Description of PCRN code

# Datasets

The fist dataset used to train PCRN is the modified ShapeNet Core V2. This dataset initially contains 55 different types of objects in the form of 3D models (.obj). Our algorithm is designed to be used with a depth camera producing a grid-like output like an image containing depth information for each pixel. In order to be able to use the ShapeNet dataset with our algorithm, it was mandatory to generate the aforementioned type of data from the original dataset. Thanks to Panamari’s code [Stanford Shapenet Renderer](https://github.com/panmari/stanford-shapenet-renderer#stanford-shapenet-renderer), we were able generate, after minor modifications, a usable dataset for our PCRN algorithm. The generated dataset now constitutes of 55 labelled folders containing 6 different depth views of each objects. In total the dataset contains near 250 000 depth images.

The second dataset used to train PCRN is the modified the famous ModelNet40 dataset. This dataset initially contains 40 different types of objects in the form of point cloud files (.off). In order to be able to use the ModelNet40 dataset with our algorithm, it was mandatory to generate the depth image data from the original dataset. Again, thanks to Panamari’s code [Stanford Shapenet Renderer](https://github.com/panmari/stanford-shapenet-renderer#stanford-shapenet-renderer), we were able generate, after minor modifications, a dataset for our PCRN algorithm. The generated dataset now constitutes of 40 labelled folders containing 6 different depth views of each objects. In total the dataset contains near 75 000 depth images.

The detailed procedure to use those datasets goes as follow.

1. Download the dataset from their official source
2. On a Unix system, extract the datasets
3. Get the latest version of Blender

For ShapNet:

1. Navigate to the root folder of the dataset and copy the taxonomy.json as well as the jsonTolabels.py files from the GitHub’s ShapeNet folder at this location
2. Edit the paths from the jsonToLabels.py file to correspond to your setup and run the script to rename the folders to their class ID
3. Place a Blender shortcut into the folder containing all the class folders
4. Copy the render\_blender.py and the Dataset\_FileMover.py files from the GitHub’s ShapeNet folder to the same location
5. Run the render\_blender.py script to generate depth images for each model (this will take many hours)
6. Create a new folder names “ShapeNetCoreV2 – Depth” next to the root folder
7. Run the Dataset\_FileMover.py to move the images into that new folder while keeping the same structure: this new folders now contains what’s necessary to train our models

For ModelNet40 - Depth:

1. Install the “.OFF” addon extension from the GitHub’s blender-off-addon-master folder by selecting the “import\_off – modified.py” file when prompted (see [here](https://github.com/rlguy/Blender-FLIP-Fluids/wiki/Addon-Installation-and-Uninstallation) for a guide on how to install a blender addon)
2. Place a Blender shortcut into the folder containing all the class folders
3. Copy the render\_blender.py and the Dataset\_FileMover.py files from the GitHub’s ModelNet40 folder to the same location
4. Run the render\_blender.py script to generate depth images for each model (this will take a few hours)
5. Create a new folder names “ModelNet40 – Depth” next to the root folder
6. Run the Dataset\_FileMover.py to move the images into that new folder while keeping the same structure: this new folders now contains what’s necessary to train our models

To use these datasets in PyTorch, it is imperative to create a Dataset class. The DrawDataset file from the GitHub project contains this class. The class itself is named MyDataset and inherits form the PyTorch Dataset object. Only three methods are redefined in order to use this class.

1. def \_\_init\_\_ (self, data\_dir, transform = None):

In this method, we define a list of all the depth images in the dataset folder. We also define the transformations to be applied to each of them we they are loaded into the training procedure.

1. def \_\_getitem\_\_(self, idx):

This method is used to generate a label for each image based on their paths. Each path contains the label information at a precise location. Then, the image and the label are loaded and returned in a tuple.

1. def \_\_len\_\_(self):

This last method is only there to return the proper length of the dataset. It will be useful when loading the dataset into train, validation and test sets.

Similarly, the Modelnet40Dataset class has been created to use the Modelnet40 dataset.

The raw datasets contains the depth images with a pixel ranging from 0 to 1, with 1 being infinite. In order to be able to use the dataset, we had to reverse the meaning of these bounds. To do so we created a class object that performs this custom transformation.

class MyTransform(object):

This class simply subtracts one to every pixel before applying the absolute value function. We finally reshape the tensor into the right dimensions ([batch, channels, height, width]) using the unsqueeze method.

For the original ModelNet40 dataset, a renderer module has been introduced. The renderer makes use of trimesh, pyrender and the osmesa backend. It loads the .OFF mesh and rotates it randomly within the renderer.py code. It then creates an normalized depth image similar to the one created by the pre-processing procedure above. The image can directly be fed to the algorithm afterwards.

