Machine Learning Final Project

(Topic 1 – Task 2)

Group Members:

- **Brij Popatbhai Patel (1146509)**
- **Aman Amrutbhai Patel (1150794)**

Topic 1 (Task 2): Object-centric Small Image recognition task with medium datasets (Datasets of Task 2)

Datasets of Task 2: CIFAR10, CIFAR100 (we have used built-in function to load the datasets in Tensorflow)

Platform Used: Google Colab Pro (GPU)

Final Accuracy Result: 86.5050 %

Methodology:

Considering project description and evaluation pattern we have selected 1st topic which is related to the object-centric recognition task to be more precise, we have considered task number two which is for medium datasets and performed recognition tasks on 2 datasets which are CIFAR10 and CIFAR100.

Firstly, we have imported the necessary libraries to perform the operation in which we have considered the “tensorflow_datasets” library to import cifar datasets. Then setting up the variables and gathered the datasets with the “load()” function.

Approaching the next step, we have used the **BiT model** [1] for transfer learning which can improve the accuracy significantly. To make the dataset more appropriate with the model, we have done some preprocessing tasks in which we have flipped the dataset features of training and testing data as well we have resized the image size from 32*32 to 224*224.

Feature Extraction Process: In this step, we have made two separate files for features and labels than perform the task on the training dataset and collected features by fitting training data into the BiT model. Apart from this, we have also used one-hot coding to get labels values.

In the main part of the code, we have done **ELM based CUDA C** programming in which firstly we have considered features and labels gathered from the dataset as described in the above process and passed it into the ELM CUDA model. This returns the result and we have got weights from it which can be used for training and testing of the model.

After that, newly generated weights have been transferred to the BiT model and updated the weights for further evaluation. In the last part, we have trained and tested the model by making a batch of original data in which we have implemented 5 runs and at the end, we have calculated the average of all 5 runs. And results for the tasks are as below.

Cifar10:

The CIFAR-10 dataset, which consists of 60000 32x32 colour images and is a subset of the Tiny Images dataset, has ten classes. The photographs are divided into ten categories: air plane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. Each class contains 6000 photographs, comprising 5000 training and 1000 assessment images [2].

1) Training Accuracy:

```
Training accuracy for  1  run: 0.9481
Training accuracy for  2  run: 0.9479
Training accuracy for  3  run: 0.9478
Training accuracy for  4  run: 0.9486
Training accuracy for  5  run: 0.9475
```

2) Testing Accuracy:

```
Testing accuracy for  1  run: 0.9363
Testing accuracy for  2  run: 0.9344
Testing accuracy for  3  run: 0.9361
Testing accuracy for  4  run: 0.9365
Testing accuracy for  5  run: 0.9360
```

3) Average Testing Accuracy for Cifar10 dataset:

Average accuracy of 5 run has been calculated and displayed as below and we got 93.58% average testing accuracy for cifar10 dataset.

```
[ ] average_testing = Average_acc(testing_accuracy)
    print("Average Testing Accuracy of 5 Runs = %.4f" % (float(average_testing)*100))

    Average Testing Accuracy of 5 Runs = 93.5860
```

Cifar100:

The CIFAR-100 dataset contains 60000 pictures and is a subset of the Images dataset. The CIFAR-100's 100 classes are divided into 20 superclasses. Each class has 600 photos in the class. Each class has 500 and 100 images for training and testing respectively [3]. I have considered the CIFAR-100 dataset by using the tensorflow library which automatically gives me training and testing separate dataset. Training and testing accuracy has been calculated by following methodology discussed above and we got the result as below.

1) Training Accuracy:

```
Training accuracy for  1  run: 0.8336
Training accuracy for  2  run: 0.8338
Training accuracy for  3  run: 0.8334
Training accuracy for  4  run: 0.8325
Training accuracy for  5  run: 0.8329
```

2) Testing Accuracy:

```
Testing accuracy for  1  run: 0.7940
Testing accuracy for  2  run: 0.7949
Testing accuracy for  3  run: 0.7934
Testing accuracy for  4  run: 0.7945
Testing accuracy for  5  run: 0.7944
```

3) Average Testing Accuracy for Cifar100 dataset:

