# Lipschitz regularization in CNNs : Design Document

Anirudh Singh

*Code is simply a body to the soul that is the logic and design philosophy. Reading code without understanding the soul is like trying to read a a foreign language for the first time with just a dictionary.*

## 1 High level design

### 1.1 Objectives

The objectives of this project are twofold.

- To implement the functionality of space separable convolutions, space depth separable convolutions and convolutions along depth as fully functioning neural network layers which can be trained from end to end.
- To implement $L_\infty - L_\infty$ lipschitz normalization for the above layers during training. The bound use to normalize the kernel is mathematically formulated as follows: For a 4 dimensional kernel $K_{IO}(x,y)$ where the first subscript denotes input layer index, second subscript denotes output layer index and the two arguments are the spatial indices, we have

$$L = \max_{O} \sum_{I,x,y} K_{IO}(x,y) \tag{1}$$

The implementation has been achieved using tensorflow 2.x. The design will be outlined in this document. As far as possible, the implementations are modular and extend tensorflow's own layer functionality. This allows us to not only separate the code involving logically disparate sections of functionality, but also allows us to replace existing architectures with our custom layers and regularizations extremely easily.

### 1.2 Design

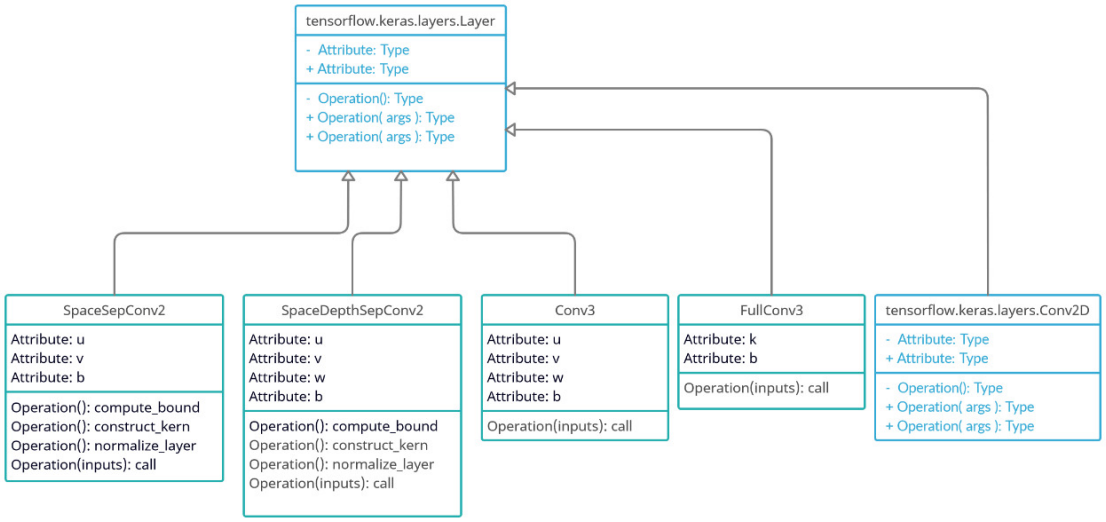

Figure 1: Subclassing the layer interface to implement custom convolutions

The functionality for space separable convolutions, space depth separable convolutions and convolutions along depth (hereby referred to as constrained convolutions) is implemented using subclassing of the standard layer interface provided by tensorflow. This is the same interface which acts as a parent class to existing layers like Dense and Conv2D. Since all the required methods prescribed by the interface are implemented, we may use our custom classes in places of these standard existing classes by means as simple as changing the layer name when specifying the architecture.