# Models

## PCRN

The first model designed is based on Pr. Reza Hoseinnezhad’s idea to use residuals from different 3D geometric primitives. In simpler words, for each point in the input depth image, the shortest distance to the chosen shape is computed and stored in a matrix of the same dimensions as the initial image. In theory, three residual images are required to produce a riche representation of the input. In our basic case, we compute three residual images of the input for each basic primitive i.e.: plane, sphere, cylinder. It was then necessary to implement a layer for each type of shapes with trainable parameters.

1. PlaneResLayer:

The PlaneresLayer first initializes useful information such as the batch size, height and width of the images as well as the parameters that will be learned by the algorithm. These parameters represent the coefficients characterizing a plane in the 3D space.

\*\*\*\*\*

The Residuals are then computed using basic operations form the PyTorch library. The result is reshaped into an image-like shape and rotated to keep the orientation of the original image.

1. SphereResLayer:

The SphereResLayer works the same as the PlaneresLayer except that its parameters have different signification. The parameters here represent the center of the sphere while represents the radius of the sphere. It is important that we enforce this point to be within the unit cube containing the depth image. The residuals are computed by calculating the distance of each points to the center of the sphere and by subtracting the radius .

1. CylResLayer:

The CylResLayer works the same as the other except that it uses more parameters to define the cylinder. The first three parameters, represent a point in the unit cube containing the depth image. The three next parameters, represent the slope of the 3D line starting from the initial point and the last parameter, , represents the radius of the cylinder. The residuals are computed by calculating the smallest distance of each point to the 3D line and then subtracting the radius .

Every ResLayer ‘s forward method takes as input a tensor of shape [batch\_size, height\*width,3]. This tensor contains every (x, y, z) point in the image and uses this format to ease the residual calculation. As mentioned before, the output of each ResLayer is a residual image.

Then, three ResLayers of the same kind are called within a ResBlock. The ResBlock concatenate the outputs of the three ResLayers along the channel dimension and passes the result to a small ConvNet which further process the residuals. It has been found that the best activation for this ConvNet block is the tanh function. The output of the ResBlock is a feature map containing 16 channels.

When building the first part of the PCRN algorithm, the ResidualNet, a ResBlock is initialized for each type of shapes (plane,sphere,cylinder). In the forward pass, each of the ResBlock outputs tensor of the following shape [batch\_size, 16, height, width]. These tensors are again concatenated along the channel dimension, resulting in a tensor of shape [batch\_size, 48, height, width].

The result of ResidualNet is then feed to the popular ResNext101-32x4d. The ResNext101-32x4d network must firstly be modified to receive the 48 channel images as input and output a 55-dimensional vector. The procedure to achieve this is located in the PCRN method. The pretrained ResNext101-32x4d model is initially loaded to make use of the pretrained feature extractor before applying the modifications mentioned above. A visual representation of the model is shown in Figure 1.

Lastly, another feature is added to the model to make sure that the desired shape’s parameters stay between zero and one (i.e. the center of the sphere stays inside the unit cube). We define a class object called WeightClipper that contains the following methods:

1. def \_\_init\_\_(self, frequency=1):

This method simply initializes the frequency at which the WeightClipper is applied.

1. def \_\_call\_\_(self, module):

The call method is used to use the class as a function and will firstly check if the module that it is applied to has the attribute related to the parameters we want to clamp between 0 and 1. If true, then the clamping operation is performed on the parameters.

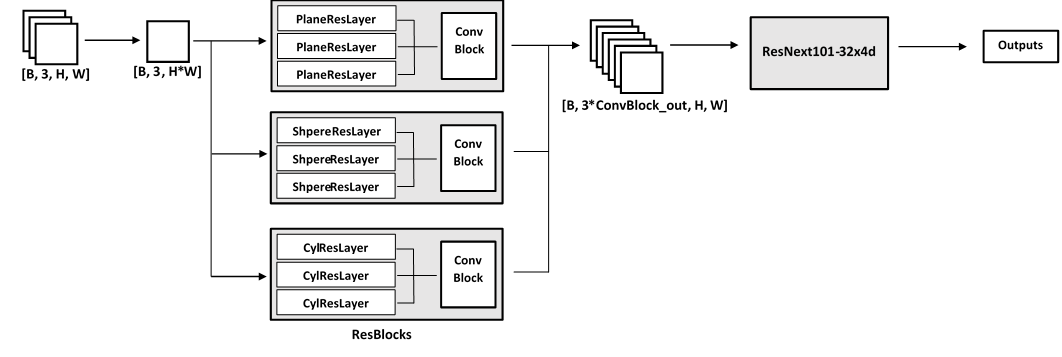


Figure 1:PCRN model

## PrimalNet

The second model that was implemented is a generalization of the PCRN model. The hypothesis is that the residuals computed with respect to a basic shape is equivalent to apply a 1 x 1 covolution on a representation of the input. Since each shape has its own parametric equation, it was thought that projecting the input into other dimensions by using a variant of the kernel trick would mimic the calculation of the residual.