Average accuracy of 5 run has been calculated and displayed as below and we got 79.42% average testing accuracy for cifar100 dataset.

```
average_testing = Average_acc(testing_accuracy)
print("Average Testing Accuracy of 5 Runs = %.4f" % (float(average_testing)*100))

Average Testing Accuracy of 5 Runs = 79.4240
```

Final Testing Accuracy Results (%):

Dataset	1st Run	2nd Run	3rd Run	4th Run	5th Run	Average Testing Accuracy
Cifar10	93.63	93.44	93.61	93.65	93.60	93.5860
Cifar100	79.40	79.49	79.34	79.45	79.44	79.4240

Average Top 1 Accuracy of Topic 1 Task 2:

Final Top 1 Accuracy: (Accuracy of Cifar10 + Accuracy of Cifar100)/2

: (93.5860 + 79.4240)/2

: 86.5050%

Source Code:

1) Cifar10:

```
import numpy as np
import tensorflow as tf
import tensorflow_hub as hub
import tensorflow_datasets as tfds
from keras.datasets import cifar10
import matplotlib.pyplot as plt

total_class = 10
value_batch = 64
new_size = 224

total_training_data = 50000
total_testing_data = 10000

url = "https://tfhub.dev/google/bit/m-r50x1/1"
module = hub.KerasLayer(url, trainable=False)

class bit_model(tf.keras.Model):
  def __init__(self, total_class, module):
    super().__init__()

    self.total_class = total_class
    self.head = tf.keras.layers.Dense(total_class, kernel_initializer='zeros', use_bias=False)
    self.bit_model = module
    self.bit_model.trainable = False

  def call(self, images):
    bit_embedding = self.bit_model(images)
    return bit_embedding, self.head(bit_embedding)

model = bit_model(total_class=total_class, module=module)


(training_data, testing_data), ds_info = tfds.load('cifar10', split=['train', 'test'],
shuffle_files=True, as_supervised=True, with_info=True)

def preprocess(cifar_data_features, cifar_data_label):
  cifar_data_features = tf.image.random_flip_left_right(cifar_data_features)
  cifar_data_features = tf.image.resize(cifar_data_features, [new_size, new_size])
  cifar_data_features = tf.cast(cifar_data_features, tf.float32) / 255.0
```

```
  return cifar_data_features, cifar_data_label

preprocess_training = (training_data.shuffle(total_training_data).map(preprocess,
num_parallel_calls=4).batch(value_batch).prefetch(2))

preprocess_testing = (testing_data.shuffle(total_testing_data).map(preprocess,
num_parallel_calls=1).batch(value_batch).prefetch(2))

def feat_lab(feat_training, lab_training):
  feature_string = ''
  label_string = ''
  for line in np.array(feat_training):
      for element in line:
         feature_string += (format(element, '.12f') + " ")
      feature_string += '\n'
  for line in np.array(lab_training):
      for element in line:
         label_string += (str(element) + " ")
      label_string += '\n'

  return feature_string, label_string

!rm -f cifar10_data_features.txt labels10.txt
feature_file = open('cifar10_data_features.txt', 'a')
label_file = open('labels10.txt', 'a')

for step, (training_Xbatch, training_Ybatch) in enumerate(preprocess_training):
    feat_training, _ = model(training_Xbatch, training=False)
    lab_training = tf.one_hot(training_Ybatch, total_class)
