COMP-5011-FDE Machine Learning & Neural Network

Project

Name: Khushal Paresh Thaker

Student ID: 1106937

Task: Object-centric Small Image recognition task with medium datasets

Dataset: CIFAR10, CIFAR100

Project Report

Module Descriptions:

The project coding is done in google colab workseets using tensorflow based python and cuda c programming. To understand the project code, it can be divided into sections of:

1. Library imports
2. BiT model load
3. Dataset loading and necessary preprocessing
4. Cuda C based ELM code
5. Training the model
6. Testing the model

Code can be run in google colab. Attached files – 1106937_ML_Project_cifar10.ipynb and 1106937_ML_Project_cifar100.ipynb can be uploaded and run with *'GPU'* as the runtime. Also, attached are the respective python files for reference.

Content of the report:

1. Module description with average training and testing accuracies for cifar10 and cifar100 datasets after three runs
2. Source Code with output screenshot

Module Description

Library Imports

Code begins by importing the necessary libraries to run the code in python. There are variables created for the program such as number of classes, batch size, etc.

BiT Model Load

We use Big Transfer (BiT), a pre-trained model proposed by the google brain for deep learning computer vision problems to improve the accuracy. The model is obtained from its link: https://tfhub.dev/google/bit/m-r50x1/1

function used:

def call(self, images)-> It is a normal method to call the BiT model

Dataset loading and necessary preprocessing

We use the tensorflow libraries to load the existing train and test dataset of cifar. This is followed by preprocessing each of it that includes dividing the pixels by 255 so that they are in [0,1] range. The images are also upscaled from 32*32 to 128*128 which helped achieve higher accuracy.

functions used:

def cast_to_tuple(features): -> returns images and its respective labels
def trainSet_preprocess(features, label): -> resize and randomize the selection of training dataset
def testSet_preprocess(features, label): -> similar for test dataset

Cuda C based ELM code

We now extract all the training data features and labels and store them into files named 'features.txt' and 'labels.txt', respectively. These two files will be opened by the CUDA-based ELM, after which it will write the output into another file called 'weights.txt' which be read later to evaluate the model.

The next module is the Cuda C based ELM code where the training data feature and labels are read initially. It makes use of cublas and standard c libraries. Multi-streams is used in order to achieve higher accuracy. Multi streams allow concurrent streams to access certain part of the dataset so that there are no issues while running the code. The ELM code makes use of transposing and inversing the matrix to provide the Pseudo Inverse Hessian matrix. Once the elm part is coded, the labels are read and weights are generated by run the cuda based elm. These weights are stored in a file as well which will be used to evaluate the model later.

Training and Testing

The weights stored are used to evaluate the model for the training dataset as well as the testing dataset.
The model for **cifar10 dataset** is run *thrice* and the
Training accuracy for first run: 93.1260%
Training accuracy for second run: 93.1840%
Training accuracy for third run: 93.1820%

Average Training accuracy achieved is 93.1640%

Testing accuracy for first run: 91.84%
Testing accuracy for second run: 91.84%
Testing accuracy for third run: 91.84%
Average Testing accuracy achieved is 91.84%

The model for **cifar100 dataset** is run *thrice* and the
Training accuracy for first run: 80.1420%
Training accuracy for second run: 80.0440%
Training accuracy for third run: 79.9620%

Average Training accuracy achieved is 80.0493%

Testing accuracy for first run: 75.46%
Testing accuracy for second run: 75.46%
Testing accuracy for third run: 75.46%
Average Testing accuracy achieved is 75.46%

The lower accuracy for cifar100 dataset model could be because of the number of classes.

Experimental Results:

The code *(which is not present in the colab notebook)* was initially run without upscaling and without using multistreams. The accuracies were very low and once code was upscaled and multistreams added, accuracy increased as shown in the table below

Model	**Dataset**	**Avg. Training Accuracy**	**Avg. Testing Accuracy**
Without upscaling and multistreams	Cifar10	79.7034%	76.13%
	Cifar100	61.4598%	60.33%
With upscaling and multistreams	Cifar10	93.164%	91.84%
	Cifar100	80.0493%	75.46%

Source Code:

