0.299*Project 3: ADAS Traffic Sign Detection

Team Members

Sanskar Agrawal - 16EE10041
Aryan Jaiswal - 16EE10059
Manu Maheshwari - 16EE10062
Harsh Maheshwari - 16EE30010
Siddhant Haldar - 16EE30025
Rohan Yadav - 16EE10039
Umang Fogla - 16EE10051

Introduction

In recent years, deep learning has become a hot topic of discussion in the technical community. Deep learning is a machine learning technique that teaches computers to do what comes naturally to humans: learn by example. Its ability to learn to extract important feature is one of the biggest reason for its success. It has found many applications in the real world and here we are trying to prove its mettle by one of its application i.e. classification of the image. We are training a neural network to classify the given image in the traffic sign category it belongs.

Dataset

We are using GTSRB dataset for training the model. It have 43 different classes with a total of 39,209 images. Dataset has multiple copy of the same images with different sizes so we are using the images whose dimension is close to 32x32x3 as the input to the network have a dimension of that size. Thus we have a total of 1,306 number of images for training.

Preprocessing

Class Definition:

```
class preprocessing
{
public:

    float *gray_img, *hist_img, *trans_img, *norm_img;
    int h,w,channels;

    preprocessing(int h, int w, int channels);

    void BGR2GRAY(float* img);
    void Histogram_Equalization(float *img);
    void GrayLevel_Neg_Transformation(float *img);
    void GrayLevel_Log_Transformation(float *img);
    void GrayLevel_Gam_Transformation(float *img);
    void Normalization(float *img);
};
```

The aim of the class is to perform preprocessing on the input image and convert it in a suitable form for training the classifier. This not only helps to enhance the dataset, but also helps to improve the accuracy of the classifier. A number of preprocessing algorithms has been used to enhance the dataset. They are :

- **BGR2GRAY**

  This function takes in an input image of specified height(h), width(w) and number of channels(channels) and then converts it into grayscale image using cuda kernel. The kernel converts each pixel of the input image into corresponding grayscale value by assigning each thread to a pixel with the block dimension being **(h,w).**

  The kernel is defined as shown below -:

```
__global__
void bgr2gray(float *in_img, float *gray_img, int h, int w, int channel)
{
```

- **Histogram_Equalization**

  This function is used to perform histogram equalization on the input image and produce contrast enhanced images as an output. This is achieved with the help of two cuda kernel calls, one of which calculates the histogram containing the frequency of each intensity value(0 - 256) and the other to calculate CDF of the histogram and then map the normalized sum back to the output image.

  The kernel is defined as shown below -:

```
__device__ int hist[256];  //device variable to share among 2 kernels
__global__ void calcCDF(float* in_img , float* out_img, int h, int w, int num_levels)
__global__ void histogram_equalization(float *in_img, int h, int w, int num_levels)
```

  In order to avoid race conditions ,atomic operations was used to calculate the histogram in multi-threaded environment using shared memory. The following function was used :

float atomicAdd(float* address, float val); . Despite the overhead, it helped to compute the histogram more efficiently.

- **GrayLevel Transformation**
  This function is used for further enhancement of the image. Enhancing an image provides better contrast and a more detailed image as compare to non enhanced image.

  The kernel is defined as shown below -**:**

```
__global__ void negative_transformation(float *in_img, float* out_img, int h, int w, int num_levels)
__global__ void log_transformation(float *in_img, float* out_img, int h, int w, int num_levels, float param)
__global__ void gamma_transformation(float *in_img, float* out_img, int h, int w, int num_levels, float gamma)
```

