11767 - On Device Machine Learning Lab 1 Assignment

Oyeon (Owen) Kwon

September 13, 2024

Abstract

The task focuses on classifying the MNIST dataset, which will later be used to inform a weighted loss function for future object detection tasks. The dataset consists of 28x28 grayscale images, where each pixel has a value between 0 and 255, representing different shades of gray. Each image is labeled with a digit from 0 to 9, making it a 10-class classification problem. To make independent test dataset, I randomly split the training set into 80% for training and 20% for validation, and used independent testset. For consistency, seeds were fixed.

A feed-forward neural network (also known as a multilayer perceptron, or MLP) is employed. The network includes two hidden layers, each with 1024 units, and ReLU activation functions applied after each layer. The model is trained using cross-entropy loss, which is a common choice for classification tasks, and the optimization process uses the Adam optimizer with a learning rate of 0.001. The training is performed with a batch size of 64, which balances memory usage and convergence speed.

The performance of the model will be evaluated using several key metrics. Latency will be measured during both the training and inference stages to assess the model's speed. Additionally, the number of parameters and FLOPs (floating point operations) will be calculated to gauge the model's computational complexity. Finally, accuracy will be the primary metric to measure the model's ability to correctly classify the digits. These metrics together will provide a comprehensive understanding of the model's efficiency and effectiveness in the classification task.

1 Main Section

1.1 *Subsection 1: What is the accuracy of the model on the train split and dev splits? Report the resulting hyperparameters*

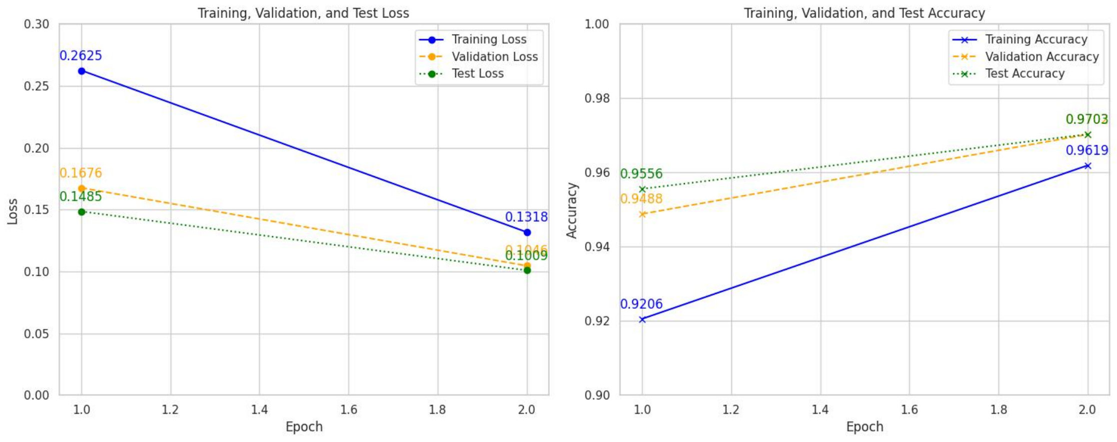

Figure 1: Figure caption

Figure 1: "Test" refers to the independent dataset ("mnist_val.csv"), and "Train" refers to the dataset ("mnist_train.csv"). To create the validation dataset, I randomly split the training set into 80% for training and 20% for validation. For consistency, seeds were fixed. In the figure, the left side shows the training, validation, and test loss trends over two epochs. The sharp decrease in loss suggests that the model training is progressing well, though there are signs of underfitting, indicating the need for more epochs. The right side illustrates accuracy trends, where accuracy increases as loss decreases. Both test and validation accuracy reached 97% by epoch 2.

1.2 *Subsection 2: What is the experimental setting used to benchmark the model? Report the software (e.g. Python and PyTorch Version, Operating System, etc.) and hardware configurations (e.g. CPU, RAM, etc.) in as much detail as is available from your operating system.*

Software Configuration

- **Python Version**: 3.10.14 (packaged by conda-forge, GCC 12.3.0).
- **PyTorch Version**: 2.2.0 with CUDA 12.1 support.
- **Operating System**: Linux 5.10 (Debian 5.10.223-1).