Cifar10:

```
# Importing necessary libraries
import tensorflow as tf
import tensorflow_hub as hub
import tensorflow_datasets as tfds
import numpy as np
# setting the constant values used in the rest of the code.
NUM_CLASSES = 10
BATCH_SIZE = 1000
DATASET_NUM_TRAIN_EXAMPLES = 50000
RESIZE_TO = 128
# Using the state-of-the-art model - Big Transfer (BiT) for acheiving high accuracy.
class FeatureNet(tf.keras.Model):
  def __init__(self, num_classes, module):
    super().__init__()

    self.num_classes = num_classes
    self.head = tf.keras.layers.Dense(num_classes, kernel_initializer='zeros', use_bias=False)
    self.bit_model = module
    self.bit_model.trainable = False #freeze the backbone network

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

# We use the state-of-the-art model for transfer learning from Google: Big Transfer
model_url = "https://tfhub.dev/google/bit/m-r50x1/1"
module = hub.KerasLayer(model_url, trainable=False)
model = FeatureNet(num_classes=NUM_CLASSES, module=module)
# loading the train and test set from cifar100 dataset
(ds_train, ds_test), ds_info = tfds.load(
    'cifar10',
    split=['train', 'test'],
    shuffle_files=True,
    as_supervised=True,
    with_info=True,
)

# printing information about the cifar10 dataset
ds_info
# using dataframe functionality to understand the dataset
tfds.as_dataframe(ds_train.take(5), ds_info)
def cast_to_tuple(features):
  return (features['image'], features['label'])

def trainSet_preprocess(features, label):
```

```
  features = tf.image.random_flip_left_right(features)
  features = tf.image.resize(features, [RESIZE_TO, RESIZE_TO])
  features = tf.cast(features, tf.float32) / 255.0
  return features, label

def testSet_preprocess(features, label):
  features = tf.image.resize(features, [RESIZE_TO, RESIZE_TO])
  features = tf.cast(features, tf.float32) / 255.0
  return features, label

trainSet_processFlow = (ds_train
                .shuffle(DATASET_NUM_TRAIN_EXAMPLES)
                .map(trainSet_preprocess, num_parallel_calls=4)
                .batch(BATCH_SIZE)
                .prefetch(2))

testSet_processFlow = (ds_test.map(testSet_preprocess, num_parallel_calls=1)
                .batch(BATCH_SIZE)
                .prefetch(2))
!rm -f features.txt labels.txt
f_f = open('features.txt', 'a')
l_f = open('labels.txt', 'a')

for step, (x_batch_train, y_batch_train) in enumerate(trainSet_processFlow):
    # extracting features
    H, _ = model(x_batch_train, training=False)
    argstr_feats = ''
    argstr_labels = ''
    t = tf.one_hot(y_batch_train, NUM_CLASSES)
    # Write the features into file
    for row in np.array(H):
        for element in row:
            argstr_feats += (format(element, '.12f') + " ")
        argstr_feats += '\n'
    f_f.write(argstr_feats)
    # Write the labels into file
    for row in np.array(t):
        for element in row:
            argstr_labels += (str(element) + " ")
        argstr_labels += '\n'
    l_f.write(argstr_labels)

f_f.close()
l_f.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;

#define CUDA_CALL(res, str) { if (res != cudaSuccess) { printf("CUDA Error : %s : %s %d : ERR %s\n", st
r, __FILE__, __LINE__, cudaGetErrorName(res)); } }
#define CUBLAS_CALL(res, str) { if (res != CUBLAS_STATUS_SUCCESS) { printf("CUBLAS Error : %s : %s
%d : ERR %d\n", str, __FILE__, __LINE__, int(res)); } }


float* d_ELM(float* H, float* t, int feat_rows, int feat_cols, int label_cols)
{
  cublasHandle_t cu_cublasHandle;
  CUBLAS_CALL(cublasCreate(&cu_cublasHandle), "Failed to initialize cuBLAS!");

  // creating multistreams
  cudaStream_t *streams = (cudaStream_t *) malloc(2*sizeof(cudaStream_t));
  cudaStreamCreate(&streams[0]);
  cudaStreamCreate(&streams[1]);

  size_t szH = feat_rows * feat_cols * sizeof(float);
  size_t szHTH = feat_cols * feat_cols * sizeof(float);
  size_t szH1 = feat_rows * feat_cols/2 * sizeof(float);
  size_t szHTH1 = feat_cols * feat_cols/2 * sizeof(float);

  // Start of multiplication to obtain HtH
  float* H1 = (float*) malloc(szH1);
  float* H2 = (float*) malloc(szH1);

  // Split the features data into two matrices
  for(int i = 0; i < feat_rows; i++) {
    for (int j = 0; j < feat_cols; j++) {
      if (j < feat_cols/2) H1[i*feat_cols/2+j] = H[i*feat_cols+j];
      if (j >= feat_cols/2) H2[i*feat_cols/2+j-feat_cols/2] = H[i*feat_cols+j];
    }
  }

  // Allocate the variables for computing HTH
  float* dHTH;
  float* dH;
  float* dH1;
  float* dH2;
  float* dHTH1;
  float* dHTH2;
```

```
    CUDA_CALL(cudaMalloc(&dH, szH), "Failed to allocate H!");
    CUDA_CALL(cudaMalloc(&dHTH, szHTH), "Failed to allocate dHTH!");
    CUDA_CALL(cudaMalloc(&dH1, szH1), "Failed to allocate dH1");
    CUDA_CALL(cudaMalloc(&dH2, szH1), "Failed to allocate dH2");
    CUDA_CALL(cudaMalloc(&dHTH1, szHTH1), "Failed to allocate dHTH1");
    CUDA_CALL(cudaMalloc(&dHTH2, szHTH1), "Failed to allocate dHTH2");

    CUDA_CALL(cudaMemcpy(dH, H, szH, cudaMemcpyHostToDevice), "Failed to copy to dH!");
    CUDA_CALL(cudaMemcpy(dH1, H1, szH1, cudaMemcpyHostToDevice), "Failed to copy to dH1!");
    CUDA_CALL(cudaMemcpy(dH2, H2, szH1, cudaMemcpyHostToDevice), "Failed to copy to dH2!");

    float alpha = 1.0;
    float beta = 0.0;

    // Multiplication with two CUDA streams. We split the H matrix into two halves.

    cublasSetStream(cu_cublasHandle, streams[0]);
    CUBLAS_CALL(cublasSgemm(cu_cublasHandle, CUBLAS_OP_N, CUBLAS_OP_T, feat_cols, feat_cols
/2, feat_rows, &alpha, dH, feat_cols, dH1, feat_cols/2, &beta, dHTH1, feat_cols), "Failed to call matri
x mult");
    CUDA_CALL(cudaDeviceSynchronize(), "Failed to synchronize after kernel call!");
    cublasSetStream(cu_cublasHandle, streams[1]);
    CUBLAS_CALL(cublasSgemm(cu_cublasHandle, CUBLAS_OP_N, CUBLAS_OP_T, feat_cols, feat_cols
/2, feat_rows, &alpha, dH, feat_cols, dH2, feat_cols/2, &beta, dHTH2, feat_cols), "Failed to call matri
x mult");
    CUDA_CALL(cudaDeviceSynchronize(), "Failed to synchronize after kernel call!");

    float* HTH1 = (float*) malloc(szHTH1);
    float* HTH2 = (float*) malloc(szHTH1);
