Assignment 1

The goal of this assignment is to implement a basic deep learning framework, miniTorch, which is capable of performing operations on tensors with automatic differentiation and necessary operators. You will use this framework to construct a simple feedforward neural network for a sentiment classification task. miniTorch was originally developed by Sasha Rush for education purpose. We extend the original framework to support real cuda kernels and many other accelerations. In this assignment, students will implement the automatic differentiation framework, simple neural network architecture, training and evaluation algorithms in Python and implement the low level operators in C++ and CUDA.

Environment Setup

Please check your version of Python (Python 3.8+), run either:

```
>>> python --version
>>> python3 --version
```

We also highly recommand setting up a virtual environment. The virtual environment lets you install packages that are only used for your assignments and do not impact the rest of the system. We suggest venv or anaconda.

For example, if you choose venv, run the following command:

```
>>> python -m venv venv
>>> source venv/bin/activate
```

If you choose anaconda, run the following command:

```
>>> conda create -n minitorch python=3.9
>>> conda activate minitorch
```

Then clone the starter codes from the git repo and install packages.

```
>>> git clone https://github.com/llmsystem/llmsys_s24_hw1.git
>>> cd llmsys_s24_hw1
>>> python -m pip install -r requirements.txt
>>> python -m pip install -r requirements.extra.txt
>>> python -m pip install -Ue .
```

Compile the CUDA file. Create a directory `cuda_kernels` first, and compile the file.

```
>>> mkdir minitorch/cuda_kernels
>>> nvcc -o minitorch/cuda_kernels/combine.so --shared src/combine.cu
 ↪  -Xcompiler -fPIC
```

Make sure that everything is installed by running python and then checking:

```
>>> import minitorch
```

Problem 1 (20)

Implement automatic differentiation. We have provided the derivative operations for internal Python operators in `minitorch.Function.backward` call. Your task is to write the two core functions needed for automatic differentiation: `topological_sort` and `backpropagate`. This will allow us to traverse the computation graph and compute the gradients along the way.

Complete the following functions in `minitorch/autodiff.py`. The places where you need to fill in your code are highlighted with `BEGIN ASSIGN1_1` and `END ASSIGN1_1`

1. Implement the computation for the reversed topological order of the computation graph. Hints:

- Ensure that you visit the computation graph in a post-order depth-first search.
- When the children nodes of the current node are visited, add the current node at the front of the result order list.

```
def topological_sort(variable: Variable) -> Iterable[Variable]:
    """
    Computes the topological order of the computation graph.
    """
    ...
```

2. Implement the backpropagation on the computation graph in order to compute derivatives for the leave nodes.

```
def backpropagate(variable: Variable, deriv: Any) -> None:
    """
    Runs backpropagation on the computation graph in order to
    compute derivatives for the leave nodes.
    """
    ...
```

3. After correctly implementing the functions, you should be able to pass tests marked as `autodiff`.

```
>>> python -m pytest -l -v -k "autodiff"
```

Problem 2 (40)

In the second part, you need to implement operators for the CUDA Backend `CudaKernelOps`. The places where you need to fill in your code are highlighted with `BEGIN ASSIGN1_2` and `END ASSIGN1_2`

1. Implement CUDA kernels for matrix multiplication, map, zip and reduce functions in `src/combine.cu`.

```
__global__ void MatrixMultiplyKernel(scalar_t* out, ...) {
    ...
}

__global__ void mapKernel(scalar_t* out, ...){
    ...
}


__global__ void reduceKernel(scalar_t* out, ...){
    ...
}

__global__ void zipKernel(scalar_t* out, ...){
    ...
}
```

Hints:

- Data layout and strides.

  To represent multidimensional tensor in memory, we use strides format. To represent a 2D matrix in 1D array, we usually use row major representation, A[i, j] = Adata[i * cols + j]. While for strides format, A[i, j] = Adata[i * strides[0] + j * strides[1]]. For example

```
Adata = [1, 2, 3, 4, 5, 6, 7, 8]
A = [[1, 2, 3, 4], [5, 6, 7, 8]]

# To access (1, 2)
# Row major format
rows, cols = 2, 4
A[1][2] == Adata[1 * cols + 2]
# Strides format
strides = (4, 1)
A[1][2] = Adata[1 * strides[0] + 2 * strides[1]]
```

- Parallelization for matrix multiplication

  A simple way to parallel matrix multiplication is to have every element in the output matrix calculated individually in each block, as is shown in figure 1.