Hardware Configuration

- **CPU**: Utilizes AVX512 instructions with multiple cores, showing significant idle time (2121025.06 seconds).
- **RAM**: 14.65 GB total, 11.22 GB available.
- **GPU**: Tesla T4, supported by CUDA 12.1.

PyTorch Build Details

- **Compiler**: GCC 9.3, built with C++17 standards.
- **Math Libraries**: Intel MKL, MKL-DNN, OpenMP, and LAPACK for optimized performance.
- **CUDA & CuDNN**: CUDA 12.1, CuDNN 8.9.2, supporting architectures up to compute 90 (sm_90).

Optimizations

- **Optimizations**: NNPACK, XNNPACK, and FBGEMM enabled for fast neural network operations. AVX512, AVX2, and AVX optimizations enabled for efficient CPU processing.

1.3 Subsection 3: Report the average training time per epoch and inference latency per example of your model.

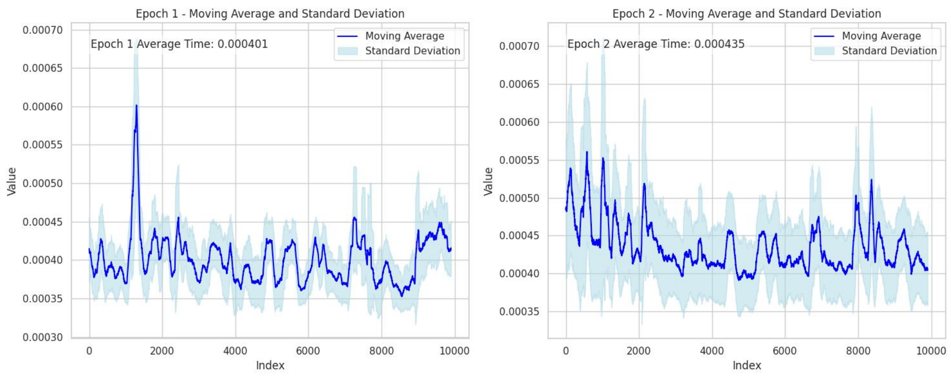

Figure 2: The X-axis indicates the number of test samples in sequential order, while the Y-axis represents the inference time per sample on a CPU. The figure illustrates key trends and variability in the dataset by applying a moving average and standard deviation. The moving average is calculated over a fixed window size of 100, smoothing the data to highlight overall trends. Additionally, the standard deviation is computed within each window to reflect the variability in the data. The shaded area around the moving average visually represents the range within which the data fluctuates. Between Epoch 1 and Epoch 2, both the latency and the degree of variance appear to be significantly higher compared to Epoch 1.

1.3.1 Explain any phenomena you note

I believe the trend in inference time is quite similar. On average, it is around 0.0004 ms for Epoch 1 and 0.00435 for Epoch 2 which seems the average latency increased little bit. This change is not arbitrary; Repetitive inference increases computational demands, leading to greater latency and making the system slightly unstable.

Additionally, since the model is already loaded into memory from the training step, the initial indices do not show outliers. After initializing all variables and starting the inference step, the first inference time appears significantly longer, likely due to the model being loaded into memory.

1.4 Subsection 4: Report the parameter count of the model as manually calculated; and as determined by your programatic code solution. Does this align with your expectations? Why or why not?

Model Parameter Calculation

- **Input Layer**: A fully connected layer that connects the input size (`input_size`) to the hidden size (`hidden_size`).
- **Hidden Layers**: Multiple hidden layers, each connecting the previous hidden layer to the same width (`hidden_layer_width`), defaulting to `hidden_size`.
- **Output Layer**: A fully connected layer that connects the last hidden layer to the output size (`output_size`).

