CNN Model and Associate Function Definition

In this document I will give a walkthrough guide for creation of the CNN models in the current framework. The guide will be based on the model in the GitHub repository at:
HAhRD/GSOC18/models/model1_definition.py

Step 1: Importing CNN modules

First of all we import the required layers from the 2D convolution from **CNN_Module.utils.conv2d_utils** and layers for 3D convolution from the **CNN_Module.utils.conv3d_utils.**

These module will give access to the following layers for model creation:

1. **fully connected layers**
2. **conv2d layers** with optional relu rectification,batch normalization,dropout and weight decay inbuilt into it.
3. **max pooling layers**
4. **Residual layer** with two flavours (identity and convolutional type)
5. **Inception layers**
6. **3D counterpart of all the CNN related layers above.**

```
1  import tensorflow as tf
2  import sys
3  import os
4
5  #Directly using the package utilities
6  from CNN_Module.utils.conv2d_utils import *
7  from CNN_Module.utils.conv3d_utils import *
```

Step 2: Creation of Convolutional model

Now we are ready to define the model using this layers on the input 3D image of the detector. Any model will take two input arguments:

1. **X:** the full 3D image of the hits in the detector.
2. **is_training:** the flag which describe the whether we are in training state or validation/inference state. Since depending on this the nature of Batch Normalization and dropout layers will change internally.

```
719  def model6(X,is_training):
720      '''
721  >   DESCRIPTION:...
732  >   USAGE:...
735      '''
736      #Model Hyperparameter
737      bn_decision=False
738      lambd=0.0
739      dropout_rate=0.0
740
741      #Reshaping the image to add the channel dimension
742      X_img=tf.expand_dims(X,axis=-1,name='add_channel_dim')
743
744      #Starting the first layer
```

Then we decide on the hyperparameters of the model. They are:

1. **bn_decision:** whether we want to apply the batch normalization or not.
2. **lambd:** this is a scalar that will be multiplied to the L2-regularization of the parameter for controlling the variance in the model.
3. **dropout_rate:** a number between 0-1, which specify the rate of dropout in the model.

Now, since we will be doing 3D convolution, we will add a dimension to the end of image. This will be like the channel to the image (like RGB, but only one here).
So, at this point the shape of the image will be : [batch_size, resolution_x, resolution_y, 40, 1]

Step 3: Adding the convolutional layers

Now we will start with the creation of the convolution layer. We will follow the pattern of AlexNet (2012 ImageNet Challenge winner) in this model in the x,y spatial dimensions of convolution.

So we create the first convolutional layer by the **rectified_conv3d** layer present in the conv3d_utils script. This layer take the following argument:

1. **X:** the 3D image input to this layer
2. **name:** to give a unique name (and also create unique variable scope internally)
3. **filter_shape:** the shape of the 3D convolution filter of dimension [fx,fy,fz]. Here we took all the 40 layer in the z-dimension for the convolution.
4. **stride:** the stride in all the three dimension [sx,sy,sz]
5. **padding_type:** the type of padding we want to apply to the input of this layer. (SAME/VALID)
    a. **SAME:** means that the output shape will be same as the input shape.(padded with zero)
    b. **VALID:** no padding will be applied

6. **is_training**: the training flag to know whether we are in training or testing mode.
7. **dropout_rate**: the hyperparameter to control the dropout in the layer.
8. **apply_batchnorm:** a boolean to specify whether we want to apply batch norm or not.
9. **weight_decay:** a scalar to control the L2-Regularization of layer.
10. **apply_relu:** whether we want to apply relu rectification to the output of this layer.
11. **initializer:** (optional) The initializer function handle to specify which which initializer we want to use to initialize the variables. Default to **tf.glorot_uniform_initializer().**

```
744     #Starting the first layer
745     A1=rectified_conv3d(X_img,
746                         name='conv3d1',
747                         filter_shape=(11,11,40),
748                         output_channel=96,
749                         stride=(4,4,1),
750                         padding_type='VALID',
751                         is_training=is_training,
752                         dropout_rate=dropout_rate,
753                         apply_batchnorm=bn_decision,
754                         weight_decay=lambd,
755                         apply_relu=True)#it is default no need to write
756     A1Mp=max_pooling3d(A1,
757                         name='mpool1',
758                         filter_shape=(3,3,1),
759                         stride=(2,2,1),
760                         padding_type='VALID')
761
762     #Defining the second layer
763     A2=rectified_conv3d(A1Mp,
764                         name='conv3d2',
765                         filter_shape=(5,5,1),
766                         output_channel=256,
767                         stride=(1,1,1),
768                         padding_type='SAME',
769                         is_training=is_training,
770                         dropout_rate=dropout_rate,
771                         apply_batchnorm=bn_decision,
772                         weight_decay=lambd,
773                         apply_relu=True)#it is default no need to write
```