  - **GrayLevel_Neg_Transformation**
    In negative transformation, each value of the input image is subtracted from the maximum intensity value of each pixel (L-1) and mapped onto the output image. This is achieved using cuda kernel call to convert the entire image into it's negative by assigning each thread a single pixel.
  - **GrayLevel_Log_Transformation**
    During log transformation, the dark pixels in an image are expanded as compared to the higher pixel values. The higher pixel values are kind of compressed in log transformation.This is also achieved using cuda kernel call which assigns each thread to a pixel and performs corresponding mathematical operation.
  - **GrayLevel_Gam_Transformation**
    Images which are not properly corrected can look either bleached out, or too dark. Gamma correction controls the overall brightness of an image using the following formula :

$$I^{'} = 255 \times \left(\frac{I}{255}\right)^{\gamma}$$

    Variation in the value of gamma varies the enhancement of the images. The value of gamma is obtained from the macro definition. Here also a single kernel call assigns each thread to a pixel value which performs mathematical operations and maps them back to output image.

- **Normalization**
  Finally, the enhanced image is normalized with zero mean and unit variance to make the input image suitable to be fed to the network. This is achieved by calling the kernel and performing the sum operations using *atomicAdd()* and finally mapping the calculated value to each pixel in the image thread-wise.

_____The kernel is defined as shown below -:

```
__global__ void normalize_img(float* in_img, float* out_img, int h, int w, int num_levels)
```

Network Architecture

Below is the architecture of the model which was provided in the paper -

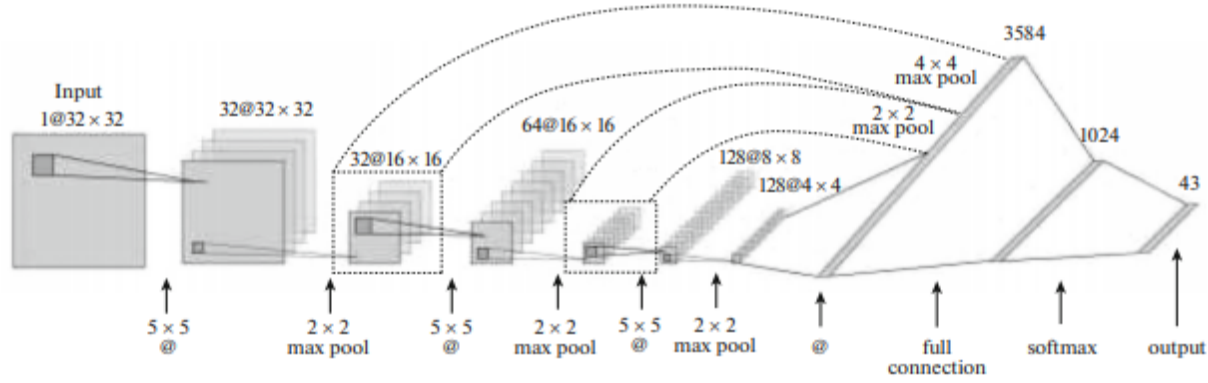


Fig. 15. Convolutional network model.

Below is the code snippet of the network architecture, we have used.

```
//Defining Model
Conv2d C1(1, 32, 5);
ReLU R1(IMAGE_DIM, IMAGE_DIM, 32);
MaxPool M1(IMAGE_DIM, IMAGE_DIM, 32);

Conv2d C2(32, 64, 5);
ReLU R2(IMAGE_DIM/2, IMAGE_DIM/2, 64);
MaxPool M2(IMAGE_DIM/2, IMAGE_DIM/2, 64);
Dropout D1(0.2, IMAGE_DIM/4, IMAGE_DIM/4, 64);

Conv2d C3(64, 128, 5);
ReLU R3(IMAGE_DIM/4, IMAGE_DIM/4, 128);
MaxPool M3(IMAGE_DIM/4, IMAGE_DIM/4, 128);
Dropout D2(0.2, IMAGE_DIM/8, IMAGE_DIM/8, 128);

FC F1((IMAGE_DIM*IMAGE_DIM/64)*128, 1024);
ReLU R4(1, 1, 1024);
Dropout D3(0.2, 1, 1, 1024);

FC F2(1024, n_classes);
Sigmoid S1(1,1,n_classes);
```

Each sub-module used in the architecture has been explained in further sections.