Let's assume the following values for the parameters:

- `input_size` $= 784 \quad (28 \times 28 \quad \text{pixels})$
- `hidden_size` $= 1024$
- `output_size` $= 10 \quad (\text{for 10-class classification})$
- `num_hidden_layers` $= 2$

The number of parameters for each layer can be calculated as follows:

- **Input Layer**:

$$\text{Parameters} = \texttt{input\_size} \times \texttt{hidden\_size} + \texttt{hidden\_size} = 784 \times 1024 + 1024 = 803,840$$

- **Hidden Layers**: Each hidden layer has parameters:

$$\text{Parameters} = \texttt{hidden\_size} \times \texttt{hidden\_size} + \texttt{hidden\_size} = 1024 \times 1024 + 1024 = 1,049,600$$

  Since there are two hidden layers, the total for all hidden layers is:

$$1,049,600 + 1,049,600 = 2,099,200$$

- **Output Layer**:

$$\text{Parameters} = \texttt{hidden\_size} \times \texttt{output\_size} + \texttt{output\_size} = 1024 \times 10 + 10 = 10,250$$

Total Parameters (Manual Calculation):

$$803,840 + 2,099,200 + 10,250 = 2,913,290$$

Determined Model Parameters Calculation

```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)
```

Use the Python function above to calculate the total parameters to 2,913,290 parameters.

1.5 *Subsection 5: Report the number of FLOPs that your model requires to perform single-batch inference.*

- **FLOPs**: 2910208

```
from thop import profile

def compute_flops(model):
    input_tensor = torch.randn(1, CONFIG['input_dims'])
    flops, _ = profile(model, inputs=(input_tensor,), verbose=False)
    return flops
```

1.6 *Subsection 6: Train and evaluate your model using a variety of depths. Report the parameter counts, FLOPs, and latency, and generate three plots*

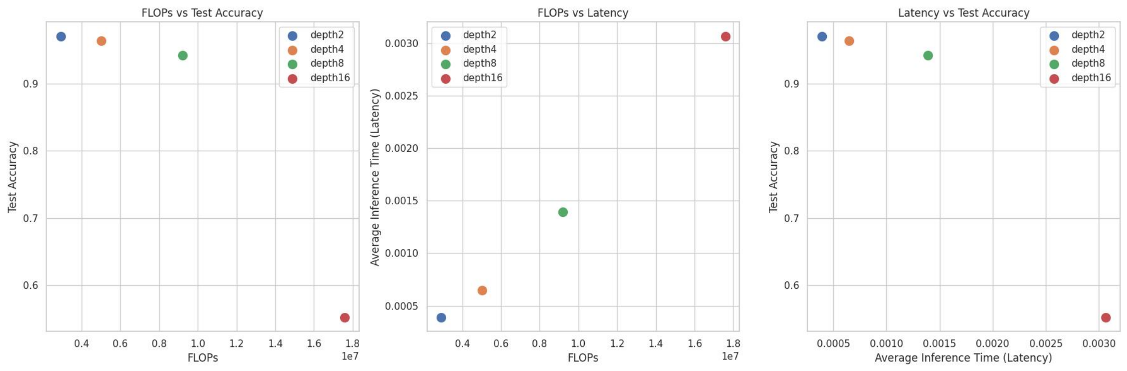

Figure 3: **Left Side**: This figure shows the relationship between FLOPs (x-axis) and accuracy (y-axis). As the model depth increases, the number of FLOPs rises proportionally. Despite adding more layers, the accuracy tends to decrease. Other techniques have not been applied to the depth layers, so the performance appears to be declining. **Middle Side**: This figure illustrates the relationship between FLOPs (x-axis) and latency (y-axis). With increasing model depth, the average latency time (calculated the avreage from the inference time per batch across all individual samples) also rises proportionally. **Right Side**: This figure presents the relationship between latency (x-axis) and accuracy (y-axis). We can observe that the average latency time increases by nearly six times from depth 2 to depth 16.

1.7 Subsection 7: Train and evaluate your model using a variety of widths. Make sure you try both going narrower than the input, and wider than the input, if memory permits. Report the parameter counts, and generate the same three plots as in Question 6, and discuss.