    feature_string, label_string = feat_lab(feat_training, lab_training)
    feature_file.write(feature_string)
    label_file.write(label_string)

feature_file.close()
label_file.close()

%%writefile elm.cu
#include <string>
#include <cuda_runtime.h>
#include <cublas_v2.h>

#include <iostream>
#include <fstream>
#include <string>
```

```
#include <sstream>

using namespace std;

void split_data(float* H_A, float* H, float* H_B, int feature_DimR, int feature_DimC){
   for(int i = 0; i < feature_DimR; i++) {
      for (int j = 0; j < feature_DimC; j++) {
         int x = i*feature_DimC;
         if (j < feature_DimC/2) H_A[x/2+j] = H[x+j];
         if (j >= feature_DimC/2) H_B[x/2+j-feature_DimC/2] = H[x+j];
      }
   }

}
void multiply_x_H(float* x_H,float* x_H_A,float* x_H_B,int feature_DimC,int feature_DimR){
   for(int i = 0; i < feature_DimC/2; i++) {
      for (int j = 0; j < feature_DimC; j++) {
         int x = i*feature_DimC;
         x_H[x + j] = x_H_A[x + j];
         x_H[(i+feature_DimC/2)*feature_DimC + j] = x_H_B[x + j];
     }
   }
   for(int i = 0; i < feature_DimC; i++) {
      for (int j = 0; j < feature_DimC; j++) {
         int x = i*feature_DimC;
         if (i == j) x_H[x + j] +=1;
     }
   }
}
float* cuda_def_elm(float* H, float* t, int feature_DimR, int feature_DimC, int number_labelC)
{
   cublasHandle_t cublas_handle;
   cublasCreate(&cublas_handle);

   cudaStream_t *strm = (cudaStream_t *) malloc(2*sizeof(cudaStream_t));
   cudaStreamCreate(&strm[0]);
   cudaStreamCreate(&strm[1]);

   size_t size_H_A = feature_DimR * feature_DimC/2 * sizeof(float);

   float* H_A = (float*) malloc(size_H_A);
   float* H_B = (float*) malloc(size_H_A);

   split_data(H_A,H,H_B,feature_DimR,feature_DimC);
```

```
   float* d_H_T; float* d_H; float* d_H_A; float* d_H_B; float* d_H_T_A; float* d_H_T_B;

   size_t size_H = feature_DimR * feature_DimC * sizeof(float);
   size_t size_H_T = feature_DimC * feature_DimC * sizeof(float);
   size_t size_H_T_A = feature_DimC * feature_DimC/2 * sizeof(float);

   cudaMalloc(&d_H, size_H);
   cudaMalloc(&d_H_T, size_H_T);
   cudaMalloc(&d_H_A, size_H_A);
   cudaMalloc(&d_H_B, size_H_A);
   cudaMalloc(&d_H_T_A, size_H_T_A);
   cudaMalloc(&d_H_T_B, size_H_T_A);

   cudaMemcpy(d_H, H, size_H, cudaMemcpyHostToDevice);
   cudaMemcpy(d_H_A, H_A, size_H_A, cudaMemcpyHostToDevice);
   cudaMemcpy(d_H_B, H_B, size_H_A, cudaMemcpyHostToDevice);

   float alpha = 1.0;
   float beta = 0.0;

   cublasSetStream(cublas_handle, strm[0]);
   cublasSgemm(cublas_handle, CUBLAS_OP_N, CUBLAS_OP_T, feature_DimC, feature_DimC/2,
feature_DimR, &alpha, d_H, feature_DimC, d_H_A, feature_DimC/2, &beta, d_H_T_A,
feature_DimC);
   cudaDeviceSynchronize();
   cublasSetStream(cublas_handle, strm[1]);
   cublasSgemm(cublas_handle, CUBLAS_OP_N, CUBLAS_OP_T, feature_DimC, feature_DimC/2,
feature_DimR, &alpha, d_H, feature_DimC, d_H_B, feature_DimC/2, &beta, d_H_T_B,
feature_DimC);
   cudaDeviceSynchronize();

   float* x_H_A = (float*) malloc(size_H_T_A);
   float* x_H_B = (float*) malloc(size_H_T_A);
   cudaMemcpy(x_H_A, d_H_T_A, size_H_T_A, cudaMemcpyDeviceToHost);