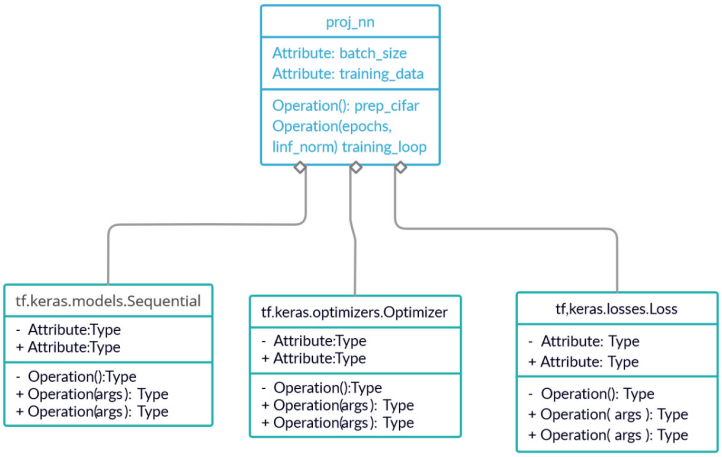


Figure 2: Top level class nnetSystem encapsulating the training phase

For the functionality involving regularization, our custom layer classes implement a normalization function which is responsible for normalizing its layer. We also create a new class called nnetSystem which encapsulates a tensorflow sequential model, a tensorflow loss function, a tensorflow optimizer training data and batch size. This class implements a training loop method which manually updates the trainable weights of the tensorflow model using the gradients of each batch. For our custom layers, we also call the normalize function implemented in the layer to regularize the layer weights after a certain number of batch gradient updates. The advantage of this encapsulation is that it ensures modularity, so that any future implementations of different functionalities can use the same training loop function to perfom regularization, with no change to code.

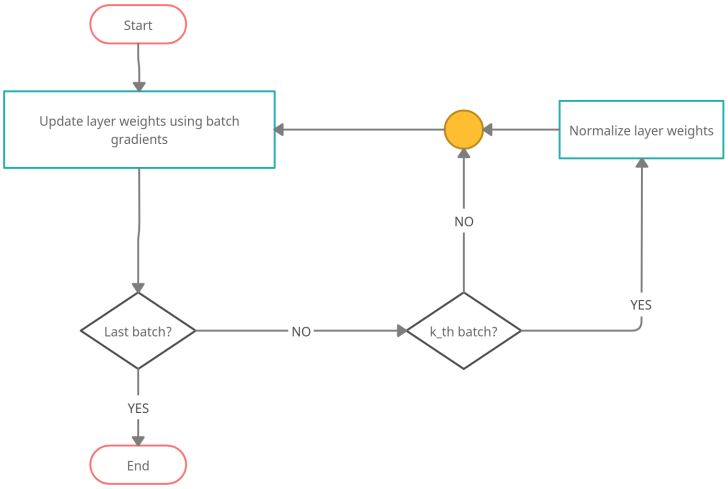


Figure 3: Simple code flow assuming a single layer network and a single training epoch

# 2 Low Level Design

The design is object oriented for the most part, so all of the data and associated methods are encapsulated within classes. Following are the classes with the associated data, methods and signatures.

## 2.1 Classes

- nnetSystem: Encapsulator for our training loop.
  - Attributes:
    1. model (type : tensorflow.keras.models.Sequential): Model object which encapsulates the architecture used during forward and backward pass during training.
    2. batch_size (type : integer): Size of batch used for a single gradient update.
    3. train_data (type: tensorflow.data.Dataset): Training data broken up into iterable batches on the basis of batch_size.
    4. loss_function (type: tensorflow.keras.losses.Loss): Loss function used during training.
    5. opt (type : tensorflow.keras.optimizers.Optimizer): Optimizer used for performing gradient descent.
  - Methods:
    1. nnetSystem(model, data='cifar10', batch_size = 256, opt = tf.keras.optimizers.Adam(), loss_fn =tf.keras.losses.SparseCategoricalCrossEntropy() ): Constructor. model argument is mandatory and specifies a tensorflow sequential model (tf.models.Sequential). 'cifar10' is the only currently supported option for the data argument and loads the CIFAR10 dataset. opt is of type tf.keras.optimizers.Optimizer while loss_fn is of type tf.keras.losses.Loss.
    2. prep_cifar(): Prepares the CIFAR10 dataset and populates train_dataset attribute with it.
    3. training_loop(epochs, linf_norm, k): Runs the training loop with the given number of epochs. If lin_norm is True, perform regularization every k steps.
- SpaceDepthSepConv2: Class implementing convolutions separable along space and depth.
  - Attributes:
    1. in_channels (type: integer): Number of input channels.
    2. out_channels (type: integer): Number of output channels.
    3. kern_size (type: integer): Size of the filter kernel in space.
    4. u(type: float, shape: [out_channels, kernel_size]): Vectors denoting kernel component over width, one for each output channel; learnable.
    5. v(type: float, shape: [out_channels, kernel_size]): Vectors denoting kernel component over height, one for each output channel; learnable.
    6. w(type: float, shape: [out_channels, in_channels]): Vectors denoting kernel component over depth, one for each output channel; learnable.
    7. b(type: float): Vector denoting bias component, same shape as output; learnable.
  - Methods:
    1. SpaceDepthSepConv2(input_dim, out_channels, name, kern_size=3): Constructor. The input_dim argument is mandatory and specifies the input dimensions to this layer as a tuple of integers. out_channels is the number of output channels. kern_size if the field of convolution in space.
    2. construct_kern(): Construct the kernel which is used for convolution. The kernel is given by k[i,j,k,l] = u[l,j]*v[l,i]*w[l,k].
    3. call(): Construct the kernel and perform the forward pass through the layer via tensorflow.nn.conv2d and tensorflow.keras.activations.relu.
    4. compute_bound(): Compute the kernal and find its bound as prescribed in the high level design.
    5. normalize_layer(): Normalize the layer by dividing each u, v and w by the cube root of the bound computed using the above function, say L. Effectively, the kernel computed by the layer is divided by L.

