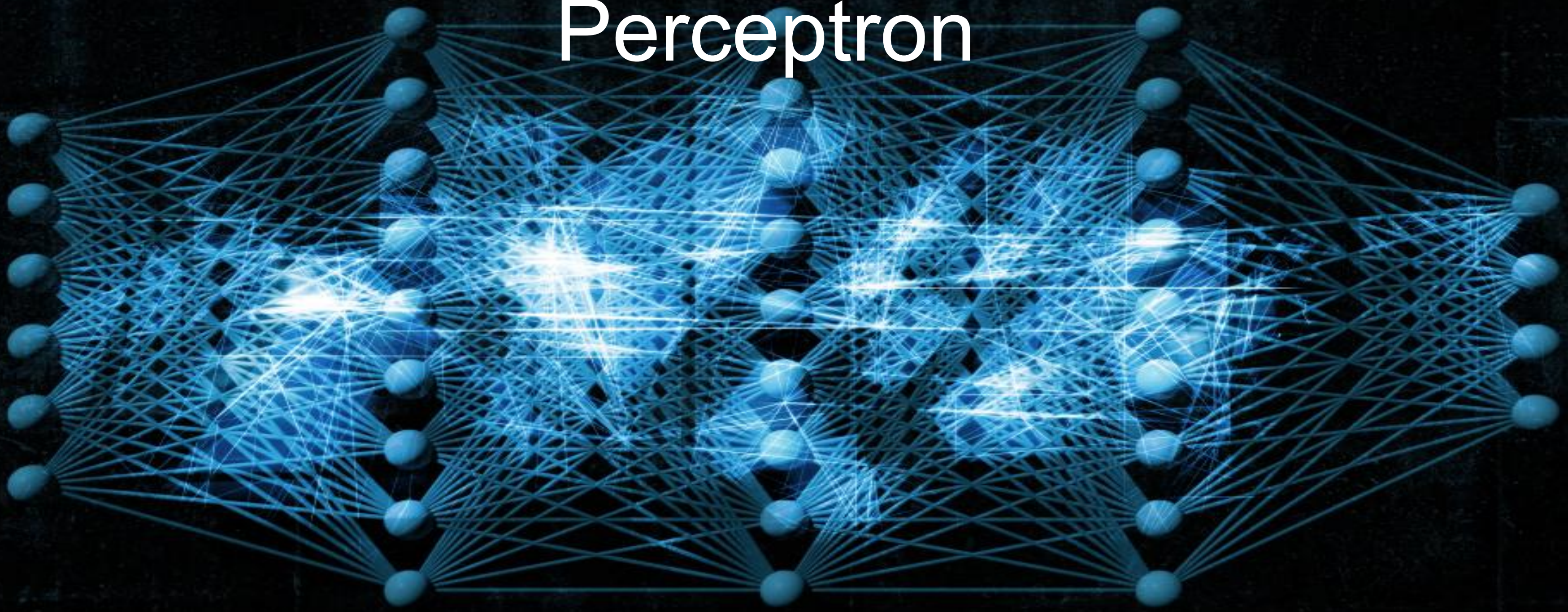
CUDA Accelerated Training of Multilayer Perceptron



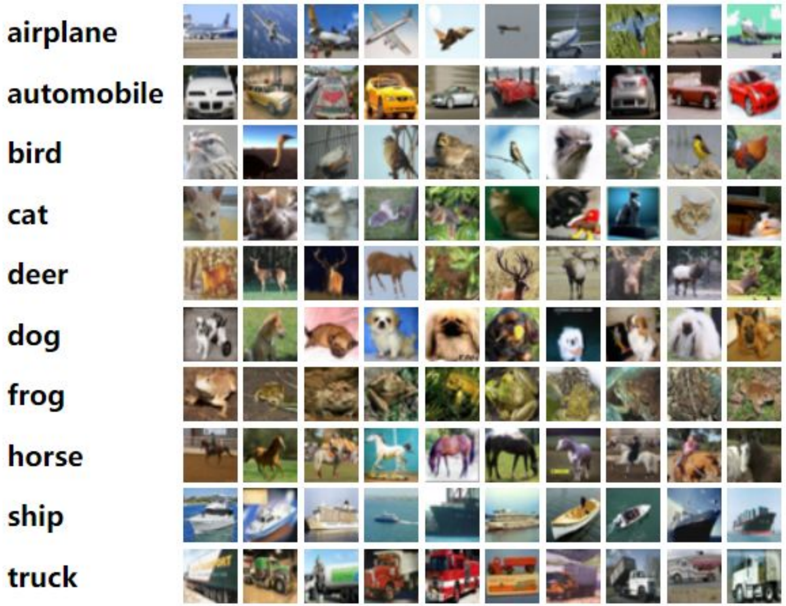
Koffivi F. Gbagbe, Oluwafemi Adejumobi / HPC 2023 Final Project

Skoltech

Dataset Description

The CIFAR-10 dataset is a collection of images that are commonly used to train machine learning and computer vision algorithms. The dataset consists of:

