ECE 449/590, Fall 2022
Project 4: Inference with Neural Networks

Due: 11/13 (Sun.), by the end of the day (Chicago time)

1 Summary

The process to make predictions with neural networks is usually known as inference. To perform inference, one need to train the neural network model first to obtain weights and biases. Then, for a given input example, the output vector is computed from the network using the input and the trained weights and biases. For a classification problem, typically the last step is to locate the maximum element from the output vector and to output the index as the class.

To achieve better computational efficiency, both training and inference of neural network models process examples in batch. It is best to think of inputs to a neural network implementaion as a design matrix of multiple rows where each row being an example. For our tensor implementation, this implies that the first dimension `shape[0]` is the number of examples.

When handling images, the inputs are further represented by a tensor with 4 dimensions as each image is represented by an array of 3 dimensions for its height, width, and color channels. Such tensor format is usually known as the NHWC format where N=`shape[0]` stands for the number of examples, H=`shape[1]` and W=`shape[2]` are height and width respectively, and C=`shape[3]` stands for channels. On the other hand, as computation happens within the neural network, it is more efficient to arrange the dimensions of the tensor differently as NCHW, i.e. N=`shape[0]`, C=`shape[1]`, H=`shape[2]`, and W=`shape[3]`. Therefore, when necessary, one need to convert from one tensor format to another.

In addition, with images as 4D tensors, it is helpful to treat a large tensor as slices of sub-tensors, e.g. a 4D tensor of the shape (N, C, H, W) could be treated as slices of tensors with the shape (C, H, W) per example and as slices of tensors with the shape (H, W) per example per channel.

In this project, we will implement the following tensor operations to support inference with multilayer perceptron (MLP) and with convolutional neural network (CNN) for EasyNN in C++.

- `ReLU()`: apply the ReLU operation element-wise to the elements in the input tensor.

- `Flatten()`: form a design matrix from the input tensor by flattening each example into a vector using row-major order. For example, if the input tensor is in the NCHW format, the output tensor will has two dimensions where `shape[0]`=N will remain unchanged and `shape[1]`=C*H*W.

- `Input2d(name, height, width, in_channels)`: obtain the input tensor in NHWC format using `name` and output it in NCHW format. The parameters `height`, `width`, and `in_channels` allow to verify the shapes of the input tensor, though you don't have to check for them.

- `Linear(weight, bias)`: the input tensor is a matrix of N rows and I columns, `weight` is a matrix with O rows and I columns, `bias` is a vector of O elements, and the output tensor is a matrix of N rows and O columns. For each row x of the input tensor, compute a row in the output tensor as $(\text{weight } x^\top + \text{bias})^\top$.

- `MaxPool2d(kernel_size, stride)`: the input tensor is a 4D tensor with the shape (N, C, H, W). The parameters `kernel_size` and `stride` are both integers (you will need to extract them from the parameter tensors and convert them to integers). For simplicity, assume `kernel_size` = `stride`. For each `kernel_size` × `kernel_size` patch from one slice per example per channel from the input tensor, their maximum is put into the output tensor. Since we assume `kernel_size` = `stride`, patches won't overlap and partial patches at the boundary will be discarded. Overall, the output should be a 4D tensor with the shape (N, C, H/`kernel_size`, W/`kernel_size`).

- `Conv2d(in_channels, out_channels, kernel_size, weight, bias)`: the input tensor is a 4D tensor with the shape (N, `in_channels`, H, W), `weight` is a 4D tensor with the shape (`out_channels`, `in_channels`, `kernel_size`, `kernel_size`), `bias` is a vector with `out_channels` elements. (Note that the parameters `in_channels`, `out_channels`, and `kernel_size` allow to verify the shapes of the input tensor, `weight`, and `bias`, though you don't have to check for them.) For each `in_channels` × `kernel_size` × `kernel_size` tensor from one slice per example from the input tensor, it is multiplied element-wise with one slice per `out_channels` from `weight`, and then the result elements, plus the corresponding element from `bias` for `out_channels`, are added together and put into the output tensor. Overall, the output should be a 4D tensor with the shape (N, `out_channels`, H-`kernel_size`+1, W-`kernel_size`+1).

This project should be done individually. Discussions are encouraged. However, all the programs (except those from the lectures) and writings should be by yourself. COPY without proper CITATION will be treated as PLAGIARISM and called for DISCIPLINARY ACTION.

NEVER share your programs/writings with others.

2 Working with Your Projects

Please continue to work with your Git repository for Project 4. Here is a brief introduction of the files.

- `easynn.py`, `easynn_golden.py`, `easynn_cpp.py`, `Makefile`: same as those in Project 1 to 3. You should not modify them.
- `src`: this directory contains all your C++ implementations. Update them as needed.
- `easynn_test.cpp`: continue to use this file to test and to debug your C++ implementations. Start with a test that is as simple as possible and then move to more complicated cases.
- `grade_p4.py`: this is the grading script to verify whether your C++ implementations in `src` are correct or not. There are 10 questions. You should not modify this file.
- `mnist_test.npz`: the test examples from the MNIST dataset, stored as NumPy arrays and used by `grade_p4.py` to verify your implementation.
- `msimple_params.npz`: trained weights and biases for the MLP model used in `grade_p4.py` to verify your implementation.
- `mnist_params.npz`: trained weights and biases for the CNN model used in `grade_p4.py` to verify your implementation.

After creating the shared library using "make", run the grading script to see if all questions pass.

```
make
python3 grade_p4.py
```

3 Deliverables and Grading

We obtain a copy of all your source files in `src` as you push the changes to the central Git repository so there is no need for you to submit them to us using any other mechanisms. Moreover, please be advised that since to learn the use of Git and a CI system is among the objectives of this course, we will NOT accept project submissions outside the central Git repository, e.g. via emails. If you have difficulty accessing the central Git repository, it is your responsibility to act promptly to seek help from us well before the project deadline; otherwise, not able to access the central Git repository is NOT an excuse for late submissions.

Project 4 will have a full grade of 100 points. Each function, if passed, will give you 10 points. Since you are required to use the CI system to troubleshoot any issues with your code before the deadline, a failed function will earn 0 points.

The following submission checklist is provided for your convinience. Detailed instructions are available from Section IV of Guide to System Setup and Work Flow.

- ☐ Run `python3 grade_p4.py` in VM to make sure all 10 tests pass.
- ☐ Commit and push your changes to the central Git repository.
- ☐ Run `/home/ece449/show` on uranus.ece.iit.edu to access your grading report and correct any issues before the deadline.