   cudaMemcpy(x_H_B, d_H_T_B, size_H_T_A, cudaMemcpyDeviceToHost);
   float* x_H = (float*) malloc(size_H_T);
   multiply_x_H(x_H, x_H_A, x_H_B, feature_DimC, feature_DimR);
   cudaMemcpy(d_H_T, x_H, size_H_T, cudaMemcpyHostToDevice);

   float** a_bb; float** a_cc; float* b_cc;
   int* a_pivot; int* a_info;

   size_t size_A = feature_DimC * feature_DimC * sizeof(float);
```

```
    cudaMalloc(&a_bb, sizeof(float*));
    cudaMalloc(&a_cc, sizeof(float*));
    cudaMalloc(&b_cc, size_A);
    cudaMalloc(&a_pivot, feature_DimC * sizeof(float));
    cudaMalloc(&a_info, sizeof(float));

    cudaMemcpy(a_bb, &d_H_T, sizeof(float*), cudaMemcpyHostToDevice);
    cudaMemcpy(a_cc, &b_cc, sizeof(float*), cudaMemcpyHostToDevice);

    cublasSgetrfBatched(cublas_handle, feature_DimC, a_bb, feature_DimC, a_pivot, a_info, 1);
    cudaDeviceSynchronize();

    cublasSgetriBatched(cublas_handle, feature_DimC, (const float **)a_bb, feature_DimC,
a_pivot, a_cc, feature_DimC, a_info, 1);
    cudaDeviceSynchronize();

    cudaFree(a_bb);
    cudaFree(a_cc);
    cudaFree(a_pivot);
    cudaFree(a_info);

    float* b_H_T;
    size_t size_bH_T = feature_DimR * feature_DimC * sizeof(float);
    cudaMalloc(&b_H_T, size_bH_T);

    cublasSgemm(cublas_handle, CUBLAS_OP_N, CUBLAS_OP_N, feature_DimC, feature_DimR,
feature_DimC, &alpha, b_cc, feature_DimC, d_H, feature_DimC, &beta, b_H_T, feature_DimC);
    cudaDeviceSynchronize();

    float* c_HtD;
    size_t size_tD = feature_DimR * number_labelC * sizeof(float);
    cudaMalloc(&c_HtD, size_tD);
    cudaMemcpy(c_HtD, t, size_tD, cudaMemcpyHostToDevice);

    float* c_DtH;
    size_t size_tH = feature_DimR * number_labelC * sizeof(float);
    cudaMalloc(&c_DtH, size_tH);

    cublasSgemm(cublas_handle, CUBLAS_OP_N, CUBLAS_OP_T, number_labelC, feature_DimC,
feature_DimR, &alpha, c_HtD, number_labelC, b_H_T, feature_DimC, &beta, c_DtH,
number_labelC);
    cudaDeviceSynchronize();
```

```
    float* final_result = (float*) malloc(size_tH);
    cudaMemcpy(final_result, c_DtH, size_tH, cudaMemcpyDeviceToHost);

    cublasDestroy(cublas_handle);
    return final_result;
}
void load_feature(float* H,float* t,string a,int feature_DimC,int feature_DimR){
    string line;
    ifstream featfile(a);
    if(featfile.is_open())
    {
        int r = 0;
        char delim='\n';
        while (getline(featfile, line, delim)) {
            string subs;
            int c = 0;
            string delimiter = " ";
            size_t position = 0;
            string token;
            while ((position = line.find(delimiter)) != string::npos) {
                token = line.substr(0, position);
                H[r*feature_DimC + c] = stof(token);
                line.erase(0, position + delimiter.length());
                c++;
            }
            r++;
        }
        featfile.close();
    }
}
void load_label(float* H,float* t,string b,int number_labelC,int feature_DimR){
    string line;
    ifstream labelfile(b);
    if(labelfile.is_open())
    {
        int r = 0;
        char delim='\n';