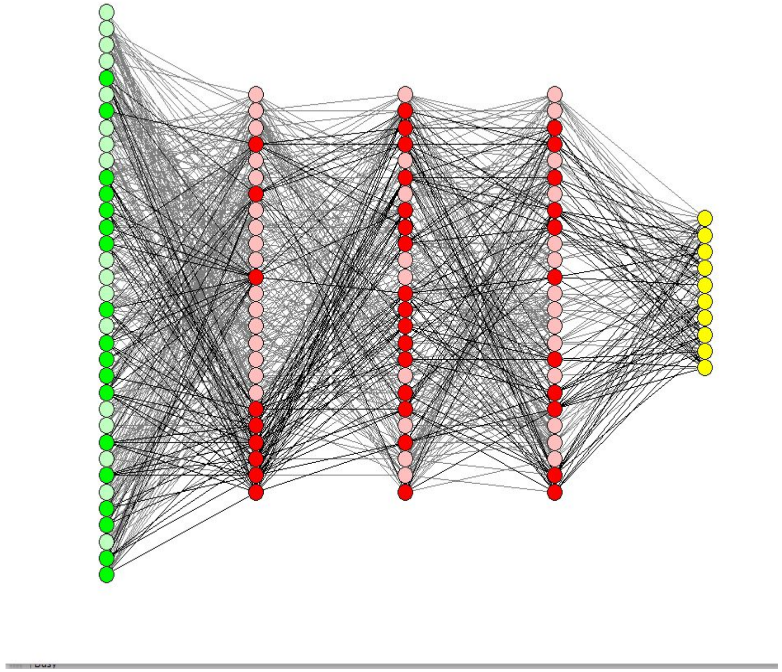
- ❑ 60000 32x32 colour images in 10 classes,
- ❑ with 6000 images per class.
- ❑ There are 50000 training images and 10000 test images



Ref: https://www.cs.toronto.edu/~kriz/cifar.html

The Multilayer Perceptron Architecture

- Input layer : 3072 (32 * 32 * 3) nodes
- First Hidden Layer: 2048 nodes
- Second Hidden layer: 1024 nodes
- Third Hidden layer: 512 nodes
- Output layer: 10 nodes (numbers of classes)



Matrices Operations Kernels:

We implement kernels for matrices operations in the neural network layers

- Dot_Product_Kernel
- Multiplication_Kernel
- Addition_Kernel
- Subtraction_Kernel
- ReLU_Activation_Kernel
- Derivative_ReLU_Kernel
- Tanh_Activation_Kernel
- Exponentiation_Kernel
- Power_Kernel
- Sum_Kernel
- Division_Kernel

```
__global__ void Dot_Product_Kernel(Matrix *A, Matrix *B, Matrix *C, bool trans1, bool trans2) {
    double Cvalue = 0.0;
    int row = threadIdx.y + blockIdx.y * blockDim.y;
    int col = threadIdx.x + blockIdx.x * blockDim.x;
    if (trans1 == false && trans2 == false)
        if (row < A->height && col < B->width) {
            for (int i = 0; i < A->width; ++i)
                Cvalue += Get_Mat_element(A, row, i) * Get_Mat_element(B, i, col);
            Set_Mat_element(C, row, col, Cvalue);
        }
    if (trans1 == true)
        if (row < A->width && col < B->width) {
            for (int i = 0; i < A->height; i++)
                Cvalue += Get_Mat_element(A, i, row) * Get_Mat_element(B, i, col);
            Set_Mat_element(C, row, col, Cvalue);
        }
    if (trans2 == true)
    if (row < A->height && col < B->height) {
        for (int i = 0; i < A->width; i++)
            Cvalue += Get_Mat_element(A, row, i) * Get_Mat_element(B, col, i);
        Set_Mat_element(C, row, col, Cvalue);
    }
}

__global__ void Multiplication_Kernel(Matrix *A, Matrix *B, Matrix *C) {
    double Cvalue = 0.0;
    int row = threadIdx.y + blockIdx.y * blockDim.y;
    int col = threadIdx.x + blockIdx.x * blockDim.x;
    if (row < A->height && col < A->width) {
        Cvalue = Get_Mat_element(A, row, col) * Get_Mat_element(B, row, col);
        Set_Mat_element(C, row, col, Cvalue);
    }
}

__global__ void Multiplication_Kernel(Matrix *A, double k) {
    double Avalue = 0.0;
    int row = threadIdx.y + blockIdx.y * blockDim.y;
    int col = threadIdx.x + blockIdx.x * blockDim.x;
    if (row < A->height && col < A->width) {
        Avalue = Get_Mat_element(A, row, col) * k;
        Set_Mat_element(A, row, col, Avalue);
    }
}

__global__ void Addition_Kernel(Matrix *A, Matrix *B, Matrix *C) {
    double Cvalue = 0.0;
    int row = threadIdx.y + blockIdx.y * blockDim.y;
    int col = threadIdx.x + blockIdx.x * blockDim.x;
    if (B->height == 1)
        if (row < A->height && col < A->width) {
            Cvalue = Get_Mat_element(A, row, col) + Get_Mat_element(B, 0, col);
            Set_Mat_element(C, row, col, Cvalue);
        }
}

__global__ void Subtraction_Kernel(Matrix *A, Matrix *B, Matrix *C) {
    double Cvalue = 0.0;
    int row = threadIdx.y + blockIdx.y * blockDim.y;
    int col = threadIdx.x + blockIdx.x * blockDim.x;
    if (row < A->height && col < A->width) {
        Cvalue = Get_Mat_element(A, row, col) - Get_Mat_element(B, row, col);
        Set_Mat_element(C, row, col, Cvalue);
    }
}
```

See matrix.cu file for detail implementations

MLP Training Functions:

- prepare_data(): to prepare the dataset for the training
- predict_y(): to predict the output of the model
- calculate_loss(): to calculate the loss of the neural network on a batch of data
- forward_propagation(): to pass the data through the model architecture
- backward_propagation(): for updating the weights.

See main.cu file for detail implementations

MLP Implementation in Pytorch

We implemented the same architecture with:

- the same hyperparameters
  (lr = 0.0001, Batch size = 64),
- optimization algorithm
  (Stochastic Gradient Descent)
  and
- same loss function
  (Cross entropy loss)

Results and Comparison