Now after the convolutional layers is defined, we will apply a max pooling layer to the output of the last layer.This 3D max pooling is implemented by the **max_pooling3d** layer present in the same conv3d_utils script.It takes the following arguments:

1. **activation:** the output from the previous layer.
2. **name:** the name of the layer (will be used by tensorboard for proper visualization)

3. **filter_shape:** the shape of the filter for max pooling of form [fx,fy,fz].
4. **stride:** the stride of the max pooling operation [sx,sy,sz]
5. **padding type** : whether we want to have VALID or SAME padding.

Step 3.1: Additional layer: fully connected layer

So following this same pattern we could construct the full model. Now some more layer are available for use so I will describe the arguments they take as input and some comments on when and how to use them.
So the fully connected layer will be used in last stage of the model pipeline when we have a high-level representation of the input image and it will not be very computationally expensive to add the full connection between the two layers instead of the convolution (which is local in nature).

```
855     #Now flattening and making fully connected layers
856     A8=simple_fully_connected(A7Mp,
857                               name='fc1',
858                               output_dim=1024,
859                               is_training=is_training,
860                               dropout_rate=dropout_rate,
861                               apply_batchnorm=bn_decision,
862                               weight_decay=lambd,
863                               flatten_first=True,
864                               apply_relu=True)
865
866     #Now making the socnd fully connected layer
867     A9=simple_fully_connected(A8,
868                               name='fc2',
869                               output_dim=1024,
870                               is_training=is_training,
871                               dropout_rate=dropout_rate,
872                               apply_batchnorm=bn_decision,
873                               weight_decay=lambd,
874                               apply_relu=True)
875
876     #Now making the final layer (un-rectified version)
877     Z10=simple_fully_connected(A9,
878                               name='fc3',
879                               output_dim=6,
880                               is_training=is_training,
881                               dropout_rate=dropout_rate,
882                               apply_batchnorm=bn_decision,
883                               weight_decay=lambd,
884                               apply_relu=False)
885
886     return Z10
```

This layer takes the following argument:

1. **input:** the input from the previous layer output.
2. **name:** a unique name given to this layer
3. **outptput_dim:** the output dimension of this layer
4. **is_training:** the training flag to specify whether we are in training or testing mode.
5. **dropout_rate:** the rate of dropout in this layer (a real number between 0 and 1)
6. **apply_batchnorm:** whether we want to apply batch normalization or not
7. **weight_decay:** scalar that will be multiplied to the L2-regularization to control the overfitting/variance problem.
8. **flatten_first:** when this layer takes the input from the output of the convolution layer, then this argument will be used to first flatten the output into a one dimensional vector before applying the linear transformation. (set **True** to use it)
9. **apply_relu:** whether we want to apply the relu activation to output of this layer. One **important remark**: Do not normalize the the output of the last layer of the model. This will be done (if required) separately in the cost calculation function since there are other regression variable in the output also.