Convolution Layer

The primary operation while implementing a Convolutional Neural network is convolution which helps us to reduce the required number of parameters and to take account of contextual information which is important for images. Mathematically, we convolve a kernel of fixed height, width, depth to the feature map. The convolution operation has been shown in the figure below.

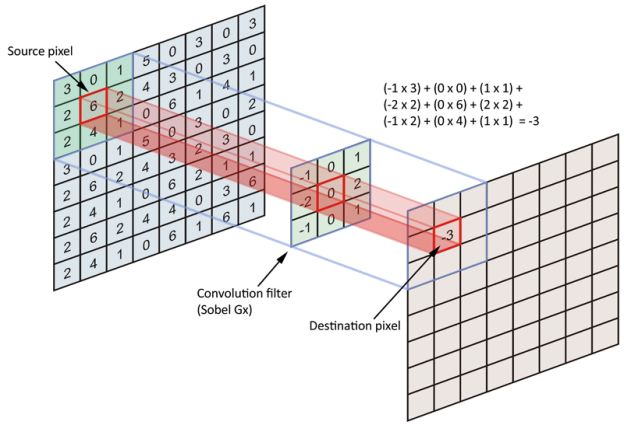


Figure : Convolution Operation

Our implementation

We have defined a class Conv2d as shown below in the snippet -

Weight is used to denote the weights of the kernel which will be convolved with the feature map.

```
class Conv2d
{
public:
    bool is_first; //For Momentum initialization
    float *weight; //channels_out, channels_in, kernel_size, kernel_size
    float bias;
    float* del_weight;
    float* del_vw; //For momentum weight
    int channel_in, channel_out, kernel_size;

    Conv2d(int channel_in, int channel_out, int kernel_size);
    float* forward(float* image, int img_width, int img_height);
    float* backward(float* del_out, float* input, int input_height, int input_width);
    void step(float l_rate, float beeta=0.9);
};
```

Implementation of forward() function :

In this, we consider the input feature as a 1D array in order to pass it to Cuda Kernel. So the arrangement of each feature map is such that first comes the width and then the height and in the same way data of every channel comes. The dimension of grid is (1,1,channel_out) and the block is (image_height, image_width, 1). The kernel function is implemented such that each thread computes convolution of the kernel with the image.

Implementation of backward() function :

We have two kernel function to compute derivative of output with respect to the image and filter(or kernel). The Kernel function used to compute the derivative with respect to the filter have grid dimension of (kernel_size, kernel_size, channel_out) and block dimension of (1, 1, channel_in). We have used the kernel_size in the grid dimension so as to incorporate the fact that a maximum number of threads allowed in a block is 1024. In the kernel function, each thread is used to compute the derivative with respect to each value of filter. The kernel function used to compute derivative with respect to the input have the same grid dimension and block dimension is (input_height, input_width, 1). In the kernel function, each thread is used to compute the derivative with respect to each value of the input feature.

Implementation of step() function :

After computing the derivative we are using the Momentum optimizer to update the weight with the default value of beta being 0.9. This includes the running average of the values of the derivative which we have computed over the previous iteration. The formula of momentum is

$$v(t) = v(t-1) * \beta + (1-\beta) * dw$$

$$W = W - \alpha * v(t)$$

Fully Connected Layers

Fully connected layers connect every neuron in one layer to every neuron in another layer. It follows the same principle as the traditional multi-layer perceptron neural network. The flattened matrix obtained from the convolutional layers are passed on to a series of fully connected layers in order to classify the images.