    CUDA_CALL(cudaMemcpy(HTH1, dHTH1, szHTH1, cudaMemcpyDeviceToHost), "Failed to copy to r
es!");
    CUDA_CALL(cudaMemcpy(HTH2, dHTH2, szHTH1, cudaMemcpyDeviceToHost), "Failed to copy to r
es!");
    float* HTH = (float*) malloc(szHTH);
    for(int i = 0; i < feat_cols/2; i++) {
       for (int j = 0; j < feat_cols; j++) {
          HTH[i*feat_cols + j] = HTH1[i*feat_cols + j];
          HTH[(i+feat_cols/2)*feat_cols + j] = HTH2[i*feat_cols + j];
     }
    }
    for(int i = 0; i < feat_cols; i++) {
       for (int j = 0; j < feat_cols; j++) {
          if (i == j) HTH[i*feat_cols + j] +=1;
     }
    }
    CUDA_CALL(cudaMemcpy(dHTH, HTH, szHTH, cudaMemcpyHostToDevice), "Failed to copy to HTH
!");
    // End of multiplication to obtain HTH
```

```
  // Matrix Inversion using LU decomposition

  float** adL;
  float** adC;
  float* dC;
  int* dLUPivots;
  int* dLUInfo;

  size_t szA = feat_cols * feat_cols * sizeof(float);

  CUDA_CALL(cudaMalloc(&adL, sizeof(float*)), "Failed to allocate adL!");
  CUDA_CALL(cudaMalloc(&adC, sizeof(float*)), "Failed to allocate adC!");
  CUDA_CALL(cudaMalloc(&dC, szA), "Failed to allocate dC!");
  CUDA_CALL(cudaMalloc(&dLUPivots, feat_cols * sizeof(float)), "Failed to allocate dLUPivots!");
  CUDA_CALL(cudaMalloc(&dLUInfo, sizeof(float)), "Failed to allocate dLUInfo!");

  CUDA_CALL(cudaMemcpy(adL, &dHTH, sizeof(float*), cudaMemcpyHostToDevice), "Failed to copy
 to adL!");
  CUDA_CALL(cudaMemcpy(adC, &dC, sizeof(float*), cudaMemcpyHostToDevice), "Failed to copy to
 adC!");

  // We call the CUBLAS LU decomposition
  CUBLAS_CALL(cublasSgetrfBatched(cu_cublasHandle, feat_cols, adL, feat_cols, dLUPivots, dLUInfo,
 1), "Failed to perform LU decomp operation!");
  CUDA_CALL(cudaDeviceSynchronize(), "Failed to synchronize after kernel call!");

  CUBLAS_CALL(cublasSgetriBatched(cu_cublasHandle, feat_cols, (const float **)adL, feat_cols, dLU
Pivots, adC, feat_cols, dLUInfo, 1), "Failed to perform Inverse operation!");
  CUDA_CALL(cudaDeviceSynchronize(), "Failed to synchronize after kernel call!");

  CUDA_CALL(cudaFree(adL), "Failed to free adL!");
  CUDA_CALL(cudaFree(adC), "Failed to free adC!");
  CUDA_CALL(cudaFree(dLUPivots), "Failed to free dLUPivots!");
  CUDA_CALL(cudaFree(dLUInfo), "Failed to free dLUInfo!");

  // Multiplication of (HTH)^-1 and HT
  float* dpHT;
  size_t szpHT = feat_rows * feat_cols * sizeof(float);
  CUDA_CALL(cudaMalloc(&dpHT, szpHT), "Failed to allocate pHT!");

  CUBLAS_CALL(cublasSgemm(cu_cublasHandle, CUBLAS_OP_N, CUBLAS_OP_N, feat_cols, feat_row
s, feat_cols, &alpha, dC, feat_cols, dH, feat_cols, &beta, dpHT, feat_cols), "Failed to call matrix mult"
);
  CUDA_CALL(cudaDeviceSynchronize(), "Failed to synchronize after kernel call!");

  float* dt;
  size_t szt = feat_rows * label_cols * sizeof(float);
```

```
    CUDA_CALL(cudaMalloc(&dt, szt), "Failed to allocate t!");
    CUDA_CALL(cudaMemcpy(dt, t, szt, cudaMemcpyHostToDevice), "Failed to copy to dt!");

    float* dW;
    size_t szW = feat_rows * label_cols * sizeof(float);
    CUDA_CALL(cudaMalloc(&dW, szW), "Failed to allocate W!");

    // Multiplication of (HTH)^-1*HT and t (labels)
    CUBLAS_CALL(cublasSgemm(cu_cublasHandle, CUBLAS_OP_N, CUBLAS_OP_T, label_cols, feat_col
s, feat_rows, &alpha, dt, label_cols, dpHT, feat_cols, &beta, dW, label_cols), "Failed to call matrix m
ult");
    CUDA_CALL(cudaDeviceSynchronize(), "Failed to synchronize after kernel call!");

    float* res = (float*) malloc(szW);
    CUDA_CALL(cudaMemcpy(res, dW, szW, cudaMemcpyDeviceToHost), "Failed to copy to res!");

    CUBLAS_CALL(cublasDestroy(cu_cublasHandle), "Failed to destroy cuBLAS!");
    return res;
}

int main(int argc, char *argv[])
{
    // number of training examples, features size, number of classes
    int feat_rows = 50000;
    int feat_cols = 2048;
    int label_cols = 10;

    float* H = (float*) malloc(feat_rows * feat_cols * sizeof(float));
    float* t = (float*) malloc(feat_rows * label_cols * sizeof(float));

    // load the features from file.
    string line;
    ifstream featfile("features.txt");
    if(featfile.is_open())
    {
        int row = 0;
        char delim='\n';
        while (getline(featfile, line, delim)) {
            string subs;
            int col = 0;
            string delimiter = " ";
            size_t pos = 0;
            string token;
            while ((pos = line.find(delimiter)) != string::npos) {
                token = line.substr(0, pos);
                H[row*feat_cols + col] = stof(token);
                line.erase(0, pos + delimiter.length());
                col++;
```

```
      }
      row++;
    }
    featfile.close();
  }

  // load the labels from file
  ifstream labelfile("labels.txt");
  if(labelfile.is_open())
  {
    int row = 0;
    char delim='\n';
    while (getline(labelfile, line, delim)) {
      string subs;
      int col = 0;
      string delimiter = " ";
      size_t pos = 0;
      string token;
      while ((pos = line.find(delimiter)) != string::npos) {
        token = line.substr(0, pos);
        t[row*label_cols + col] = stof(token);
        line.erase(0, pos + delimiter.length());
        col++;
      }
      row++;
    }
    labelfile.close();
  }

  // we run the ELM and get the weights
  float* res = d_ELM(H, t, feat_rows, feat_cols, label_cols);

  // we store the weights obtained from ELM into a file
  ofstream output;
  output.open ("weights.txt");
  for (int i=0; i<feat_cols; i++){
    for (int j=0; j<label_cols; j++){
      output << res[j+label_cols*i];
      output << " ";
    }
    output << endl;
  }
  output.close();
  return 0;
}

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

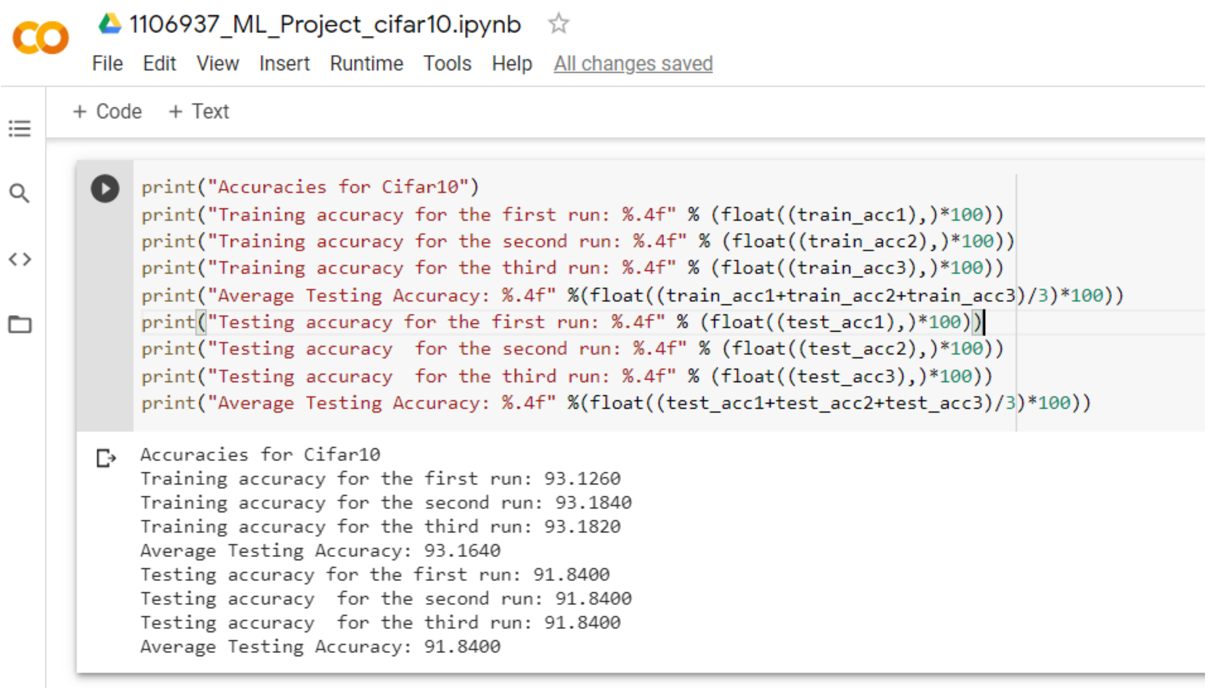
```
weights = np.zeros((2048, NUM_CLASSES), dtype=np.float)
with open('weights.txt') as f:
    for i, line in enumerate(f.readlines()):
        for j, w in enumerate(line.split()):
            weights[i,j] = float(w)
# training accuracy metric
train_acc_metric1 = tf.keras.metrics.SparseCategoricalAccuracy()
train_acc_metric2 = tf.keras.metrics.SparseCategoricalAccuracy()
train_acc_metric3 = tf.keras.metrics.SparseCategoricalAccuracy()