        while (getline(labelfile, line, delim)) {
            string subs;
            int c = 0;
            string delimiter = " ";
            size_t position = 0;
            string token;
            while ((position = line.find(delimiter)) != string::npos) {
```

```
            token = line.substr(0, position);
            t[r*number_labelC + c] = stof(token);
            line.erase(0, position + delimiter.length());
            c++;
          }
          r++;
        }
        labelfile.close();
    }
}
int main(int argc, char *argv[])
{

    int feature_DimR = 50000;
    int feature_DimC = 2048;
    int number_labelC = 10;

    float* H = (float*) malloc(feature_DimR * feature_DimC * sizeof(float));
    float* t = (float*) malloc(feature_DimR * number_labelC * sizeof(float));

    load_feature(H, t, "cifar10_data_features.txt", feature_DimC, feature_DimR);
    load_label(H, t, "labels10.txt", number_labelC, feature_DimR);

    float* res = cuda_def_elm(H, t, feature_DimR, feature_DimC, number_labelC);

    ofstream result;
    result.open ("trained_weights10.txt");
    for (int i=0; i<feature_DimC; i++){
        for (int j=0; j<number_labelC; j++){
            result << res[j+number_labelC*i];
            result << " ";
        }
        result << endl;
    }
    result.close();
    return 0;
}
```

```
!nvcc -o elm ./elm.cu -lcublas
```

```
!./elm
```

```
weights = np.zeros((2048, 10), dtype=np.float)
with open('trained_weights10.txt') as f:
```

```
    for i, row in enumerate(f.readlines()):
        for j, w in enumerate(row.split()):
            weights[i,j] = float(w)

accuracy_train_m = tf.keras.metrics.SparseCategoricalAccuracy()
accuracy_test_m = tf.keras.metrics.SparseCategoricalAccuracy()

model.head.set_weights([weights])

i = 1
training_accuracy = []
while i < 6:
  for training_Xbatch, training_Ybatch in preprocess_training:
    _, l1 = model(training_Xbatch, training=False)
    accuracy_train_m.update_state(training_Ybatch, l1)
  t_acc = accuracy_train_m.result()
  training_accuracy.append(t_acc)
  accuracy_train_m.reset_states()
  print("Training accuracy for ",str(i)," run: %.4f" % (float(t_acc)))
  i += 1

j = 1
testing_accuracy = []
while j < 6:
  for testing_Xbatch, testing_Ybatch in preprocess_testing:
    _, l1 = model(testing_Xbatch, training=False)
    accuracy_test_m.update_state(testing_Ybatch, l1)
  te_acc = accuracy_test_m.result()
  testing_accuracy.append(te_acc)
  accuracy_test_m.reset_states()
  print("Testing accuracy for ",str(j)," run: %.4f" % (float(te_acc),))
  j += 1

def Average_acc(lst):
    return sum(lst) / len(lst)

average_training = Average_acc(training_accuracy)
print("Average Training Accuracy of 5 Runs = %.4f" % (float(average_training)*100))

average_testing = Average_acc(testing_accuracy)
print("Average Testing Accuracy of 5 Runs = %.4f" % (float(average_testing)*100))

x = [1, 2, 3, 4, 5]
plt.plot(x, training_accuracy, label = "Training Accuracy")
```

```
plt.plot(x, testing_accuracy, label = "Testing Accuracy")
plt.xlim([1, 5])
plt.ylim([0.85, 1.00])
plt.legend()
plt.show()
```