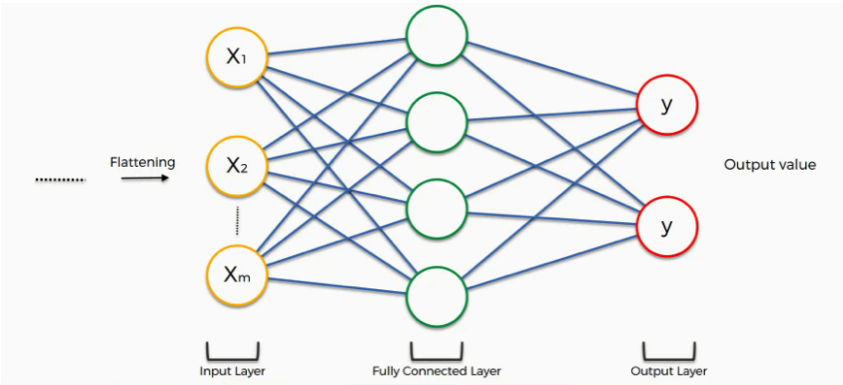


Figure : Fully Connected Layer

Our Implementation
A snippet of the class defined for fully connected layers is as follows :

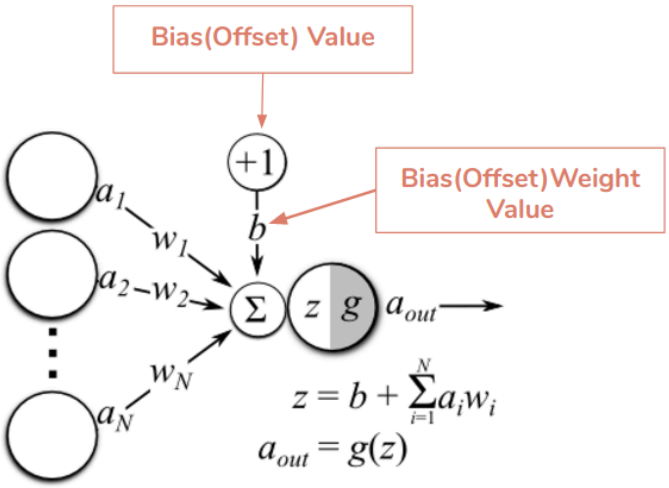
```
class FC
{
public:

    float *weight;
    float *bias;
    float *dw, *dw_old;
    float *db, *db_old;
    float *d_in, *out;
    int in_size, out_size,first;

    FC(int in_features, int out_features);
    float* forward(float *in);
    float* backward(float *in,float *d_out);
    void step(float lr, float beta);
};
```

An object of this class stores the weight and bias mapping the input data from the preceding class to the output of the layer. The output produced by the layer and the gradients flowing backward from the layer are stored in the variables *"out"* and *"d_in"* respectively. The weight and bias updates have also been stored in the object of this class.

Implementation of forward() function
The equation for forward propagation of fully connected layers is -



During forward propagation, each thread on the device computes the weighted sum of the neurons of the previous layer corresponding to a specific output node. In the implemented network, the maximum dimension of the output fully connected layer is 1024. Hence, we have restricted the operation to be implemented on a single block in order to allow warped implementation of the Cuda kernel, thus making it more efficient.

Implementation of backward() function:

During backward propagation, the function takes the input layer and the incoming gradients as input and outputs the outgoing gradients.

Backpropagation in the fully connected layer:

$$d_{in} = W^T d_{out}$$
$$d_w = d_{out} x^T$$
$$d_b = d_{out}$$

We can see that we need to implement matrix transpose and matrix multiplication for implementing the backward function. We have implemented matrix transpose as follows:

```
if(i<m && j<n)
{
    int index_in  = i*n+j;
    int index_out = j*m+i;
    w_transpose[index_out] = w[index_in];
}
__syncthreads();
```

We have used also used an optimized matrix multiplication. For maximum utilization of threads in each block, we have used a thread allocation scheme as shown in the bottom figure.

```
dim3 grid(in_size,2,1);
dim3 block(1,max(out_size, in_size)/2,1);
```

Implementation of step() function:

Step function for the fully connected layer has been implemented in the same fashion as in the convolution layer.

Activation Function

Sigmoid

Sigmoid function is a S-shaped function which transforms the input data to a range of [0,1]. In deep learning, we have to predict a probability of the output and hence, sigmoid is often the right choice when it comes to applying activation functions on the last layer of the network. The shape of the function is shown in the following figure.