# testing accuracy metric
test_acc_metric1 = tf.keras.metrics.SparseCategoricalAccuracy()
test_acc_metric2 = tf.keras.metrics.SparseCategoricalAccuracy()
test_acc_metric3 = tf.keras.metrics.SparseCategoricalAccuracy()

model.head.set_weights([weights])
# first run
for x_batch_train, y_batch_train in trainSet_processFlow:
    _, train_logits1 = model(x_batch_train, training=False)
    train_acc_metric1.update_state(y_batch_train, train_logits1)
train_acc1 = train_acc_metric1.result()
train_acc_metric1.reset_states()
print("Training accuracy: %.4f" % (float(train_acc1),))
# second run
for x_batch_train, y_batch_train in trainSet_processFlow:
    _, train_logits2 = model(x_batch_train, training=False)
    train_acc_metric2.update_state(y_batch_train, train_logits2)
train_acc2 = train_acc_metric2.result()
train_acc_metric2.reset_states()
print("Training accuracy: %.4f" % (float(train_acc2),))
# third run
for x_batch_train, y_batch_train in trainSet_processFlow:
    _, train_logits3 = model(x_batch_train, training=False)
    train_acc_metric3.update_state(y_batch_train, train_logits3)
train_acc3 = train_acc_metric3.result()
train_acc_metric3.reset_states()
print("Training accuracy: %.4f" % (float(train_acc3),))
# Average of the training accuracies
print("Average Testing Accuracy: %.4f" %(float((train_acc1+train_acc2+train_acc3)/3)*100))
# test run 1
for x_batch_test, y_batch_test in testSet_processFlow:
    _, test_logits1 = model(x_batch_test, training=False)
    test_acc_metric1.update_state(y_batch_test, test_logits1)
test_acc1 = test_acc_metric1.result()
test_acc_metric1.reset_states()
print("Testing accuracy for the first run: %.4f" % (float(test_acc1),))
# test run 2
for x_batch_test, y_batch_test in testSet_processFlow:
```