2) Cifar100:

```
import numpy as np
import tensorflow as tf
import tensorflow_hub as hub
import tensorflow_datasets as tfds
from keras.datasets import cifar100
import matplotlib.pyplot as plt

total_class = 100
value_batch = 64
new_size = 224

total_training_data = 50000
total_testing_data = 10000

url = "https://tfhub.dev/google/bit/m-r50x1/1"
module = hub.KerasLayer(url, trainable=False)

class bit_model(tf.keras.Model):
  def __init__(self, total_class, module):
    super().__init__()

    self.total_class = total_class
    self.head = tf.keras.layers.Dense(total_class, kernel_initializer='zeros', use_bias=False)
    self.bit_model = module
    self.bit_model.trainable = False

  def call(self, images):
    bit_embedding = self.bit_model(images)
    return bit_embedding, self.head(bit_embedding)

model = bit_model(total_class=total_class, module=module)

(training_data, testing_data), ds_info = tfds.load('cifar100', split=['train', 'test'],
shuffle_files=True, as_supervised=True, with_info=True)

def preprocess(cifar_data_features, cifar_data_label):
```

```
  cifar_data_features = tf.image.random_flip_left_right(cifar_data_features)
  cifar_data_features = tf.image.resize(cifar_data_features, [new_size, new_size])
  cifar_data_features = tf.cast(cifar_data_features, tf.float32) / 255.0
  return cifar_data_features, cifar_data_label

preprocess_training = (training_data.shuffle(total_training_data).map(preprocess,
num_parallel_calls=4).batch(value_batch).prefetch(2))

preprocess_testing = (testing_data.shuffle(total_testing_data).map(preprocess,
num_parallel_calls=1).batch(value_batch).prefetch(2))

def feat_lab(feat_training, lab_training):
  feature_string = ''
  label_string = ''
  for line in np.array(feat_training):
      for element in line:
         feature_string += (format(element, '.12f') + " ")
      feature_string += '\n'
  for line in np.array(lab_training):
      for element in line:
         label_string += (str(element) + " ")
      label_string += '\n'

  return feature_string, label_string

!rm -f cifar_data_features.txt labels.txt
feature_file = open('cifar_data_features.txt', 'a')
label_file = open('labels.txt', 'a')

for step, (training_Xbatch, training_Ybatch) in enumerate(preprocess_training):
    feat_training, _ = model(training_Xbatch, training=False)
    lab_training = tf.one_hot(training_Ybatch, total_class)
    feature_string, label_string = feat_lab(feat_training, lab_training)
    feature_file.write(feature_string)
    label_file.write(label_string)

feature_file.close()
label_file.close()

%%writefile elm.cu
#include <string>
#include <cuda_runtime.h>
#include <cublas_v2.h>
```

```
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>

using namespace std;

void split_data(float* H_A, float* H, float* H_B, int feature_DimR, int feature_DimC){
  for(int i = 0; i < feature_DimR; i++) {
    for (int j = 0; j < feature_DimC; j++) {
      int x = i*feature_DimC;
      if (j < feature_DimC/2) H_A[x/2+j] = H[x+j];
      if (j >= feature_DimC/2) H_B[x/2+j-feature_DimC/2] = H[x+j];
    }
  }

}
void multiply_x_H(float* x_H,float* x_H_A,float* x_H_B,int feature_DimC,int feature_DimR){
  for(int i = 0; i < feature_DimC/2; i++) {
    for (int j = 0; j < feature_DimC; j++) {
      int x = i*feature_DimC;
      x_H[x + j] = x_H_A[x + j];
      x_H[(i+feature_DimC/2)*feature_DimC + j] = x_H_B[x + j];
   }
  }
  for(int i = 0; i < feature_DimC; i++) {
    for (int j = 0; j < feature_DimC; j++) {
      int x = i*feature_DimC;
      if (i == j) x_H[x + j] +=1;
   }
  }
}


float* cuda_def_elm(float* H, float* t, int feature_DimR, int feature_DimC, int number_labelC)
{
  cublasHandle_t cublas_handle;
  cublasCreate(&cublas_handle);

  cudaStream_t *strm = (cudaStream_t *) malloc(2*sizeof(cudaStream_t));
  cudaStreamCreate(&strm[0]);
  cudaStreamCreate(&strm[1]);
```

```
    size_t size_H_A = feature_DimR * feature_DimC/2 * sizeof(float);

    float* H_A = (float*) malloc(size_H_A);
    float* H_B = (float*) malloc(size_H_A);