Step 3.2: Additional layer: Residual Identity-Layers

This layers provide the functionality for the residual layers in the model and is based on the paper https://arxiv.org/abs/1512.03385. These layers will be particularly useful when we will go for more deeper network like the~20-50 layers or even beyond. There are two flavours

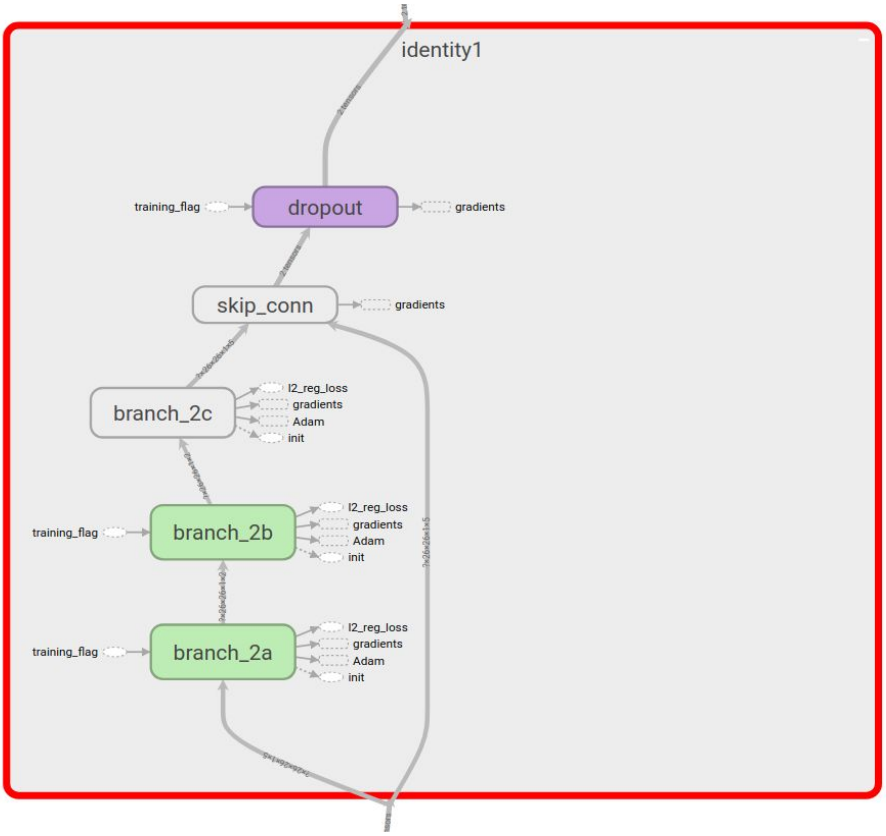
1. Identity Residual Block
2. Convolutional Residual Block

In this section we will discuss about the **Identity Residual Block**.This layer consist of two branches. The input is from the bottom of the picture.The branches are

1. **Main branch:** The branch which contain block named as branch_2x.
2. **Shortcut/Skip branch**: The one which have a direct connection from bottom to a node named as skip_conn above.

Steps of Computation:

1. The **first node (branch_2a)** in Main branch is a convolutional block which perform 1x1 convolution. The main purpose of this is to decrease the number of channels so that computationally expensive filters like 3x3 or 5x5 could be performed on smaller number of channels.
2. Then the **mid node(branch_2b)** is the place where actual convolution with filters like 3x3 or 5x5 happens.
3. Then we finally increase the number of channels as it was at input in **branch_2c node**, again using one-one convolution.
4. After that we add both the main branch and the shortcut branch together element wise in the skip_conn node of graph.



Now we will define this layer using the **identity3d(or 2D)_residual_block** defined in the conv2d/3d_utils script.

```
#Passing the activation to the fourth layer which is identity
A4=identity3d_residual_block(A3Mp,
                             name='identity1',
                             num_channels=[2,2,10],
                             mid_filter_shape=(3,3,3),
                             is_training=is_training,
                             dropout_rate=dropout_rate,
                             apply_batchnorm=bn_decision,
                             weight_decay=lambd)
```

This layer take the following arguments:

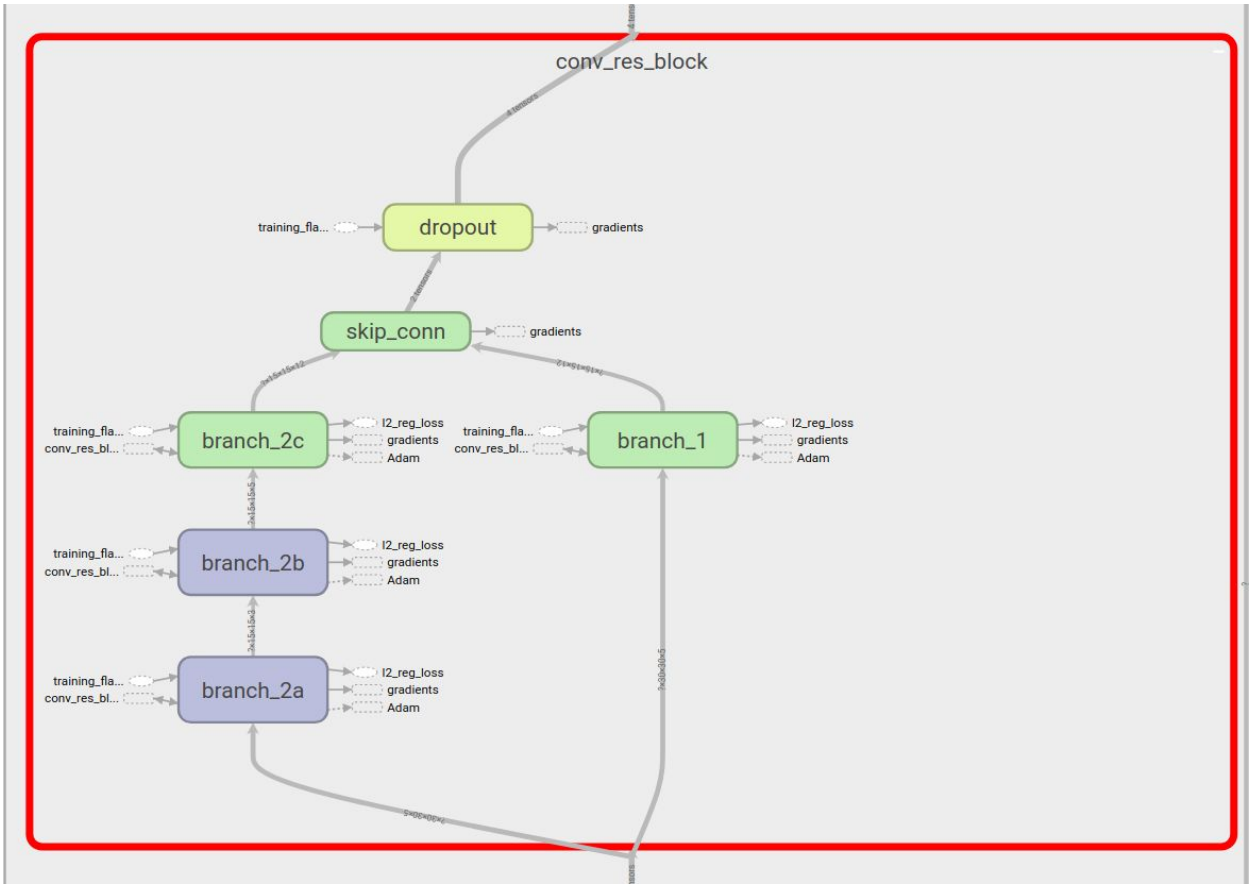
1. **input :** the input from the previous layers output.
2. **name:** a unique name given to this layer.
3. **num_channels**: As you can see in the figure, the main branch has three different intermediate node. Each of them is a convolution layer as described above. So, here we specify the number of output channels in branch_2a,branch_2b,branch_2c node as a list.

4. **mid_filter_shape**: since the branch_2b node does the convolution with user specified filter, we have to specify the filter shape. For 3D convolution specify [fx,fy,fz] and for 2D convolution specify [fx,fy].
5. **is_training**: same a described in earlier layers

Other arguments to this layer are also same as described in previous layers.

Step 3.3: Additional layer: Residual Convolutional-Layers

This layer is a slight modification of the previous identity residual block. All the components are similar to the Identity Residual block except for one difference in main branch (brach_2a block) and corresponding block in shortcut branch.