Simple parallelization

```
__global__ void mm(float A[N][N], float
B[N][N], float C[N][N]) {
  int idx = threadIdx.x + blockIdx.x *
blockDim.x;
  int row = idx / N;
  int col = idx % N;
  if (row < N && col < N) {
     float sum = 0.0;
     for(int k = 0; k < N; k++) {
       sum += A[row][k] * B[k][col];
    }
    out[row][col] = sum;
}
```

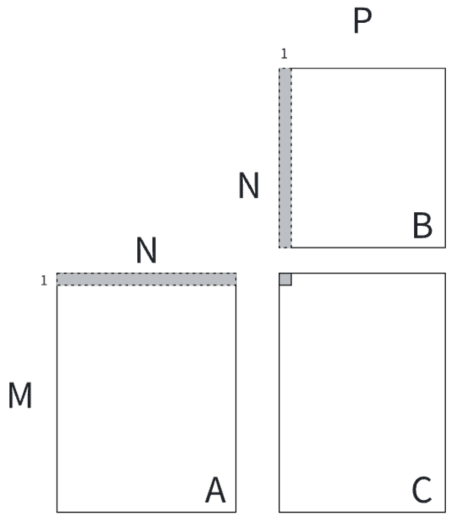

Figure 1: Simple parallelization pseudocode.

A more advanced way to accelerate matrix multiplication with shared memory tiling is illustrated in figure 2. It is resource intensive to only utilize one block of CUDA kernel to calculate one element of the output matrix. We can allocate a chunk of output elements for each block, and create threads inside the block to compute the results in parallel. Each block takes care of a chunk of [S, S] elements. Each thread inside the block calculates smaller parts of $[s_i, \mathrm{L}] \times [\mathrm{L}, s_i]$, and accesses the shared memory across the block.

Shared memory tiling

```
__global__ void mm(float A[N][N], float B[N][N], float C[N][N]) {
  __shared__ float sA[S][L], sB[L][S];
  float tC[S][S] = {0};
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;

  for(int ks = 0; ks < N; ks += L) {
    sA[:, :] = A[i:i+S, ks:ks+L];
    sB[:, :] = B[i:i+S, ks:ks+L];
    __syncthreads();
    for (int ki = 0; ki < L; ++ki) {
        tC[:] += sA[:][ki] * sB[ki][:];
    }
    __syncthreads();
  }
  C[i][j] = tC[:];
}
```

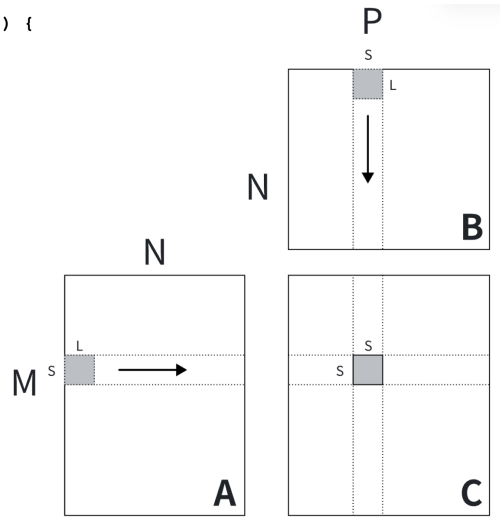

Figure 2: Shared memory tiling pseudocode.

- Parallelization for reduce function

  A simple way to parallel the reduce function is to have every reduced element in the output calculated individually in each block. The basic idea of `ReduceSum` is shown in figure 3. In each block, it is important to think about how to calculate the **step** across the data to be reduced based on `reduce_dim` and `strides`.

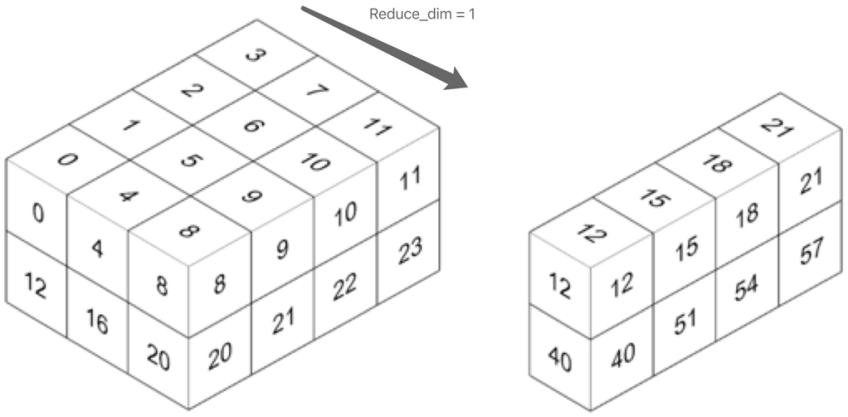

Figure 3: Basic idea of reduce add function.

You can also try optimizing the parallelization for a single reduce operation. Threads inside the block first load the data to a shared memory space, then perform parallel reduction with a tree-based method, as is shown in figure 4. This is a simple optimized

version for ReduceSum[1]. In our implementation, you need to think over how to apply the paradigm to ReduceMultiply and ReduceMax as well. You have to also carefully consider how to apply the reduction over certain axis as we are operating a multidimensional tensor represented as a continguous array. Calculating the positions with helper functions `to_index` and `index_to_position` is necessary.