- SpaceSepConv2: Class implementing convolutions separable along space only.
  - Attributes:
    1. in_channels (type: integer): Number of input channels.
    2. out_channels (type: integer): Number of output channels.
    3. kern_size (type: integer): Size of the filter kernel in space.
    4. u(type: float, shape: [in_channels, out_channels, kernel_size]): Vectors denoting kernel component over width, one for each ()input_channel, output_channel) tuple; learnable.
    5. v(type: float, shape: [in_channels, out_channels, kernel_size]): Vectors denoting kernel component over height, one for each ()input_channel, output_channel) tuple; learnable.
    6. b(type: float): Vector denoting bias component, same shape as output; learnable.
  - Methods:
    1. SpaceSepConv2(input_dim, out_channels, name, kern_size=3): Constructor. The input_dim argument is mandatory and specifies the input dimensions to this layer as a tuple of integers. out_channels is the number of output channels. kern_size if the field of convolution in space.
    2. construct_kern(): Construct the kernel which is used for convolution. The kernel is given by k[i,j,k,l] = u[k,l,j]*v[k,l,i].
    3. call(): Construct the kernel and perform the forward pass through the layer via tensorflow.nn.conv2d and tensorflow.keras.activations.relu.
    4. compute_bound(): Compute the kernal and find its bound as prescribed in the high level design.
    5. normalize_layer(): Normalize the layer by dividing each u, v by the square root of the bound computed using the above function, say L. Effectively, the kernel computed by the layer is divided by L.
- Conv3: Class implementing convolution along height, width and depth. Kernel is still evaluated seperably.
  - Attributes:
    1. in_channels (type: integer): Number of input channels.
    2. out_channels (type: integer): Number of output channels.
    3. out_mult (type: integer): out_channels/in_channels
    4. kern_size (type: integer): Size of the filter kernel in space.
    5. kern_depth (type: integer): Depth of the filter kernel.
    6. u(type: float, shape: [ out_mult kernel_size]): Vectors denoting kernel component over width, one for each output multiplier; learnable.
    7. v(type: float, shape: [out_mult, kernel_size]): Vectors denoting kernel component over height, one for each output multiplier; learnable.
    8. w(type: float, shape: [out_mult, kernel_size]): Vectors denoting kernel component over depth, one for each output multiplier; learnable.
    9. b(type: float): Vector denoting bias component, same shape as output; learnable.
  - Methods:
    1. Conv3(input_dim, name, out_mult=1, kern_size=5, kern_depth=5, ): Constructor. The input_dim argument is mandatory and specifies the input dimensions to this layer as a tuple of integers. out_mult is the multiplier to input depth used to calculate the output depth. kern_depth is the depth of convolutions. kern_size if the field of convolution in space.
    2. call(): Construct the kernel and perform the forward pass through the layer via tensorflow.nn.conv3d and tensorflow.keras.activations.relu.
- FullConv3: Class implementing convolution along height, width and depth.
  - Attributes:
    1. in_channels (type: integer): Number of input channels.
    2. out_channels (type: integer): Number of output channels.
    3. out_mult (type: integer): out_channels/in_channels
    4. kern_size (type: integer): Size of the filter kernel in space.

5. kern_depth (type: integer): Depth of the filter kernel.
6. k(type: float, shape: [ out_mult, kern_depth, kern_size, kern_size]):3 dimensional kernel used to convolve with the 3D input, one for each output multiplier; learnable.
7. b(type: float): Vector denoting bias component, same shape as output; learnable.

- Methods:
  1. FullConv3(input_dim, name, out_mult=1, kern_size=5, kern_depth=5, ): Constructor. The input_dim argument is mandatory and specifies the input dimensions to this layer as a tuple of integers. out_mult is the multiplier to input depth used to calculate the output depth. kern_depth is the depth of convolutions. kern_size if the field of convolution in space.
  2. call(): Construct the kernel and perform the forward pass through the layer via tensorflow.nn.conv3d and tensorflow.keras.activations.relu.