    split_data(H_A,H,H_B,feature_DimR,feature_DimC);

    float* d_H_T; float* d_H; float* d_H_A; float* d_H_B; float* d_H_T_A; float* d_H_T_B;

    size_t size_H = feature_DimR * feature_DimC * sizeof(float);
    size_t size_H_T = feature_DimC * feature_DimC * sizeof(float);
    size_t size_H_T_A = feature_DimC * feature_DimC/2 * sizeof(float);

    cudaMalloc(&d_H, size_H);
    cudaMalloc(&d_H_T, size_H_T);
    cudaMalloc(&d_H_A, size_H_A);
    cudaMalloc(&d_H_B, size_H_A);
    cudaMalloc(&d_H_T_A, size_H_T_A);
    cudaMalloc(&d_H_T_B, size_H_T_A);

    cudaMemcpy(d_H, H, size_H, cudaMemcpyHostToDevice);
    cudaMemcpy(d_H_A, H_A, size_H_A, cudaMemcpyHostToDevice);
    cudaMemcpy(d_H_B, H_B, size_H_A, cudaMemcpyHostToDevice);

    float alpha = 1.0;
    float beta = 0.0;

    cublasSetStream(cublas_handle, strm[0]);
    cublasSgemm(cublas_handle, CUBLAS_OP_N, CUBLAS_OP_T, feature_DimC, feature_DimC/2,
feature_DimR, &alpha, d_H, feature_DimC, d_H_A, feature_DimC/2, &beta, d_H_T_A,
feature_DimC);
    cudaDeviceSynchronize();
    cublasSetStream(cublas_handle, strm[1]);
    cublasSgemm(cublas_handle, CUBLAS_OP_N, CUBLAS_OP_T, feature_DimC, feature_DimC/2,
feature_DimR, &alpha, d_H, feature_DimC, d_H_B, feature_DimC/2, &beta, d_H_T_B,
feature_DimC);
    cudaDeviceSynchronize();

    float* x_H_A = (float*) malloc(size_H_T_A);
    float* x_H_B = (float*) malloc(size_H_T_A);
    cudaMemcpy(x_H_A, d_H_T_A, size_H_T_A, cudaMemcpyDeviceToHost);
    cudaMemcpy(x_H_B, d_H_T_B, size_H_T_A, cudaMemcpyDeviceToHost);
    float* x_H = (float*) malloc(size_H_T);
```

```
    multiply_x_H(x_H, x_H_A, x_H_B, feature_DimC, feature_DimR);

    cudaMemcpy(d_H_T, x_H, size_H_T, cudaMemcpyHostToDevice);

    float** a_bb; float** a_cc; float* b_cc;
    int* a_pivot; int* a_info;

    size_t size_A = feature_DimC * feature_DimC * sizeof(float);

    cudaMalloc(&a_bb, sizeof(float*));
    cudaMalloc(&a_cc, sizeof(float*));
    cudaMalloc(&b_cc, size_A);
    cudaMalloc(&a_pivot, feature_DimC * sizeof(float));
    cudaMalloc(&a_info, sizeof(float));

    cudaMemcpy(a_bb, &d_H_T, sizeof(float*), cudaMemcpyHostToDevice);
    cudaMemcpy(a_cc, &b_cc, sizeof(float*), cudaMemcpyHostToDevice);

    cublasSgetrfBatched(cublas_handle, feature_DimC, a_bb, feature_DimC, a_pivot, a_info, 1);
    cudaDeviceSynchronize();

    cublasSgetriBatched(cublas_handle, feature_DimC, (const float **)a_bb, feature_DimC,
a_pivot, a_cc, feature_DimC, a_info, 1);
    cudaDeviceSynchronize();

    cudaFree(a_bb);
    cudaFree(a_cc);
    cudaFree(a_pivot);
    cudaFree(a_info);

    float* b_H_T;
    size_t size_bH_T = feature_DimR * feature_DimC * sizeof(float);
    cudaMalloc(&b_H_T, size_bH_T);

    cublasSgemm(cublas_handle, CUBLAS_OP_N, CUBLAS_OP_N, feature_DimC, feature_DimR,
feature_DimC, &alpha, b_cc, feature_DimC, d_H, feature_DimC, &beta, b_H_T, feature_DimC);
    cudaDeviceSynchronize();

    float* c_HtD;
    size_t size_tD = feature_DimR * number_labelC * sizeof(float);
    cudaMalloc(&c_HtD, size_tD);
    cudaMemcpy(c_HtD, t, size_tD, cudaMemcpyHostToDevice);

    float* c_DtH;
```

```
    size_t size_tH = feature_DimR * number_labelC * sizeof(float);
    cudaMalloc(&c_DtH, size_tH);

    cublasSgemm(cublas_handle, CUBLAS_OP_N, CUBLAS_OP_T, number_labelC, feature_DimC,
feature_DimR, &alpha, c_HtD, number_labelC, b_H_T, feature_DimC, &beta, c_DtH,
number_labelC);
    cudaDeviceSynchronize();

    float* final_result = (float*) malloc(size_tH);
    cudaMemcpy(final_result, c_DtH, size_tH, cudaMemcpyDeviceToHost);

    cublasDestroy(cublas_handle);
    return final_result;
}
void load_feature(float* H,float* t,string a,int feature_DimC,int feature_DimR){
    string line;
    ifstream featfile(a);
    if(featfile.is_open())
    {
        int r = 0;
        char delim='\n';
        while (getline(featfile, line, delim)) {
            string subs;
            int c = 0;
            string delimiter = " ";
            size_t position = 0;
            string token;
            while ((position = line.find(delimiter)) != string::npos) {