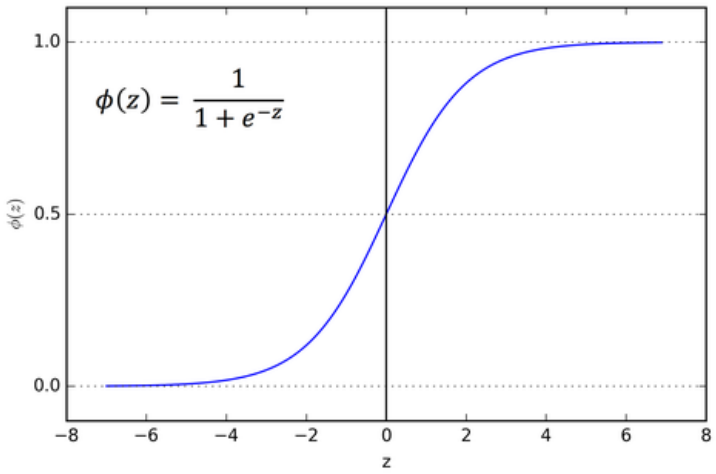


Figure : Sigmoid Function

Our implementation :
A snippet of the class defined for the sigmoid function is as follows :

```
class Sigmoid
{
public:
    float *out,*d_in;
    Sigmoid(int h, int w, int channel);
    void forward(float *in, int h, int w, int channel);
    void backward(float* d_out, int h, int w, int channel);
};
```

An object of this class stores the output obtained after incoming data is subject to the sigmoid function. The variable *"out"* has been used for this purpose. Also, the gradients to be passed on to the previous layers during back propagation are stored in the variable *d_in".*

Implementation of forward() function
During forward propagation, the *forward()* function takes the incoming feature maps and passes them to the device. Each thread is assigned a particular neuron on which it applies the sigmoid function. The results corresponding to each thread are stored in an output variable which are then transferred from the device to the host. These values have been stored in the *"out"* variable since they will also be required for calculating the gradients during backpropagation.

Implementation of backward() function
The derivative of the sigmoid function is given by -

$$\frac{d\sigma}{dx} = \sigma(x)(1 - \sigma(x))$$

Hence, in order to compute the gradient flowing backwards from the sigmoid layer, the values stored during forward propagation are utilized. As done in forward propagation, the incoming gradients from the succeeding layers are assigned to individual threads on the device and the gradients at each neuron are computed and stored in the variable *"d_in"*.

ReLU

The Rectified Linear Unit is the most commonly used activation function in deep learning models. The function returns 0 if it receives any negative input, but for any positive value it returns that value back. So it can be written as -

$$f(x) = max(0, x)$$

The curve for ReLU is as shown -

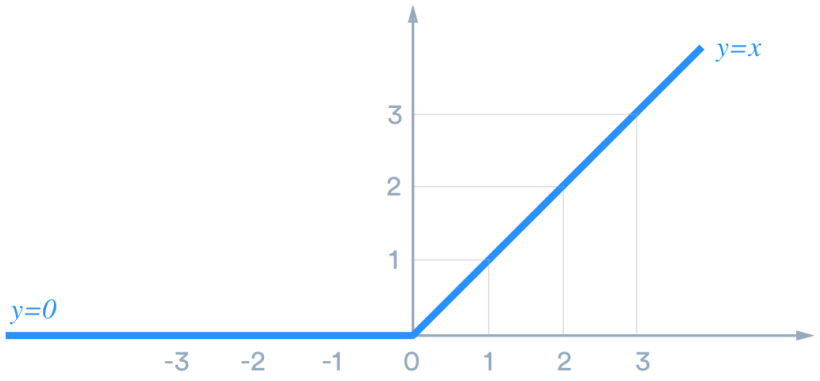


Figure : ReLU function

Our Implementation

A snippet of the class defined for the ReLU function is as follows :

```
class ReLU
{
public:
    float *out,*d_in;
    ReLU(int h, int w, int channel);
    float* forward(float *in, int h, int w, int channel);
    float* backward(float* d_out, int h, int w, int channel);
};
```