```
    _, test_logits2 = model(x_batch_test, training=False)
    test_acc_metric2.update_state(y_batch_test, test_logits2)
test_acc2 = test_acc_metric2.result()
test_acc_metric2.reset_states()
print("Testing accuracy  for the second run: %.4f" % (float(test_acc2),))
# test run 3
for x_batch_test, y_batch_test in testSet_processFlow:
    _, test_logits3 = model(x_batch_test, training=False)
    test_acc_metric3.update_state(y_batch_test, test_logits3)
test_acc3 = test_acc_metric3.result()
test_acc_metric3.reset_states()
print("Testing accuracy  for the third run: %.4f" % (float(test_acc3),))
# Average of the testing accuracies
print("Average Testing Accuracy: %.4f" %(float((test_acc1+test_acc2+test_acc3)/3)*100))
print("Accuracies for Cifar10")
print("Training accuracy for the first run: %.4f" % (float((train_acc1),)*100))
print("Training accuracy for the second run: %.4f" % (float((train_acc2),)*100))
print("Training accuracy for the third run: %.4f" % (float((train_acc3),)*100))
print("Average Testing Accuracy: %.4f" %(float((train_acc1+train_acc2+train_acc3)/3)*100))
print("Testing accuracy for the first run: %.4f" % (float((test_acc1),)*100))
print("Testing accuracy  for the second run: %.4f" % (float((test_acc2),)*100))
print("Testing accuracy  for the third run: %.4f" % (float((test_acc3),)*100))
print("Average Testing Accuracy: %.4f" %(float((test_acc1+test_acc2+test_acc3)/3)*100))
```

Output Screenshot



Cifar100:

```
# Importing necessary libraries
import tensorflow as tf
import tensorflow_hub as hub
import tensorflow_datasets as tfds
import numpy as np
# setting the constant values used in the rest of the code.
NUM_CLASSES = 100
BATCH_SIZE = 1000
DATASET_NUM_TRAIN_EXAMPLES = 50000
RESIZE_TO = 128
# Using the state-of-the-art model - Big Transfer (BiT) for acheiving high accuracy.
class FeatureNet(tf.keras.Model):
  def __init__(self, num_classes, module):
    super().__init__()

    self.num_classes = num_classes
    self.head = tf.keras.layers.Dense(num_classes, kernel_initializer='zeros', use_bias=False)
    self.bit_model = module
    self.bit_model.trainable = False #freeze the backbone network

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

# We use the state-of-the-art model for transfer learning from Google: Big Transfer
model_url = "https://tfhub.dev/google/bit/m-r50x1/1"
module = hub.KerasLayer(model_url, trainable=False)
model = FeatureNet(num_classes=NUM_CLASSES, module=module)
# loading the train and test set from cifar100 dataset
(ds_train, ds_test), ds_info = tfds.load(
    'cifar100',
    split=['train', 'test'],
    shuffle_files=True,
    as_supervised=True,
    with_info=True,
)

# printing information about the cifar100 dataset
ds_info
# using dataframe functionality to understand the dataset
tfds.as_dataframe(ds_train.take(5), ds_info)
def cast_to_tuple(features):
  return (features['image'], features['label'])

def trainSet_preprocess(features, label):
  features = tf.image.random_flip_left_right(features)
  features = tf.image.resize(features, [RESIZE_TO, RESIZE_TO])
```