The convolution would then act as the shape descriptor by applying its weights on the projected dimensions. In the case that the shape is a plane, a 1 x 1 convolution on three channels (x grid, y grid, z grid) is equivalent to calculating the residual from a plane like in the method above. In the case we augment the number of channels at the input by creating combinations of the initial channels, we will be creating the equivalent of a representation of the input in a higher dimension, which is comparable to try to calculate the residuals with respect to other shapes. This version of the PCRN, called the PrimalNet will then project the input into a predefines number of powers, creating a polynomial form of the input for each power, and each polynomial formed will be processed by a 1 x 1 convolution operation before being passed to our modified ResNext101-32x4d module. See Figure 2 for a visual representation of the model.

In this specific case, it is desired that every input channel (representing a monomial of a polynomial), is convolved with three different kernels for a certain degree of polynomial d. Then, we will sum over the first feature maps of each channel for that degree of polynomial to obtain the first out of three feature maps for that very specific polynomial. We sum over the seconds and finally the thirds to obtain the two last feature maps for that degree of polynomial. This procedure is obviously repeated over each degree of polynomial, which yields a three channel feature map for each degree.

This convolutional operation is not standard and requires to adapt to our situation one of PyTorch’s convolution features: grouping. Usually, PyTorch’s 2d convolution uses only one group and applies n\_out number of kernels to the input and summing every intermediate feature map for a kernel together to generate one feature map. In our case, we need to have three different kernels going over every input channel and sum only the output of the first kernels of every channel, then the second and the third individually, which is not possible with the initial grouped convolution. See Figure 2 for visual representation of the operation.

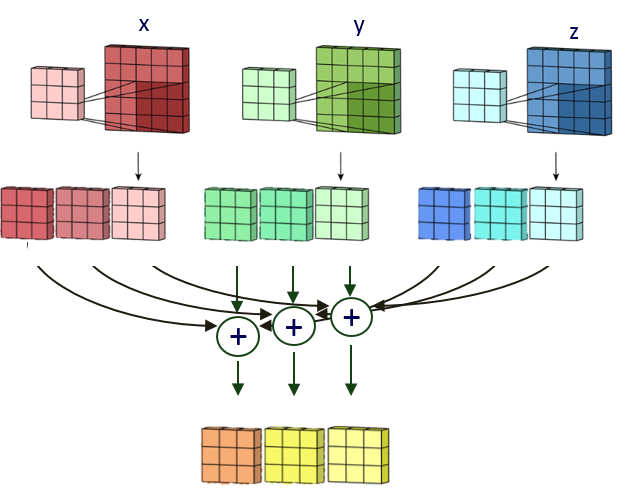


Figure 2:Modified grouped convolution

What was done to overcome this issue is to have as many groups as inputs for a degree of polynomial and have every input channel to generate three feature maps. We then reshape the tensors into the following form : [batch, out\_channels/3, 3, height, width] and sum along the first axis to obtain the desired summation. We repeat this for every degree of polynomial with a modified version of the grouped convolution that can accept variable size of splits. This is necessary since the first degree will have three input channels, the second will have nine, the third nineteen and so on.

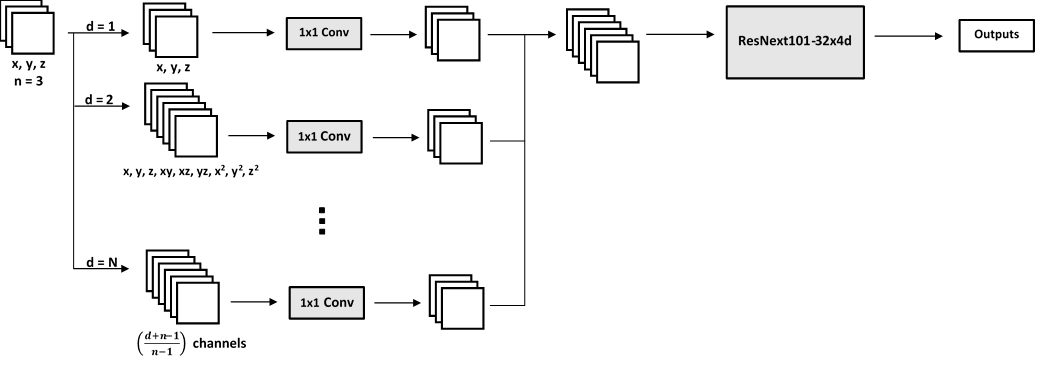


Figure 3:PrimalNet model

The code of PrimalNet model is located into the model2.py file in the GitHub repository. The classes and methods contained within are described in the following:

1. def get\_polynomial(vars, power):

This methods iterates over all possible combinations of the input (X, Y, Z) for each powers and stacks the corresponding matrices together in a list and concatenate them into the channel dimension of a tensor.