The difference come from the fact that a **variable stride** with “VALID” padding is allowed in the **branch_2a block** which shrinks the image height and width. So, to compensate for that we have to make same adjustment in the shortcut (branch_1) block.

Now we can use this layer by using the **convolutional3d_residual_block** defined in the conv3d_utils.py. Also, the 2D version of this layer is available in conv2d_utils.

```
#Passing through the fifth layer
A5=convolutional3d_residual_block(A4Mp,
                                  name='conv_res1',
                                  num_channels=[2,2,15],
                                  first_filter_stride=(1,1,1),
                                  mid_filter_shape=(3,3,3),
                                  is_training=is_training,
                                  dropout_rate=dropout_rate,
                                  apply_batchnorm=bn_decision,
                                  weight_decay=lambd)
```

The different argument from the previous identity-block is:

1. **first_filter_stride:** this gives the stride of convolution in the first brach_2a block. Also internally this same stride is also given to the branch_1 block.

Rest of the arguments are same as the identity residual block.

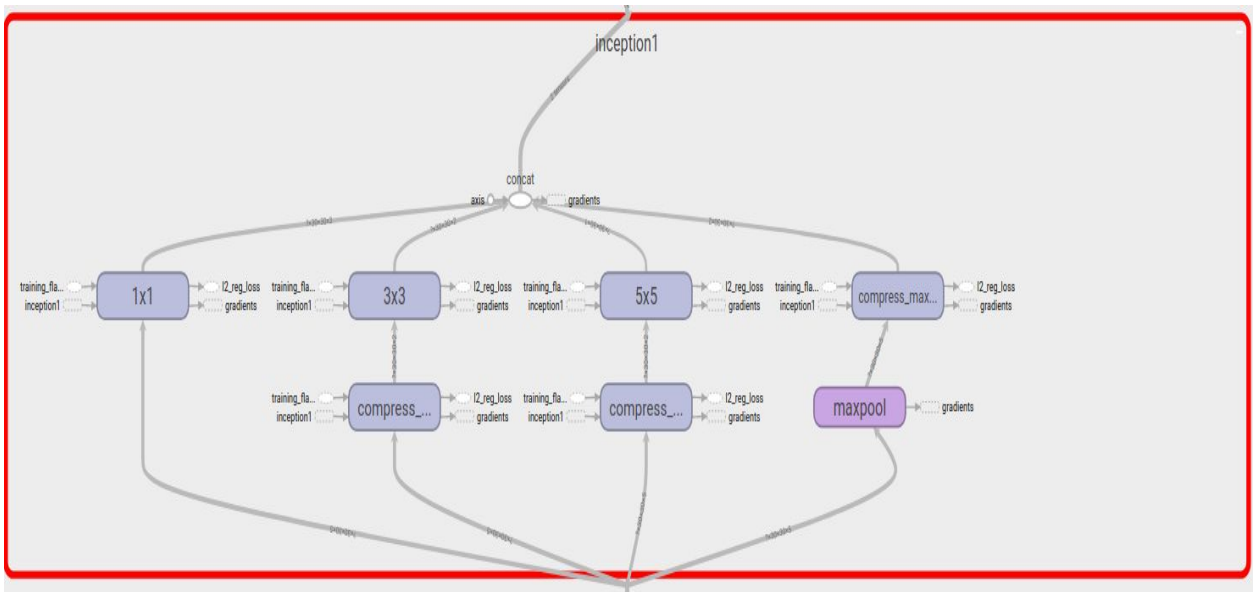
One remark: Prefer to use the identity_residual block than this block as that will help in better flow of gradient hence faster learning in deep networks.

Step 3.4: Additional layer: Inception block

This layer is based on the paper https://arxiv.org/pdf/1409.4842.pdf (Going deeper with Convolution). This layer implements the idea of using filters of different shape for the same input to extract the information from input with different level of resolution and then stacking all those together to give output to the next layer.