```
  features = tf.cast(features, tf.float32) / 255.0
  return features, label

def testSet_preprocess(features, label):
  features = tf.image.resize(features, [RESIZE_TO, RESIZE_TO])
  features = tf.cast(features, tf.float32) / 255.0
  return features, label

trainSet_processFlow = (ds_train
                 .shuffle(DATASET_NUM_TRAIN_EXAMPLES)
                 .map(trainSet_preprocess, num_parallel_calls=4)
                 .batch(BATCH_SIZE)
                 .prefetch(2))

testSet_processFlow = (ds_test.map(testSet_preprocess, num_parallel_calls=1)
                 .batch(BATCH_SIZE)
                 .prefetch(2))
!rm -f features.txt labels.txt
f_f = open('features.txt', 'a')
l_f = open('labels.txt', 'a')

for step, (x_batch_train, y_batch_train) in enumerate(trainSet_processFlow):
    # extracting features
    H, _ = model(x_batch_train, training=False)
    argstr_feats = ''
    argstr_labels = ''
    t = tf.one_hot(y_batch_train, NUM_CLASSES)
    # Write the features into file
    for row in np.array(H):
        for element in row:
            argstr_feats += (format(element, '.12f') + " ")
        argstr_feats += '\n'
    f_f.write(argstr_feats)
    # Write the labels into file
    for row in np.array(t):
        for element in row:
            argstr_labels += (str(element) + " ")
        argstr_labels += '\n'
    l_f.write(argstr_labels)

f_f.close()
l_f.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;

#define CUDA_CALL(res, str) { if (res != cudaSuccess) { printf("CUDA Error : %s : %s %d : ERR %s\n", st
r, __FILE__, __LINE__, cudaGetErrorName(res)); } }
#define CUBLAS_CALL(res, str) { if (res != CUBLAS_STATUS_SUCCESS) { printf("CUBLAS Error : %s : %s
%d : ERR %d\n", str, __FILE__, __LINE__, int(res)); } }


float* d_ELM(float* H, float* t, int feat_rows, int feat_cols, int label_cols)
{
   cublasHandle_t cu_cublasHandle;
   CUBLAS_CALL(cublasCreate(&cu_cublasHandle), "Failed to initialize cuBLAS!");

   // creating multistreams
   cudaStream_t *streams = (cudaStream_t *) malloc(2*sizeof(cudaStream_t));
   cudaStreamCreate(&streams[0]);
   cudaStreamCreate(&streams[1]);

   size_t szH = feat_rows * feat_cols * sizeof(float);
   size_t szHTH = feat_cols * feat_cols * sizeof(float);
   size_t szH1 = feat_rows * feat_cols/2 * sizeof(float);
   size_t szHTH1 = feat_cols * feat_cols/2 * sizeof(float);

   // Start of multiplication to obtain HtH
   float* H1 = (float*) malloc(szH1);
   float* H2 = (float*) malloc(szH1);

   // Split the features data into two matrices
   for(int i = 0; i < feat_rows; i++) {
      for (int j = 0; j < feat_cols; j++) {
         if (j < feat_cols/2) H1[i*feat_cols/2+j] = H[i*feat_cols+j];
         if (j >= feat_cols/2) H2[i*feat_cols/2+j-feat_cols/2] = H[i*feat_cols+j];
      }
   }

   // Allocate the variables for computing HTH
   float* dHTH;
   float* dH;
   float* dH1;
   float* dH2;
   float* dHTH1;
   float* dHTH2;

   CUDA_CALL(cudaMalloc(&dH, szH), "Failed to allocate H!");
   CUDA_CALL(cudaMalloc(&dHTH, szHTH), "Failed to allocate dHTH!");
```

```
    CUDA_CALL(cudaMalloc(&dH1, szH1), "Failed to allocate dH1");
    CUDA_CALL(cudaMalloc(&dH2, szH1), "Failed to allocate dH2");
    CUDA_CALL(cudaMalloc(&dHTH1, szHTH1), "Failed to allocate dHTH1");
    CUDA_CALL(cudaMalloc(&dHTH2, szHTH1), "Failed to allocate dHTH2");

    CUDA_CALL(cudaMemcpy(dH, H, szH, cudaMemcpyHostToDevice), "Failed to copy to dH!");
    CUDA_CALL(cudaMemcpy(dH1, H1, szH1, cudaMemcpyHostToDevice), "Failed to copy to dH1!");
    CUDA_CALL(cudaMemcpy(dH2, H2, szH1, cudaMemcpyHostToDevice), "Failed to copy to dH2!");

    float alpha = 1.0;
    float beta = 0.0;

    // Multiplication with two CUDA streams. We split the H matrix into two halves.

    cublasSetStream(cu_cublasHandle, streams[0]);
    CUBLAS_CALL(cublasSgemm(cu_cublasHandle, CUBLAS_OP_N, CUBLAS_OP_T, feat_cols, feat_cols
/2, feat_rows, &alpha, dH, feat_cols, dH1, feat_cols/2, &beta, dHTH1, feat_cols), "Failed to call matri
x mult");
    CUDA_CALL(cudaDeviceSynchronize(), "Failed to synchronize after kernel call!");
    cublasSetStream(cu_cublasHandle, streams[1]);
    CUBLAS_CALL(cublasSgemm(cu_cublasHandle, CUBLAS_OP_N, CUBLAS_OP_T, feat_cols, feat_cols
/2, feat_rows, &alpha, dH, feat_cols, dH2, feat_cols/2, &beta, dHTH2, feat_cols), "Failed to call matri
x mult");
    CUDA_CALL(cudaDeviceSynchronize(), "Failed to synchronize after kernel call!");