An object of this class stores the output obtained after incoming data is subject to the ReLU function. The variable *"out"* has been used for this purpose. Also, the gradients to be passed on to the previous layers during back propagation are stored in the variable *d_in".*

Implementation of forward() function
During forward propagation, the *forward()* function takes the incoming feature maps and passes them to the device. Each thread is assigned a particular neuron on which it applies the ReLU function. The results corresponding to each thread are stored in an output variable which are then transferred from the device to the host. These values have been stored in the *"out"* variable since they will also be required for calculating the gradients during back propagation.

Implementation of backward() function
ReLU function zeroes out the negative inputs, thus, preventing them from affecting the output produced by the network. During backward propagation, the gradients coming from the succeeding layers are suppressed at the locations which had been zeroed out during forward propagation. Hence, the weight variables related to these locations are not updated during the backpropagation step. Similar to what had been done for the sigmoid activation function, the operations corresponding to individual neurons have been assigned to independent threads on the device in order to parallelize the operation.

Dropout

Dropout refers to ignoring neurons during training phase of certain phase of neurons which is chosen at random. These selected neurons are not considered during a particular forward or backward pass.This is usually done to prevent overfitting. A fully connected layer occupies most of the parameters, and hence, neurons develop co-dependency amongst each other during training which curbs the individual power of each neuron leading to over-fitting of training data.

Our Implementation

Class Definition

```
class Dropout
{
public:
    bool* mask;
    float drop_prob, *d_in;
    int h, w, channel;
    Dropout(float drop_prob, int h, int w, int channel);
    float* forward(float*in);
    float* backward(float* d_out);
};
```

Forward Propagation

First, we initialize a boolean map of the dimension same as that of input. We have done this parallelly by allocating the CUDA device states with a random number. We pass this map to the dropout kernel. We launch this kernel with grid dimension as (1, 1, no_of_channels) and block dimension as (width, height, 1). This way, we have different channels on different blocks. In our model, image dimensions are 32x32, so a number of threads in a block will never cross 1024.

Backward Propagation

We are saving the index of dropped elements in the corresponding map. This kernel is launched with the same kernel configurations as that of forward pass. The input derivative at the location saved by the boolean map is changed to zero. This way, the gradients at that locations will not be back propagated further.

Max Pooling

A pooling layer is an essential building block of a CNN. Pooling. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network. Pooling layer operates on each feature map independently. The most common approach used in pooling is max pooling.It has no trainable parameters.

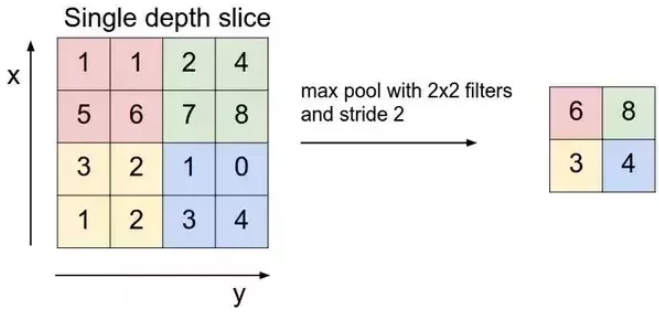


Figure : Max Pooling Operation

Our implementation

We used the following class definition in our implementation:

```
class MaxPool
{
public:
    int *mask; //to remember the location
    float *out, *d_in;
    MaxPool(int h, int w,int channel);
    float* forward(float *in, int h, int w, int channel);
    float* backward(float *d_out, int h, int w,int channel);
};
```

Mask - It is used to store the positions of the maximum elements which are forwarded out of a pooling window.It will be useful while backpropagation.

Implementation of forward() function

During forward propagation, the function takes the input feature map, size of the input feature map and the number of channels as input.It returns the output feature map of half the dimensions and same number of channels.