                token = line.substr(0, position);
                H[r*feature_DimC + c] = stof(token);
                line.erase(0, position + delimiter.length());
                c++;
            }
            r++;
        }
        featfile.close();
    }
}
void load_label(float* H,float* t,string b,int number_labelC,int feature_DimR){
    string line;
    ifstream labelfile(b);
    if(labelfile.is_open())
    {
        int r = 0;
```

```
        char delim='\n';
        while (getline(labelfile, line, delim)) {
            string subs;
            int c = 0;
            string delimiter = " ";
            size_t position = 0;
            string token;
            while ((position = line.find(delimiter)) != string::npos) {
                token = line.substr(0, position);
                t[r*number_labelC + c] = stof(token);
                line.erase(0, position + delimiter.length());
                c++;
            }
            r++;
        }
        labelfile.close();
    }
}
int main(int argc, char *argv[])
{

    int feature_DimR = 50000;
    int feature_DimC = 2048;
    int number_labelC = 100;

    float* H = (float*) malloc(feature_DimR * feature_DimC * sizeof(float));
    float* t = (float*) malloc(feature_DimR * number_labelC * sizeof(float));

    load_feature(H, t, "cifar_data_features.txt", feature_DimC, feature_DimR);
    load_label(H, t, "labels.txt", number_labelC, feature_DimR);

    float* res = cuda_def_elm(H, t, feature_DimR, feature_DimC, number_labelC);

    ofstream result;
    result.open ("trained_weights.txt");
    for (int i=0; i<feature_DimC; i++){
        for (int j=0; j<number_labelC; j++){
            result << res[j+number_labelC*i];
            result << " ";
        }
        result << endl;
    }
    result.close();
    return 0;
```

```
}

!nvcc -o elm ./elm.cu -lcublas

!./elm

weights = np.zeros((2048, 100), dtype=np.float)
with open('trained_weights.txt') as f:
    for i, row in enumerate(f.readlines()):
        for j, w in enumerate(row.split()):
            weights[i,j] = float(w)

accuracy_train_m = tf.keras.metrics.SparseCategoricalAccuracy()
accuracy_test_m = tf.keras.metrics.SparseCategoricalAccuracy()

model.head.set_weights([weights])

i = 1
training_accuracy = []
while i < 6:
  for training_Xbatch, training_Ybatch in preprocess_training:
    _, l1 = model(training_Xbatch, training=False)
    accuracy_train_m.update_state(training_Ybatch, l1)
  t_acc = accuracy_train_m.result()
  training_accuracy.append(t_acc)
  accuracy_train_m.reset_states()
  print("Training accuracy for ",str(i)," run: %.4f" % (float(t_acc)))
  i += 1

j = 1
testing_accuracy = []
while j < 6:
  for testing_Xbatch, testing_Ybatch in preprocess_testing:
    _, l1 = model(testing_Xbatch, training=False)
    accuracy_test_m.update_state(testing_Ybatch, l1)
  te_acc = accuracy_test_m.result()
  testing_accuracy.append(te_acc)
  accuracy_test_m.reset_states()
  print("Testing accuracy for ",str(j)," run: %.4f" % (float(te_acc),))
  j += 1

def Average_acc(lst):
    return sum(lst) / len(lst)
```

```
average_training = Average_acc(training_accuracy)
print("Average Training Accuracy of 5 Runs = %.4f" % (float(average_training)*100))

average_testing = Average_acc(testing_accuracy)
print("Average Testing Accuracy of 5 Runs = %.4f" % (float(average_testing)*100))

x = [1, 2, 3, 4, 5]
plt.plot(x, training_accuracy, label = "Training Accuracy")
plt.plot(x, testing_accuracy, label = "Testing Accuracy")
plt.xlim([1, 5])
plt.ylim([0.75, 0.90])
plt.legend()
plt.show()
```

References:

[1] Jessica Yung and Joan Puigcerver, "BigTransfer (BiT): State-of-the-art transfer learning for computer vision", TensorFlow Blog. May 20,2020, Available: https://blog.tensorflow.org/2020/05/bigtransfer-bit-state-of-art-transfer-learning-computer-vision.html

[2] "CIFAR-10", Papers with codes. Available: https://paperswithcode.com/dataset/cifar-10

[3] "CIFAR-100", Papers with codes. Available: https://paperswithcode.com/dataset/cifar-100