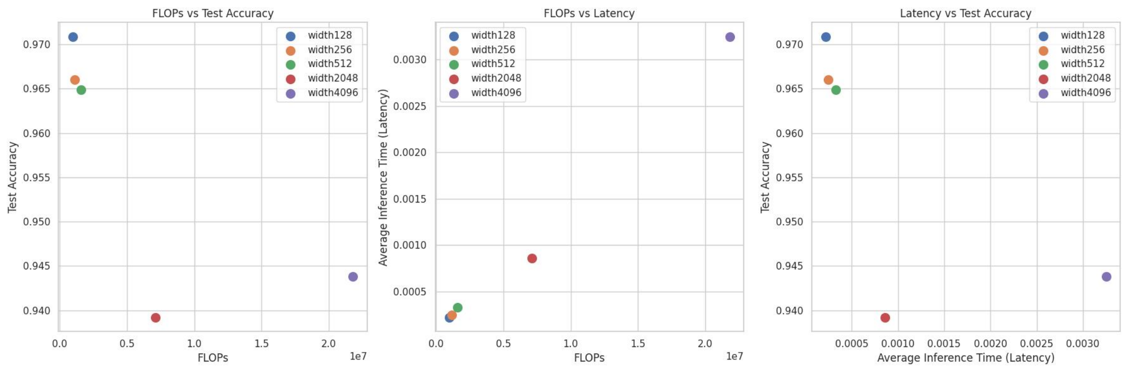

Figure 4: **Left Side**: The left figure illustrates the relationship between FLOPs (x-axis) and test accuracy (y-axis). As model width increases, FLOPs grow proportionally, while accuracy tends to decline when the width is large. **Middle Side**:The middle figure shows the correlation between FLOPs and latency, where latency significantly increases after a certain threshold, with the model at a width of 4096 exhibiting almost six times more latency than other models. **Right Side**: The right figure depicts the trade-off between latency and accuracy, where the shallowest model, without additional hyperparameters like dropout or batch normalization, achieves good performance while maintaining the lowest latency.

1.8 *Subsection 8: Vision: Downsample by resizing the image to a smaller size, and experiment with a few different downsampling rates.*

In this section, total 7 augmentation were applied to observe the trend of data augmentation performance.

- **No Transformation**: Default. Only resizing to 28 × 28 and Z-Normalization is applied.
- **Crop**: Applied with sizes 20, 14, and 7 pixels.
- **Resize**: Applied with sizes 20, 14, and 7 pixels.

Below are examples of the transformations for `crop_7` and `resize_7`:

```
elif transform_type == 'crop_7':
    return transforms.Compose([
        transforms.Lambda(lambda img: torch.tensor(img).unsqueeze(0))
        transforms.Resize((28, 28)),
        transforms.CenterCrop(7),
        transforms.Normalize((0.5,), (0.5,))
    ]), 7 * 7


elif transform_type == 'resize_7':
    return transforms.Compose([
        transforms.Lambda(lambda img: torch.tensor(img).unsqueeze(0))
        transforms.Resize((7, 7)),
        transforms.Normalize((0.5,), (0.5,))
    ]), 7 * 7
```