In the kernel function of MaxPool forward, computation of each output element is parallelized. For this, the image dimensions of the output feature map are given as the number of threads in each block along x and y-direction. The number of channels is given along z-direction.

```
dim3 grid(1,1,channel);
dim3 block(w/2,h/2,1);
maxpool_fp<<<grid, block>>>(g_in,g_out,g_mask,h,w,channel);
```

Here gridDim.z = number of channels so that the number of threads in a block do not exceed 1024 which is constrained by the device architecture.

Here we can access each element of the output feature map by directly accessing the channel number using blockIdx.z and the pixel number of each channel using threadIdx.x and threadIdx.y.

Using the indexing of each output element we can get the indexing of input feature map using appropriate constant multiplications. We are also storing the mask value for each pool window of 2X2 which will be helpful during backpropagation.

Implementation of backward() function

During backward propagation, the function takes the incoming gradients, the dimensions of the output feature map and number of channels as input. It returns the outgoing gradients with the same number of channels.

In the kernel function of MaxPool backward, computation of each input gradient is parallelized. For this, the image dimensions of the output feature map are given as the number of threads in each block along x and y-direction. The number of channels are given along z-direction.

```
dim3 grid(1,1,channel);
dim3 block(w,h,1);
maxpool_bp<<<grid,block>>>(g_d_in,g_d_out,g_mask,h,w,channel);
```

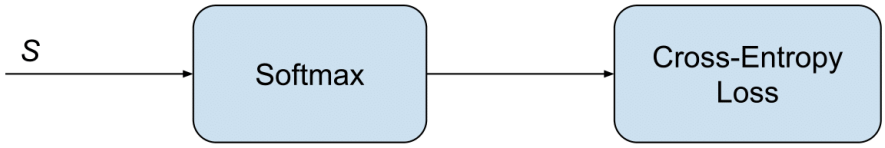
Here gridDim.z = number of channels so that the number of threads in a block do not exceed 1024 which is constrained by the device architecture.

Here we can access each element of the output feature map by directly accessing the channel number using blockIdx.z and the pixel number of each channel using threadIdx.x and threadIdx.y.

Using the indexing of each output element we can get the indexing of input feature map using appropriate constant multiplications and the mask stored during forward propagation.

Softmax cross entropy with logits:

This class contains functions which measures the probability error in discrete classification tasks in which the classes are mutually exclusive. Softmax function calculates the probability of every possible class.



$$f(s)_i = \frac{e^{s_i}}{\sum_j^C e^{s_j}} \quad CE = -\sum_i^C t_i log(f(s)_i)$$

Our Implementation

We have implemented Softmax with numeric stability. In practice, the exponential terms may be very large, and dividing large numbers can be numerically unstable. What we have done is, we are subtracting the maximum value of the array from every element of the array. This makes all the elements in logits to be less than or equal to zero. Thus all the exponential terms are less than one.

Class Definition:

```
class softmax_cross_entropy_with_logits
{
public:
    float loss;
    float* forward(float* logits, int label, int n_classes);
    float* backward(float* out, int label, int n_classes);
};
```

Implementation of forward() function

First, we are finding the softmax probabilities by dividing exponential of an element by the sum of the exponential of all the elements. Then we are applying the cross-entropy loss on the output. We are adding a small offset(1e-7) to the log term to avoid exploding the term.

Implementation of backward() function

The derivative of the cross-entropy loss function with respect to the inputs of softmax function comes out to be (y - ŷ), where ‘y’ is the output of model and ŷ is the ground truth.

Skip Connection

For implementing skip connection into the network we need a **concat** function which concatenates three vectors into one single vector. Since we are only concatenating in one dimension we are using global thread id for indexing.Since upon concatenation the output dimensions are exceeding 1024, we are keeping the number of threads in a block to be 1024 and giving the remaining elements in the grid as **ceil(output dimension/1024)** along any one of the grid dimensions.

Reported Accuracy

Currently, the model is being trained. Hence, we will report the accuracy during the final presentation.