The current inception layer has fixed structure (newer inception layer could be added later). The following computation is done in this layer:

1. The first branch (leftmost edge) is simple (1x1 for 2D) / (1x1x1 for 3D) convolutional layer.
2. In the second branch we have 1x1 convolution first to decrease the number of channel and then (3x3 for 2D) / (3x3x3 for 3D) convolution.
3. Similarly again we do the 1x1 / 1x1x1 convolution to decrease the number of channel (referenced as **compress block**) before any computationally intensive convolution. Then we do the 5x5 / 5x5x5 convolution.
4. The last branch is doing max-pooling first with “SAME” padding and then compressing the number of channels in the “compress max-pool block” by 1x1/1x1x1 convolution.
5. Finally after all the branches are computed we concatenate them into single “image” for input to the next layer of the network.



A dynamic version of this layer will be added later which will let user to specify the filter shape of these intermediate layer instead of the fixed ones.

We could use this layer by inception_block/inception3d_block defined in conv2d/3d_utils.

```
192
193     A5=inception_block(A3,
194                        name='inception1',
195                        final_channel_list=[3,2,1,2],
196                        compress_channel_list=[2,2],
197                        is_training=is_training,
198                        dropout_rate=dropout_rate,
199                        apply_batchnorm=bn_decision,
200                        weight_decay=lambd)
201
```

This layer takes the following arguments:

1. **input:** The input tensor to this layer.
2. **name:** name of this block for visualization in Tensorboard and also act as variable scope.
3. **final_channel_list**: the number of channels in the output of each branch before concatenation. [1x1 / 1x1x1 channels,3x3 / 3x3x3 channels, 5x5 / 5x5x5 channels, maxpool output channels]
4. **compress_channel_list**: this is for the compress blocks before 3x3 and 5x5 channels. The number of this channel output for this compress block is decided by this this. [compress output for 3x3 conv, compress output for 5x5 conv]

Step 4: Defining the Cost Function

Now once we have completed the the creation of model which return a vector as output, we have to define how we are going to calculate the cost/loss of the model for the optimizer to perform backpropagation.

So, each model has to be accompanied by a cost_function which take in three inputs:

1. **Z:** it is the **unnormalized output** of the model. Generally is of shape [batch_size,output_vector_length].
2. **Y:** the target/label/truth value for the model to predict. Shape is same as Z.
3. **scope:** just use as it is in the calculation of L2-loss.Don't do anything else with it.

```
78  def calculate_total_loss(Z,Y,scope=None):
79      '''
80  >   DESCRIPTION:...
91  >   USAGE:...
101     '''
102     with tf.name_scope('loss_calculation'):
103         #A list to combine all losses
104         all_loss_list=[]
105
106         #Calculating the L2_loss(scalar)
107         reg_loss_list=tf.get_collection('all_losses',scope=scope)
108         l2_reg_loss=0.0
109         if not len(reg_loss_list)==0:
110             l2_reg_loss=tf.add_n(reg_loss_list,name='l2_reg_loss')
111             tf.summary.scalar('l2_reg_loss',l2_reg_loss)
112         all_loss_list.append(l2_reg_loss)
113
114         #Calculating the model loss
115         #Calculating the regression loss(scalar)
116         regression_len=4
117         regression_output=Z[:,0:regression_len]#0,1,2,3
118         regression_label=Y[:,0:regression_len]#TRUTH
119
120         #mean_squared_error Regrassion Loss
121         regression_loss=tf.losses.mean_squared_error(regression_output,
122                                                      regression_label)
123
```

Now we will step by step define the loss of the model by slicing off the output of model and the label.

1. **L2-Regularization loss**: This is the loss due to the L2-Regularization of the parameter in the model. You don't have to change this. Just use it as it is. This is kept here and not done internally to give the transparency in the total loss function.

2. **Regression Loss:** Since the part of the output will be used for regression, we will use the mean square error to calculate this error. Now, since here the first 4 elements of the vector is for regression,we are slicing the first 4 from the prediction and the label array.