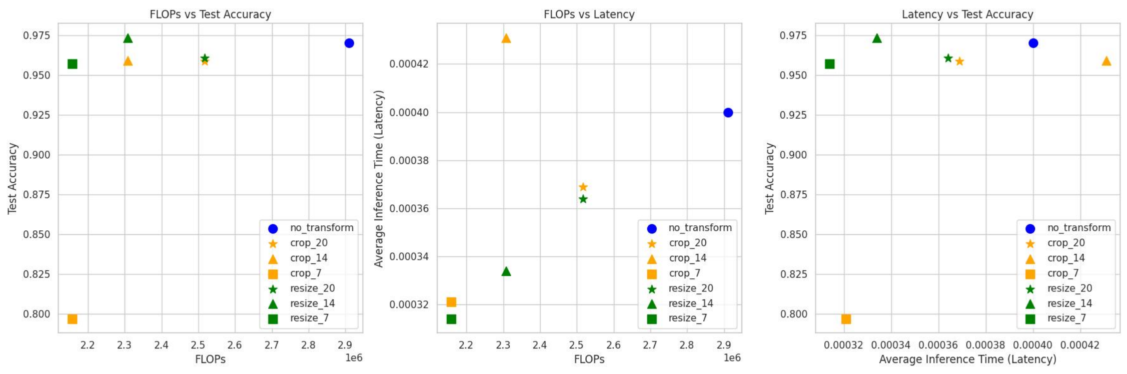

Figure 5: **Left Side**: The blue color represents no transformation, the yellow color indicates crop augmentation, and the green color shows resize augmentation. As the input resolution decreases, FLOPs also decrease. However, resizing generally yields better performance than cropping. This may be due to cropping cutting out important parts of the image, such as occluded objects, which makes it harder for the model to learn meaningful features. In contrast, resizing preserves the entire image, making it a better option in this setup. However, cropping can be valuable when training with a large dataset, as it exposes the model to more diverse situations. Notably, no transformation results in the highest FLOPs while showing similar performance to the other methods. **Middle Side**: Latency decreases as FLOPs decrease, corresponding to smaller image resolutions. Interestingly, cropping incurs more inference time than resizing, despite both having same FLOPs. This discrepancy will require further investigation. **Right Side**: Resizing consistently maintains or slightly improves performance compared to the original image resolution, offering few advantage in terms of latency reduction.

1.9 Subsection 9: Now, experiment with different settings combining different input sizes with different depths and widths. Generate the same three plots as above, reporting the results of your experiments. Explain your experimental process. Discuss the relation and tradeoffs between various efficiency metrics and model performance.

1.9.1 What combination of input resolution, depth, and width seems to correspond to the best trade-off between accuracy and efficiency?

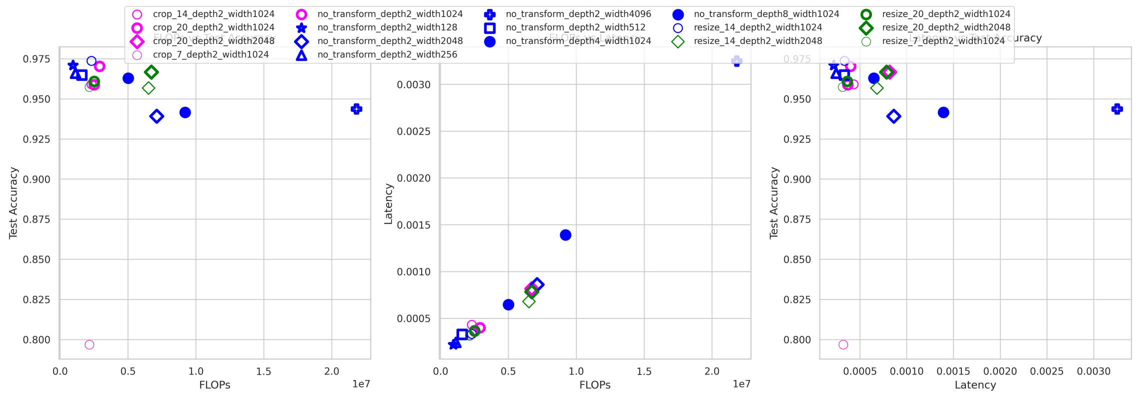

Figure 6: The plot uses distinct markers and line styles to differentiate between transformations and depth&width levels. No transformation is represented by blue, unfilled circles, resize is indicated by green, unfilled circles, and crop is shown by magenta, unfilled circles. For depth levels, depth 4 and depth 8 are displayed with filled circles in their respective colors. The shapes for input sizes (128, 256, 512, 1024, 2048, and 4096) are varied, with each size being represented by different geometric markers such as stars and other unique shapes. The thickness of the lines connecting the points increases proportionally with the input size, illustrating the relationship between input resolution and computational load. As input size grows, the lines become progressively thicker, highlighting the impact of larger inputs on performance metrics.

1.9.2 Do you notice any interesting trends? Does variation of input size or model size have a larger impact on efficiency or performance? Do you notice differences between various efficiency metrics (e.g. latency vs FLOPs)?

The model with no transformation, the smallest width, and the lowest depth yields the smallest FLOPs while surprisingly achieving the best performance. While data augmentation techniques also provide strong performance, they lead to an increase in FLOPs. Careful consideration is required when using wider networks, as they significantly increase FLOPs more than any other factors.

Reducing the width of the network while selecting a specific number of layers can effectively decrease the FLOPs, thereby offering latency advantages without heavily compromising performance.

Interestingly, data augmentation does not significantly add to the latency, making it a valuable tool for building more robust models while maintaining a reasonable inference speed. This allows for an optimal trade-off between performance and computational efficiency.

1.10 Appendix

- All experimental codes are available at: `https://github.com/Oyeon/CMU-ondeviceML/tree/main`
- The experimental environment is provided using `poetry`, ensuring easy reproduction in the same environment.