1. def get\_Input(vars, degree):

The method calls the get\_polynomials methods for each predefined powers and stores into a list all tensor corresponding to a power.

1. class SplitCNN(nn.Module):

This class inherits from the nn.Module object and contains the implementation for grouped convolutions with different split sizes. It will divide the input according to the split argument and will return a tensor with the different summed outputs concatenated along the channel dimension.

1. class PolyLayer(nn.Module):

This class simply calls the previous class and the get\_Input method to create the base of the final model.

1. def PrimalNet(degree, split, num\_classes= 55):

This final method calls the ResNext101-32x4d pretrained model and modifies it to allow the coupling with PolyLayer module and to output the correct number of classes.

# Main Programs

## Main - PCRN

The main program used to train PCRN is built along a pretty standard format. The import section although has a particular order to make sure PyTorch uses the right GPUs for the training procedure.

The following section initializes some basic hyperparameters, such as the number of epochs, number of classes, batch size, learning rate and validation ratio, all necessary to initialize the model and the optimizer.

Next, the data is loaded through the MyDataset class described in the first section of this document. Before creating the dataset object, we define a transform object containing the sequential transformations to be applied to the model. These transformations are the tensor transformation and our custom transformation. The dataset will be created using the data’s root directory and the transformations. The dataset is then shuffled into a training, validation and test sets and processed by a DataLoader iterator used to feed the data during the training phases.

For the training loop, the program first iterates over the epoch number previously set. The DataLoader iterator is then then called to yield a batch of images and labels. The images are transformed into a list of 3D coordinates using a mesh grid generated by PyTorch. The result is feed to the model and the loss is calculated. Optimization by backpropagation is finally applied to the model and followed by the WeightClipper method. Every 50 batches, the algorithm displays the current loss and the progression of the training procedure. The parameters describing the ResidualNet shapes are also saved in a text file allowing us to visualize the progression of the shapes through the training.

Once an epoch is completed, the algorithm runs on a validation set and a characterization of the accuracy is obtained. These two last steps are repeated for every epochs.

The model is saved in a .ckpt file and a plot of the training and validation learning curves is computed.

The main program also logs every parameter associated with one of each shape to allow a visual representation of their evolution throughout the training. It allowed us to discover the need for a higher learning rate for the shape parameters. It turns out that the shapes were varying only slightly from their initialization. Seeking for an option that would allow the algorithm to explore more possibilities within the optimization space, we implemented and tuned a different learning rate for the ResidualNet parameters.

## Main – PrimalNet

The main program for PrimalNet training is the same as the Main for PCRN. It only differs on two operations performed in the training loop. In the PCRN training loop, the input depth image is transformed into a tensor listing all the 3D coordinates while for PrimalNet we simply stack the X, Y, and Z matrices along the channel dimension in a tensor and feed it to the model. The model then takes care of creating the different projections into higher spaces. The second difference is that no clamping of the parameters is done for this model.

Like the first model, the PrimalNet model also has a hyperparameter that allows us to choose a different learning rate for the first layers which allows a more focalized training.

## Other features

The main codes also have other interesting features to help to visualize and understand what is happening during the training. For the PCRN main code. simply place the three shape.txt files in the appropriate location and make sure their paths are correctly referenced in the Animate\_shapes.py code located in the tools folder.

Generation of a confusion matrix, which allows us to see what are the items that classified with less precision and with exactly what other item it is being confused with. This can help us retrace design flaws or sensitive categories.

The main codes also have a function which allows to train ModelNet40 using a model pretrained on ShapeNet. Just change the pretrained variable to 1 and select the appropriate dataset o finetune on. WARNING: this hypothesis hasn’t been confirmed yet but it seems like ModelNet is actually a subset of the ShapeNet dataset. In that case, it would mean that pretraining on ShapeNet could result In training on some elements of the testset of ModelNet40.

# Loss Functions

## Cross-entropy

The first loss that was used for both models was the cross-entropy loss function. It is a common loss function for classification algorithms with more than two classes. A weighted version of that loss has been tested by according more importance to the classes that the algorithm was most likely to misclassify. However, giving a weight to those classes is a highly subjective manual process that can lead to imprecise balancing of the dataset.

## Focal loss

The focal loss function gives an alternative to manual dataset balancing. It modulates the loss with respect to the probability of the current example. If the algorithm is very confident in its prediction, the loss will be smaller than the classic cross-entropy loss, meaning that the easily classified examples will have less impact on the batch loss. This way, the loss will be more representative of the misclassified examples and will allow the algorithm to focus more efficiently on those examples.