Parallel Reduction

```
__global__ void reduce0(int *g_idata, int
*g_odata) {
  __shared__ int sdata[];
  int pos = threadIdx.x;
  int i = blockIdx.x*blockDim.x +
threadIdx.x;
  sdata[tid] = g_idata[i];
  __syncthreads();
  for(int s=1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0) {
      sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
  }
  if (tid == 0) {
    g_odata[blockIdx.x] = sdata[0]; }
}
```

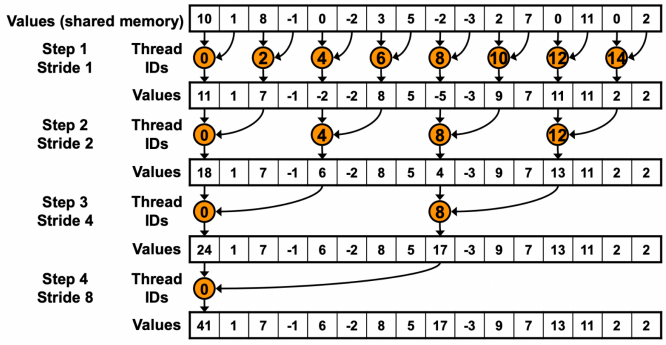

Figure 4: Reduction pseudocode.

2. Recompile the `src/combine.cu` with the following command. **Every time you make changes to the `combine.cu`, you need to compile it again**.

```
>>> nvcc -o minitorch/cuda_kernels/combine.so --shared src/combine.cu
↪  -Xcompiler -fPIC
```

3. Implement the functions in `minitorch/cuda_kernel_ops.py` to load the compiled CUDA functions. An example has been demonstrated in `map` function.

```
class CudaKernelOps(TensorOps):
    @staticmethod
    def zip(fn: Callable[[float, float], float]):
        ...

    @staticmethod
    def reduce(fn: Callable[[float, float], float], start: float = 0.0):
        ...
```

[1]https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

```
    @staticmethod
    def matrix_multiply(a: Tensor, b: Tensor) -> Tensor:
        ...
```

4. After correctly implementing the functions, you should be able to pass tests marked as `cuda`.

```
>>> python -m pytest -l -v -k "cuda"
```

If you want to separately test the four abstraction functions, you can do the followings:

```
>>> python -m pytest -l -v -k "cuda_one"         # for map
>>> python -m pytest -l -v -k "cuda_two"         # for zip
>>> python -m pytest -l -v -k "cuda_reduce"      # for reduce
>>> python -m pytest -l -v -k "cuda_matmul"      # for matrix multiplication
```

Problem 3 (20)

In this section, you will implement the neural network architecture and the training procedure. Complete the following functions in `run_sentiment.py` under the project folder. The places where you need to fill in your code are highlighted with `BEGIN ASSIGN1_3` and `END ASSIGN1_3`.

1. Implement the linear layer with 2D matrix as weights and 1D vector as bias. You need to implement both the initialization function and the forward function for the Linear class. Read the comments carefully before coding.

```
class Linear(minitorch.Module):
    def __init__(self, in_size, out_size):
        ...
    def forward(self, x):
        ...
```

2. Implement the complete neural network used for training. You need to implement both the initialization function and the forward function of the Network class. Read the comments carefully before coding.

```
class Network(minitorch.Module):
    """
    Implement a MLP for SST-2 sentence sentiment classification.

    This model should implement the following procedure:
```

```
    1. Average over the sentence length.
    2. Apply a Linear layer to hidden_dim followed by a ReLU and Dropout.
    3. Apply a Linear to size C (number of classes).
    4. Apply a sigmoid.
    """

    def __init__(
        self,
        embedding_dim=50,
        hidden_dim=32,
        dropout_prob=0.5,
    ):
        ...

    def forward(self, embeddings):
        """
        embeddings tensor: [batch x sentence length x embedding dim]
        """
        ...
```

3. After correctly implementing the functions, you should be able to pass tests marked as `network`.

```
>>> python -m pytest -l -v -k "network"
```

Problem 4 (20)

In this section, you will implement codes for training and perform training on a simple MLP for the sentence sentiment classification task. The places where you need to fill in your code are highlighted with `BEGIN ASSIGN1_4` and `END ASSIGN1_4`.

1. Implement the training loop. You need to complete the code for training and validation. Read the comments carefully before coding. What's more, we **strongly** suggest leveraging the `default_log_fn` function to print the validation accuracy. The outputs will be used for **autograding**.

```
class SentenceSentimentTrain:
    '''
        The trainer class of sentence sentiment classification
    '''
    ...
```

```
    def train(self, data_train, ...):
        ...
```

2. Train the neural network on SST-2 (Stanford Sentiment Treebank) and report your training and validation results.

```
>>> python project/run_sentiment.py
```

You should be able to achieve a best validation accuracy equal to or higher than 75%.

It might take some time to download the GloVe embedding file before the first training. Be patient!

Submission

Please submit the whole `llmsys_s24_hw1` as a zip on canvas. Your code will be automatically compiled and graded with private test cases.