    float* HTH1 = (float*) malloc(szHTH1);
    float* HTH2 = (float*) malloc(szHTH1);
    CUDA_CALL(cudaMemcpy(HTH1, dHTH1, szHTH1, cudaMemcpyDeviceToHost), "Failed to copy to r
es!");
    CUDA_CALL(cudaMemcpy(HTH2, dHTH2, szHTH1, cudaMemcpyDeviceToHost), "Failed to copy to r
es!");
    float* HTH = (float*) malloc(szHTH);
    for(int i = 0; i < feat_cols/2; i++) {
        for (int j = 0; j < feat_cols; j++) {
            HTH[i*feat_cols + j] = HTH1[i*feat_cols + j];
            HTH[(i+feat_cols/2)*feat_cols + j] = HTH2[i*feat_cols + j];
     }
    }
    for(int i = 0; i < feat_cols; i++) {
        for (int j = 0; j < feat_cols; j++) {
            if (i == j) HTH[i*feat_cols + j] +=1;
     }
    }
    CUDA_CALL(cudaMemcpy(dHTH, HTH, szHTH, cudaMemcpyHostToDevice), "Failed to copy to HTH
!");
    // End of multiplication to obtain HTH

    // Matrix Inversion using LU decomposition
```

```
float** adL;
float** adC;
float* dC;
int* dLUPivots;
int* dLUInfo;

size_t szA = feat_cols * feat_cols * sizeof(float);

CUDA_CALL(cudaMalloc(&adL, sizeof(float*)), "Failed to allocate adL!");
CUDA_CALL(cudaMalloc(&adC, sizeof(float*)), "Failed to allocate adC!");
CUDA_CALL(cudaMalloc(&dC, szA), "Failed to allocate dC!");
CUDA_CALL(cudaMalloc(&dLUPivots, feat_cols * sizeof(float)), "Failed to allocate dLUPivots!");
CUDA_CALL(cudaMalloc(&dLUInfo, sizeof(float)), "Failed to allocate dLUInfo!");

CUDA_CALL(cudaMemcpy(adL, &dHTH, sizeof(float*), cudaMemcpyHostToDevice), "Failed to copy
to adL!");
CUDA_CALL(cudaMemcpy(adC, &dC, sizeof(float*), cudaMemcpyHostToDevice), "Failed to copy to
adC!");

// We call the CUBLAS LU decomposition
CUBLAS_CALL(cublasSgetrfBatched(cu_cublasHandle, feat_cols, adL, feat_cols, dLUPivots, dLUInfo,
1), "Failed to perform LU decomp operation!");
CUDA_CALL(cudaDeviceSynchronize(), "Failed to synchronize after kernel call!");

CUBLAS_CALL(cublasSgetriBatched(cu_cublasHandle, feat_cols, (const float **)adL, feat_cols, dLU
Pivots, adC, feat_cols, dLUInfo, 1), "Failed to perform Inverse operation!");
CUDA_CALL(cudaDeviceSynchronize(), "Failed to synchronize after kernel call!");

CUDA_CALL(cudaFree(adL), "Failed to free adL!");
CUDA_CALL(cudaFree(adC), "Failed to free adC!");
CUDA_CALL(cudaFree(dLUPivots), "Failed to free dLUPivots!");
CUDA_CALL(cudaFree(dLUInfo), "Failed to free dLUInfo!");

// Multiplication of (HTH)^-1 and HT
float* dpHT;
size_t szpHT = feat_rows * feat_cols * sizeof(float);
CUDA_CALL(cudaMalloc(&dpHT, szpHT), "Failed to allocate pHT!");

CUBLAS_CALL(cublasSgemm(cu_cublasHandle, CUBLAS_OP_N, CUBLAS_OP_N, feat_cols, feat_row
s, feat_cols, &alpha, dC, feat_cols, dH, feat_cols, &beta, dpHT, feat_cols), "Failed to call matrix mult"
);
CUDA_CALL(cudaDeviceSynchronize(), "Failed to synchronize after kernel call!");

float* dt;
size_t szt = feat_rows * label_cols * sizeof(float);
CUDA_CALL(cudaMalloc(&dt, szt), "Failed to allocate t!");
CUDA_CALL(cudaMemcpy(dt, t, szt, cudaMemcpyHostToDevice), "Failed to copy to dt!");
```

```
    float* dW;
    size_t szW = feat_rows * label_cols * sizeof(float);
    CUDA_CALL(cudaMalloc(&dW, szW), "Failed to allocate W!");

    // Multiplication of (HTH)^-1*HT and t (labels)
    CUBLAS_CALL(cublasSgemm(cu_cublasHandle, CUBLAS_OP_N, CUBLAS_OP_T, label_cols, feat_col
s, feat_rows, &alpha, dt, label_cols, dpHT, feat_cols, &beta, dW, label_cols), "Failed to call matrix m
ult");
    CUDA_CALL(cudaDeviceSynchronize(), "Failed to synchronize after kernel call!");

    float* res = (float*) malloc(szW);
    CUDA_CALL(cudaMemcpy(res, dW, szW, cudaMemcpyDeviceToHost), "Failed to copy to res!");

    CUBLAS_CALL(cublasDestroy(cu_cublasHandle), "Failed to destroy cuBLAS!");
    return res;
}

int main(int argc, char *argv[])
{
    // number of training examples, features size, number of classes
    int feat_rows = 50000;
    int feat_cols = 2048;
    int label_cols = 100;

    float* H = (float*) malloc(feat_rows * feat_cols * sizeof(float));
    float* t = (float*) malloc(feat_rows * label_cols * sizeof(float));

