras backend to print values after each layer:

```
from keras import backend as K

inp = model.input                                           # input placeholder
outputs = [layer.output for layer in model.layers]          # all layer outputs
functors = [K.function([inp, K.learning_phase()], [out]) for out in outputs]    # evaluation

# Testing
test = np.random.random(input_shape)[np.newaxis,...]
layer_outs = [func([test, 1.]) for func in functors]
print layer_outs
```

- Consider a visualization library like *Tensorboard* and *Crayon*. In a pinch, we can also display weights/biases/activations.

- Layer updates should have approximately Gaussian distribution.

- Try to follow this rule: *For weights, the histograms should have an **approximately Gaussian (normal)** distribution, after some time. For biases, these histograms will generally start at 0, and will usually end up being **approximately Gaussian.***

7. Try out the *opposite golden rule*: Keep the full training set along with the validation set this time. Now just shuffle the outputs only (of training not validation) and then train the model on this set. We should have

training loss decreasing very slowly and validation loss extremely high and random. If this does not happen, repeat steps 5,6. If still not, move to more debugging steps – step 8. If this does happen, skip to step 9.

8. **Debugging a Neural Network II:** If this does not work, either move to step 9, or try these steps 5,6,8 again and look more closely. If they work, go to 9.

- **Unit Testing:** Again, please break the code into modules and check again if it gives expected output. I will follow this guide: https://medium.com/@keeper6928/how-to-unit-testmachine-learning-code-57cf6fd81765

- We can follow this library for unit testing but it is only in TensorFlow for now: https://github.com/Thenerdstation/mltest

- **Eliminate Exploding/Vanishing Gradients:** To overcome exploding gradients, we can use *gradient clipping* or we can decrease learning rate. For latter case, we can use *Relu* or *Leaky Relu* or we can increase learning rate.

- **Overcoming NaNs:** Try to decrease learning rate, or check for any calculation errors. Finally, we need to follow step 6 to debug layer by layer.

9. Take the whole training set this time along with the validation. Make sure that they are shuffled correctly and pre-processing is done on all the sets in the same manner. For now, do not use callbacks/checkpoints. Now train the model on the dataset. We should have either of these cases:
Training loss is not decreasing: This is clearly the case of *underfitting*. Go to step 10 then 12 to eliminate this problem.
Training loss is low but the validation loss is high/random: This is clearly the case of *overfitting*. Skip to step 11 then 12 to eliminate this problem. *If no issues*: Skip to step 13.

10. **Debugging a Neural Network III – eliminate *Underfitting***:

- **Is the dataset imbalanced?** If yes, then replace the current loss function with the *weighted loss function*. Then run the training process all over again.

- **Are training examples sufficient?** Try to change 80:10:10 ratio to 90:5:5 and finally to 98:1:1 and see if it improves anything.

- **Reduce batch size:** This can generalize the model better but then there will be risk of overfitting.

- **Reduce too much Data Augmentation:** See if it helps in any way.

- **Make the model complex:** This we can do by *increasing the number of hidden layers* and test it on each module (see module based debugging). Try to*increase number of hidden units* as well although former is more effective.

- **Reduce Regularization if it is too much:** It decreases overfitting but too much can cause underfitting. Try to reduce dropout, L1/L2 regularization.

- Verify the metric as well as loss function and see if it makes any sense.

- Eliminate the 'dying relu' problem by replacing it with 'leaky relu' or Prelu.

- Manually stop the training, change the learning rate, and then continue it just to avoid *local minima*.

11. **Debugging a Neural Network IV – eliminate Overfitting:**

- **Increase batch size:** Less generalization.

- **Increase data augmentation:** It has regularization effect.

- **Increase dropout/L1/L2 regularization:** Again, it prevents weights and biases from behaving erratically.

- **Make the model simpler:** Remove some layers and hidden units.

- Debug layer by layer to check the behaviour of weights and biases and then manually tune them (most difficult).

12. **Module based Debugging:**

- **Debug ResNet Backbone:** This is the bottom up layer and we need to tune the hidden layers in this and also check if its giving correct output.

- **Debug FPN:** This constitutes the top-down layer and we need to tune it in the same way as we have tuned the ResNet backbone. Check the outputs of this also. Check if there are lateral connections also.

- **Debug RPN:** Try printing *rpn([p])* just to check that the inputs are going into the RPN head. Try adjusting number of layers in this network also.

- **Debug MRCNN as a whole:** Debugging is checking every layer of this network and printing outputs from it to see if it works.

- **Anchors Output:** Check if the anchors outputted are making sense in any way and are refined properly.

- **Apply steps 10 and 11 on each of the module.**

13. **Final Tips:** Finally train and validate the model on the whole dataset and making use of *callbacks* such as *early stopping*, *model checkpoint*, *Reduce LR on Plateau* by monitoring validation loss. We can use a *cyclic learning rate*, which is said to be the best scheduler.

14. If none of the above steps work, I will try the above debugging steps once again. Finally, I will contact my mentor for further help or search for the help on communities like *stackoverflow* or *github*.

15. Try to push the *IoU* (intersection over Union) metric of training set and the validation set.

16. Now run the model on the testing set and see if it predicts properly. The prediction accuracy should be close to validation set as of now. It *should*.