3. **Classification Loss:** Now , the rest of the portion of the output vector is being used for the classification of particles, we will define a softmax cross entropy loss for them. We slice the output vector appropriately to calculate this loss.

```
131         tf.summary.scalar('regression_loss',regression_loss)
132         all_loss_list.append(regression_loss)
133
134         #Calculating the x-entropy loss(scalar)
135         class_output=Z[:,regression_len:]#4,....
136         class_label=Y[:,regression_len:]#TRUTH
137         categorical_loss=tf.losses.softmax_cross_entropy(class_label,
138                                                          class_output)
139         tf.summary.scalar('categorical_loss',categorical_loss)
140         all_loss_list.append(categorical_loss)
141
142         #calculating the total loss
143         total_loss=tf.add_n(all_loss_list,name='merge_loss')
144         tf.summary.scalar('total_loss',total_loss)
145
146     return total_loss
147
```

Finally we add all the losses, L2-regularization,Regression and classification loss to get the final total loss, which will be used by the optimizer to backpropagate.

Step 5: Defining the Accuracy of the Model

Now, we want to see the accuracy of the model, which might not be defined the same way as the loss for the model. This accuracy function has no role in training of the model. This is just for our inference to see the progress of training.
For example, sometime Percentage Error is useful for seeing the accuracy of the model and sometime just the absolute difference of the prediction and the label.
So this function provides the platform to define those metric which will be used for logging them to the tensorboard.
We calculate the Regression Accuracy by the following way:

1. Currently we define the accuracy by calculating the percentage error for all the component of regression. This is not the right approach for the psox/y/z component for which we should just the absolute difference.

2. First of all we slice the array one by one for each regression component and then use the appropriate function to calculate the mean error.
3. After that we add this error to the tensorboard by **tf.summary.scalar** function giving first the name and then the scalar value.

```
def calculate_model_accuracy(Z,Y):
    '''
    DESCRIPTION:...
    USAGE:...
    '''
    with tf.name_scope('accuracy_calculation'):
        #Regression accuracy calcuation as 100-%error
        #Energy_accuracy/error
        energy_abs_diff=tf.losses.absolute_difference(Z[:,0],
                                    Y[:,0],reduction=tf.losses.Reduction.NONE)
        energy_error=(tf.reduce_mean(tf.divide(
                                energy_abs_diff,Y[:,0]+1e-10)))*100
        tf.summary.scalar('percentage_energy_error',energy_error)

        #posx Accuracy/Error
        posx_abs_diff=tf.losses.absolute_difference(Z[:,1],
                                    Y[:,1],reduction=tf.losses.Reduction.NONE)
        posx_error=(tf.reduce_mean(tf.abs(tf.divide(
                                    posx_abs_diff,Y[:,1]+1e-10))))*100
        tf.summary.scalar('percentage_posx_error',posx_error)

        #posy Accuracy/Error
        posy_abs_diff=tf.losses.absolute_difference(Z[:,2],
                                    Y[:,2],reduction=tf.losses.Reduction.NONE)
        posy_error=(tf.reduce_mean(tf.abs(tf.divide(
                                    posy_abs_diff,Y[:,2]+1e-10))))*100
        tf.summary.scalar('percentage_posy_error',posy_error)

```

We then calculate the Classification Accuracy by finding the average correct prediction and then again add this to the tensorboard by tf.summary.scalar function.

```

        #Calculation of Classification error
        regression_len=4
        prediction=tf.argmax(Z[:,regression_len:],axis=1)
        label=tf.argmax(Y[:,regression_len:],axis=1)

        correct=tf.equal(prediction,label)
        classification_accuracy=tf.reduce_mean(tf.cast(correct,'float'))*100
        tf.summary.scalar('percentage_classification_accuracy',classification_a

    #Returning the accuracy tuple to be used in inference module
    accuracy_tuple=(energy_error,
                    posx_error,
                    posy_error,
                    posz_error,
                    classification_accuracy)
    return accuracy_tuple

```

Finally, it is important (with the point of view of the inference mode in training manager) to return all the accuracy metric value as a tuple.

So, that's is all needed for defining a full model for training. In summary we have to do the following things

1. Make the model using the appropriate function.
2. Define the model_loss_function for calculating the loss of model.
3. Define the accuracy calculation function for making inference on the performance of the model.