    // load the features from file.
    string line;
    ifstream featfile("features.txt");
    if(featfile.is_open())
    {
        int row = 0;
        char delim='\n';
        while (getline(featfile, line, delim)) {
            string subs;
            int col = 0;
            string delimiter = " ";
            size_t pos = 0;
            string token;
            while ((pos = line.find(delimiter)) != string::npos) {
                token = line.substr(0, pos);
                H[row*feat_cols + col] = stof(token);
                line.erase(0, pos + delimiter.length());
                col++;
            }
            row++;
```

```
      }
      featfile.close();
    }

    // load the labels from file
    ifstream labelfile("labels.txt");
    if(labelfile.is_open())
    {
      int row = 0;
      char delim='\n';
      while (getline(labelfile, line, delim)) {
        string subs;
        int col = 0;
        string delimiter = " ";
        size_t pos = 0;
        string token;
        while ((pos = line.find(delimiter)) != string::npos) {
          token = line.substr(0, pos);
          t[row*label_cols + col] = stof(token);
          line.erase(0, pos + delimiter.length());
          col++;
        }
        row++;
      }
      labelfile.close();
    }

    // we run the ELM and get the weights
    float* res = d_ELM(H, t, feat_rows, feat_cols, label_cols);

    // we store the weights obtained from ELM into a file
    ofstream output;
    output.open ("weights.txt");
    for (int i=0; i<feat_cols; i++){
      for (int j=0; j<label_cols; j++){
        output << res[j+label_cols*i];
        output << " ";
      }
      output << endl;
    }
    output.close();
    return 0;
  }

!nvcc -o elm ./elm.cu -lcublas
!./elm
weights = np.zeros((2048, NUM_CLASSES), dtype=np.float)
with open('weights.txt') as f:
```

```
    for i, line in enumerate(f.readlines()):
        for j, w in enumerate(line.split()):
            weights[i,j] = float(w)
# training accuracy metric
train_acc_metric1 = tf.keras.metrics.SparseCategoricalAccuracy()
train_acc_metric2 = tf.keras.metrics.SparseCategoricalAccuracy()
train_acc_metric3 = tf.keras.metrics.SparseCategoricalAccuracy()

# testing accuracy metric
test_acc_metric1 = tf.keras.metrics.SparseCategoricalAccuracy()
test_acc_metric2 = tf.keras.metrics.SparseCategoricalAccuracy()
test_acc_metric3 = tf.keras.metrics.SparseCategoricalAccuracy()

model.head.set_weights([weights])
# first run
for x_batch_train, y_batch_train in trainSet_processFlow:
    _, train_logits1 = model(x_batch_train, training=False)
    train_acc_metric1.update_state(y_batch_train, train_logits1)
train_acc1 = train_acc_metric1.result()
train_acc_metric1.reset_states()
print("Training accuracy: %.4f" % (float(train_acc1),))
# second run
for x_batch_train, y_batch_train in trainSet_processFlow:
    _, train_logits2 = model(x_batch_train, training=False)
    train_acc_metric2.update_state(y_batch_train, train_logits2)
train_acc2 = train_acc_metric2.result()
train_acc_metric2.reset_states()
print("Training accuracy: %.4f" % (float(train_acc2),))
# third run
for x_batch_train, y_batch_train in trainSet_processFlow:
    _, train_logits3 = model(x_batch_train, training=False)
    train_acc_metric3.update_state(y_batch_train, train_logits3)
train_acc3 = train_acc_metric3.result()
train_acc_metric3.reset_states()
print("Training accuracy: %.4f" % (float(train_acc3),))
# Average of the training accuracies
print("Average Testing Accuracy: %.4f" %(float((train_acc1+train_acc2+train_acc3)/3)*100))
# test run 1
for x_batch_test, y_batch_test in testSet_processFlow:
    _, test_logits1 = model(x_batch_test, training=False)
    test_acc_metric1.update_state(y_batch_test, test_logits1)
test_acc1 = test_acc_metric1.result()
test_acc_metric1.reset_states()
print("Testing accuracy for the first run: %.4f" % (float(test_acc1),))
# test run 2
for x_batch_test, y_batch_test in testSet_processFlow:
    _, test_logits2 = model(x_batch_test, training=False)
    test_acc_metric2.update_state(y_batch_test, test_logits2)
```

```
test_acc2 = test_acc_metric2.result()
test_acc_metric2.reset_states()
print("Testing accuracy  for the second run: %.4f" % (float(test_acc2),))
# test run 3
for x_batch_test, y_batch_test in testSet_processFlow:
    _, test_logits3 = model(x_batch_test, training=False)
    test_acc_metric3.update_state(y_batch_test, test_logits3)
test_acc3 = test_acc_metric3.result()
test_acc_metric3.reset_states()
print("Testing accuracy  for the third run: %.4f" % (float(test_acc3),))
# Average of the testing accuracies
print("Average Testing Accuracy: %.4f" %(float((test_acc1+test_acc2+test_acc3)/3)*100))
print("Accuracies for Cifar100")
print("Training accuracy for the first run: %.4f" % (float((train_acc1),)*100))
print("Training accuracy for the second run: %.4f" % (float((train_acc2),)*100))
print("Training accuracy for the third run: %.4f" % (float((train_acc3),)*100))
print("Average Testing Accuracy: %.4f" %(float((train_acc1+train_acc2+train_acc3)/3)*100))
print("Testing accuracy for the first run: %.4f" % (float((test_acc1),)*100))
print("Testing accuracy  for the second run: %.4f" % (float((test_acc2),)*100))
print("Testing accuracy  for the third run: %.4f" % (float((test_acc3),)*100))
print("Average Testing Accuracy: %.4f" %(float((test_acc1+test_acc2+test_acc3)/3)*100))
```

Output Screenshot: