Lab1: Benchmarking Neural Network

Jiyang Tang
jiyangta@andrew.cmu.edu
Code: https://github.com/tjysdsg/cmu-11767/tree/master/labs/lab1

Dataset

The train and dev set from SST-2. Bag-of-words is used as the feature.

Network Architecture

The network contains 1 input layer which projects the input bag-of-words vectors into hidden layer size. Then the vectors are passed to certain number of identical hidden layers are and finally be converted to a logit via the output projection layer. ReLU is used as the activation function. When performing inference, sigmoid is applied to the result and if the output is considered positive if it is larger than 0.5.

Configurations

The following options are used for all experiments:

- learning rate 0.001
- batch size 64
- epochs 2

Baseline

The baseline model has 2 hidden layers, and its hidden size is 256.

The bag-of-word feature vectors are extracted from the top 10,000 most common words in the training data. We ignore out-of-vocabulary words when processing the dev set.

The accuracy evaluated on the dev set is 0.81.

Other Experiments

Experiments are performed using a combination of all of the following hyper-parameter values:

- number of hidden layers: [1, 2, 4]
- hidden layer size: [128, 256, 512]

- vocabulary size (top k common words): [1000, 5000, 10000]

Benchmarking Metrics

- FLOPs: FLOPs is calculated for each layer following the same rule: Number of operations needed for weight matrix – input vector multiplication plus those needed for bias vector addition. We ignore the FLOPs for ReLU as it only involves comparing values and copying part of the vector. More specifically, for a linear layer whose input size is m and output size is n:

$$\text{FLOPs} = 2mn + n$$

- Number of parameters: this is easily calculated by iterating through all parameters in the PyTorch module and sum the number of elements in the weight and bias matrix/vectors. The result is highly correlated with FLOPs so we do not report them in the later sections for brevity.
- Inference time (ms): For an entire inference run of all examples in the dev set, we monitor the start and end time in nanoseconds. Then we get the average inference time of each example by dividing the value with number of examples. We repeat this process 5 times and calculate the average again to avoid instability. Note that the average inference time obtained from the first trial is discarded to let the cache warm up.
    - We did not choose to measure inference time of each example and then perform average because the Python timer is really inaccurate for time under 1ms, while most examples take much less time than that.
- Training time (seconds): Using a similar strategy, we measure the time it takes to train the model for full 2 epochs and calculate the average across 5 trials. The first trial is discarded. We found a clear positive correlation between training time and inference time, so only the inference time is reported in the results section for brevity.

For training time and inference time measurement, we did not see a high variance or outliers. We believe that the first few iterations of network training or inference is enough for warming up and since we always run multiple trials, the impact of warming up is minimal.

Benchmarking Environment

Both training and inference are performed using CPU under the same conditions. We also stopped as many applications and background services as possible.

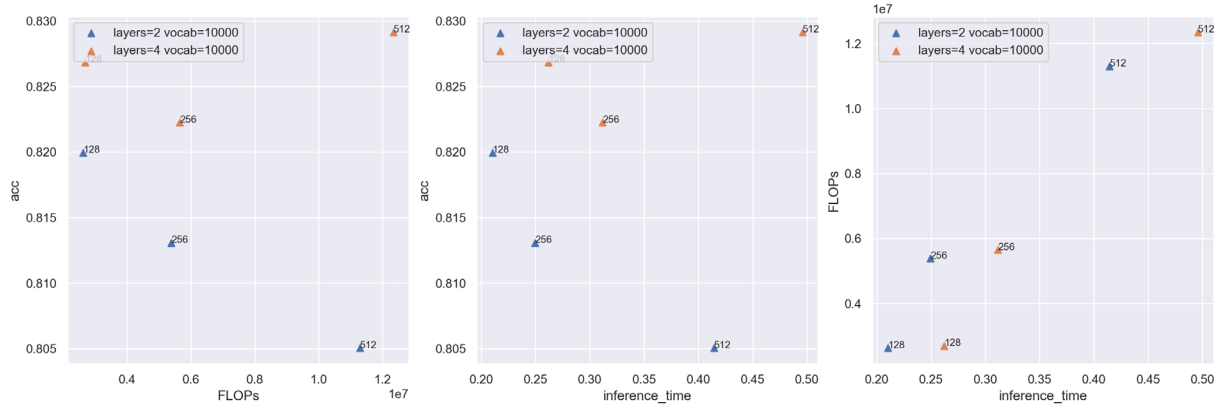
- CPU:

    - 11th Gen Intel i7-11800H @ 2.30GHz
    - Base speed: 2.30 GHz
    - Speed 3.35 GHz
    - Cores: 8
    - Logical processors: 16
    - Virtualization: Enabled
    - L1 cache: 640 KB
    - L2 cache: 10.0 MB
    - L3 cache: 24.0 MB
- Operating System: Windows Pro
- Memory: 32G
- The entire dataset is loaded into memory before training, so disk speed should be irrelevant
- Python:
    - python==3.8.8
    - pytorch==1.12.1

Result and Analysis

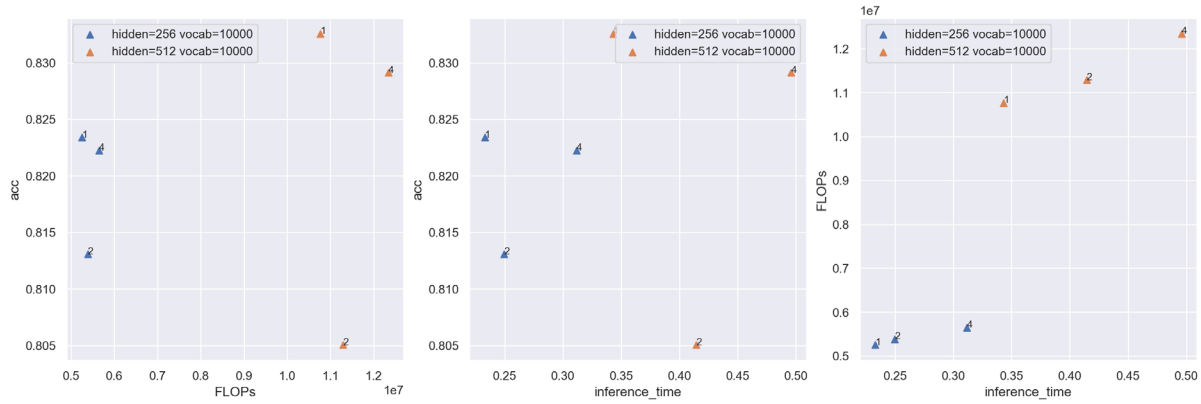
Varying hidden size

While keeping other hyper-parameters the same, we vary the hidden layer size and plot the following graphs:



The general trend is that the larger the hidden layers are, the higher FLOPs and the more inference time it takes to run the model. This is true even if we change the number of layers. In addition, in the first two graphs we see that higher FLOPs or longer inference time doesn't necessarily mean better performance. In later sections, we will discover more with more data points. Besides that, we see that FLOPs and inference time are positively correlated, and this matches our expectation.

Varying number of hidden layers

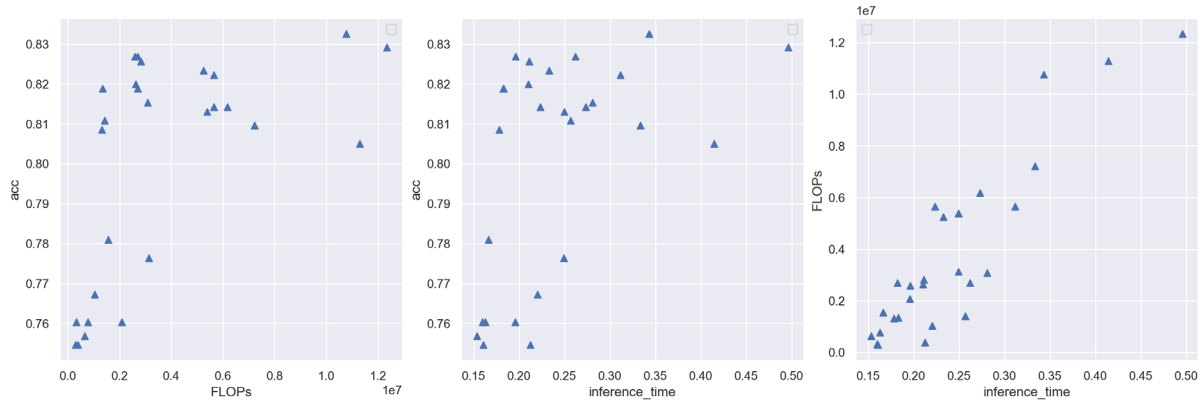


By increasing the number of hidden layers, the model takes more FLOPs and runs slower, similar to what we see in the previous section. And again, the accuracy isn't improved by increasing the number of layers. Comparing the third graph with that of the previous section, we see that doubling the "width" of the network has more impact on FLOPs and on inference time. This implies that having "narrower" but "deeper" network might be more desirable on embedded devices.

Varying input size (vocabulary size)

Unlike previous sections, the input size has a significant impact on the test accuracy. Models with larger input size have much higher accuracies. But this comes at the expense of significantly higher inference time and FLOPs. Other aspects of these three graphs match our findings in the previous two sections.

All experiments



The above three graphs contain all experiments we conducted using a combination of the hyper-parameter values mentioned above. In the first two graphs, we now can see a general positive correlation between test accuracy and FLOPs or inference time. However, there are clearly some outliers. And the accuracy increase stagnates after a certain point. Therefore, increasing the size of the network does increase its classification capability. But the accuracy bottleneck at other places might have a much higher impact passing a certain size, for example, the architecture complexity or feature design. The takeaway is that networks running on resource-constrained environments should always be benchmarked to find the optimal trade-off depending on the use case. In this case, we can almost achieve the highest accuracy with less than 2e6 FLOPs or around 0.2ms per example.