A DEEP LOOK INTO LENET WITH MNIST: PERFORMANCE ANALYSIS AND OPTIMIZER COMPARISON

Jingqi Fan
20226758
Software Engineering (International Class) 2201
fanjingqi@stumail.neu.edu.cn

ABSTRACT

This report compares the performance of Stochastic Gradient Descent and Adam optimizers on the LeNet model with the MNIST dataset. Through experiments with various learning rates and batch sizes, we find that Adam is more robust, and less sensitive to learning rates than SGD. We also examine the efficiency of different hardware setups, revealing that while one GPU reduces the training time, adding more than one GPU leads to diminishing returns due to communication overhead. The results provide guidance on optimizer choice and hardware allocation for efficient model training.

1 INTRODUCTION

LeNet (LeCun et al., 1998) is one of the earliest convolutional neural networks designed for image classification tasks. It was developed to classify handwritten digits from the MNIST dataset, a widely-used benchmark dataset containing 60,000 training and 10,000 test images of 28x28 grayscale digits. Despite its simplicity, LeNet remains a foundational model in the field of deep learning and is often used to introduce convolutional neural network architectures.

During the training process, the choice of optimization algorithms is crucial. To train LeNet, previous literature mainly uses stochastic gradient descent (Robbins & Monro, 1951) or Adam (Kingma, 2014). Stochastic gradient descent updates model weights using the average gradient from the current batch, while Adam adapts the learning rate for each parameter using the first and second moments of the gradients.

However, key hyperparameters such as learning rate and batch size significantly affect model performance (Glorot & Bengio, 2010). A well-tuned learning rate ensures efficient and stable training, allowing the model to converge quickly. In contrast, an inappropriate learning rate can lead to slow convergence or cause the model to overshoot the optimal solution. Similarly, an appropriately chosen batch size balances training efficiency and generalization, while a poorly selected batch size can result in unstable updates or sub-optimal performance.

1.1 CONTRIBUTION

In this report, we conduct thorough experiments to evaluate and compare the performance of stochastic gradient descent and Adam, when applied to the LeNet model trained on the MNIST dataset. We also experiment with different learning rates and batch sizes to examine their effects on the training process. By analyzing the convergence speed and final accuracy of these optimizers, under various hyperparameter settings, we aim to provide insights into how different optimization strategies and parameter choices affect training efficiency and model performance.

Besides, we analyze the computational efficiency of the training process by comparing the training speed across different hardware setups, including CPU, single GPU, and multi-GPU parallelization. By measuring the time taken for each configuration, we assess the practical benefits of using GPUs and parallel computing for machine learning tasks, highlighting the trade-offs between speed and computational resources.

2 PRELIMINARIES

In this section, we provide a detailed description of our experimental setup and the adjusted code.

2.1 SET UP

For these experiments, we utilize a single NVIDIA RTX 4090 GPU, running CUDA 12.2, along with torch 11.8 and torchvision 11.8. To compare the influence of different hardware configurations, we also run experiments on Intel(R) Xeon(R) Platinum 8336C CPU with 128 cores. This allows us to observe how CPU-only training compares to GPU-accelerated training. Additionally, we evaluate performance using up to four NVIDIA RTX 4090 GPUs in parallel to assess the impact of multi-GPU training on speed and scalability.

2.2 CODE IMPLEMATION

We implement LeNet in the following code. It includes two convolutional layers, each followed by ReLU and max-pooling, and three fully connected layers for classification. The forward function processes the input through these layers to produce class predictions.

Code Block 1: model.py

```python
class LeNet(nn.Module):
    def __init__(self, input_channels=1, num_classes=10):
        super().__init__()
        # TODO
        self.conv1 = nn.Conv2d(input_channels, 6, kernel_size=5)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, num_classes)
    def forward(self, x):
        # TODO
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, kernel_size=2, stride=2)
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, kernel_size=2, stride=2)
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

To accommodate different hardware configurations, we modify `train.py` to support both single-GPU and multi-GPU parallel training. For single GPU usage, we ensure that the model and data are transferred to the designated GPU device.

Code Block 2: train_gpu.py

```python
cfg["device"] = torch.device("cuda" if torch.cuda.is\_available()
    else "cpu")
print(f"Using device: {cfg['device']}")
```

For multi-GPU training, we implement data parallelism using `nn.DataParallel` module. This allows the model to be distributed across multiple GPUs, with the data split into batches for parallel processing.

Code Block 3: train_multi_gpu.py

```python
os.environ["CUDA_VISIBLE_DEVICES"] = "0,1,2,3"
```

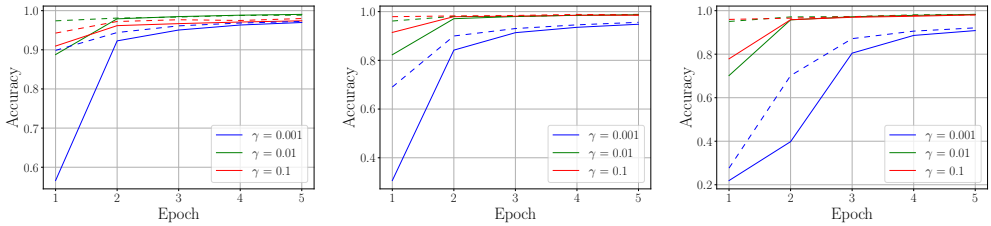
```
...
model = LeNet(cfg["num_channels"], cfg["num_classes"]).to(cfg["
    device"])
if torch.cuda.device_count() > 1:
    model = nn.DataParallel(model)
```

3 EXPERIMENTS

Define the learning rate and batch size of stochastic gradient descent as γ and N_s separately. Also define the learning rate and batch size of Adam as η and N_a. Additionally, we denote the total time of the program by T. Let v denote the number of iterations per second. Figures in this section are drawn by `matplotlib` in Python. Solid lines represent the training results, while dashed lines indicate the validation results. Detailed code and explanation can be found in Appendix A.

3.1 STOCHASTIC GRADIENT DESCENT

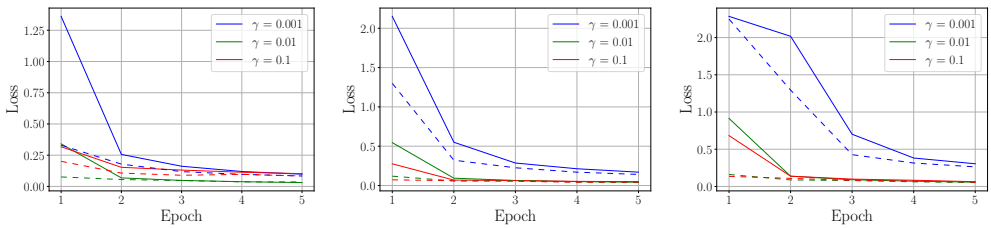
Figure 1 depicts the comparison of the accuracy of stochastic gradient descent with different γ. We observe that larger learning rates lead to faster convergence in terms of accuracy, especially in the early epochs. For smaller batch sizes in Figure 1(a), models with higher learning rates, such as $\gamma = 0.1$, achieve near-optimal accuracy much faster than those with lower γ. As the batch size increases, the convergence is slower, but by the end of training, the accuracy is comparable across different batch sizes.



(a) $N_s = 64$ (b) $N_s = 128$ (c) $N_s = 256$

Figure 1: Comparison on the accuracy of SGD between different γ

The comparison of the loss of stochastic gradient descent with different γ is shown in Figure 2. Higher learning rates result in a more rapid reduction in loss, especially in the first few epochs. In contrast, smaller batch sizes again show faster initial improvements, though larger batch sizes tend to reduce variance and produce smoother learning curves.

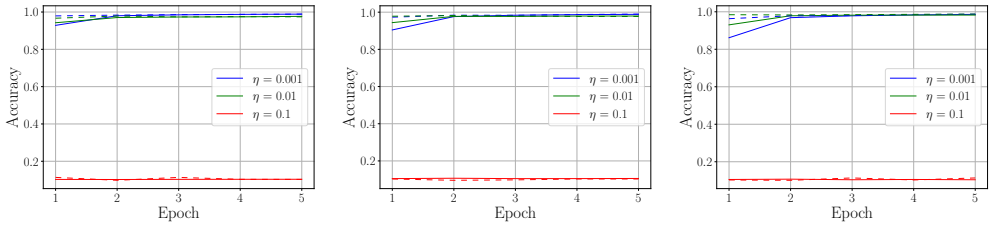


(a) $N_s = 64$ (b) $N_s = 128$ (c) $N_s = 256$

Figure 2: Comparison on the loss of SGD between different γ

3.2 ADAM

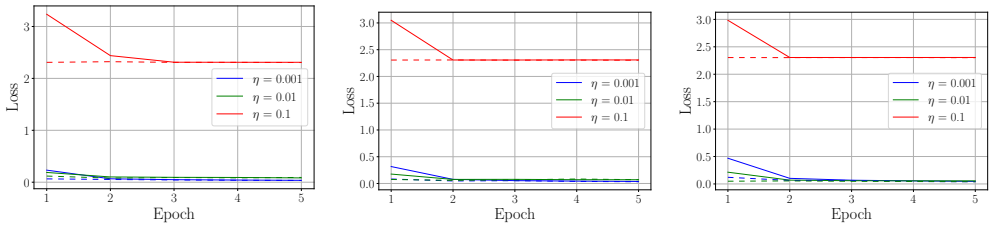
We compare the accuracy of Adam with different learning rates and batch size in Figure 3. It shows that the accuracy for large learning rates such as $\eta = 0.1$ performs poorly across all batch sizes, showing minimal improvement over the epochs. On the other hand, $\eta = 0.01$ and $\eta = 0.001$ lead to much higher accuracy with little difference between them. For all batch sizes, both of them converge quickly, achieving near-maximum accuracy by the second epoch.



(a) $N_a = 64$ (b) $N_a = 128$ (c) $N_a = 256$

Figure 3: Comparison on the accuracy of Adam between different η

Figure 4 performs a detailed comparison of the loss of Adam with different parameters. The results highlight that a higher learning rate of $\eta = 0.1$ results in a much slower loss reduction, with the model struggling to minimize loss across all batch sizes. Meanwhile, lower learning rates $\eta = 0.01$ and $\eta = 0.001$ result in faster and more stable loss reduction. The models with these lower learning rates converge to much smaller losses within the first few epochs, and the curves remain relatively stable as training progresses.



(a) $N_a = 64$ (b) $N_a = 128$ (c) $N_a = 256$

Figure 4: Comparison on the loss of Adam between different η

We note that Adam is less sensitive to the learning rate than stochastic gradient descent. While a high learning rate of $\eta = 0.1$ severely affects Adam's performance, learning rates $\eta = 0.001$ and $\eta = 0.01$ yield similarly strong results across different batch sizes. In contrast, the choice of learning rate in stochastic gradient descent has a more pronounced impact on both accuracy and loss, with higher learning rates generally leading to faster convergence, but at the risk of instability or divergence if the rate is too large.

Comparing both optimizers, we see that Adam consistently achieves high accuracy with low loss across a range of learning rates and batch sizes, making it a more robust choice for this particular task. Stochastic gradient descent requires more careful tuning to avoid sub-optimal performance or unstable training. Therefore, while stochastic gradient descent can be more effective when properly tuned, Adam offers a more reliable and less sensitive approach, particularly when learning rates are not extensively optimized.

4 ADDITIONAL EXPERIMENTS

We conduct additional experiments on different hardware with the same parameters $\eta = 0.01$ and $N_a = 128$. Figure 5 shows the total time T for running the program across different hardware setups, including CPU and 1 to 4 GPUs. The results show that using a single GPU reduces the total time compared to the CPU. However, adding more GPUs does not continue to decrease the total

time, and the improvements level off. Figure 6 presents the iteration speed v in terms of iterations per second. A single GPU also outperforms both the CPU and multi-GPU setups in terms of iteration speed.

The reason for the diminishing returns with multiple GPUs is the overhead involved in coordinating between them. While one GPU can process tasks efficiently, adding more GPUs introduces communication and synchronization costs, which can outweigh the benefits of parallel computation. This overhead leads to a slower iteration speed v and no further reduction in total time T when scaling beyond one GPU. This suggests that for small tasks, adding more GPUs can be inefficient due to the increased complexity of task distribution and communication.

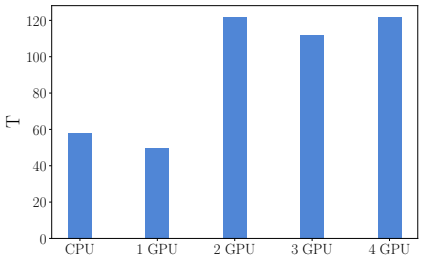


Figure 5: Comparison on T

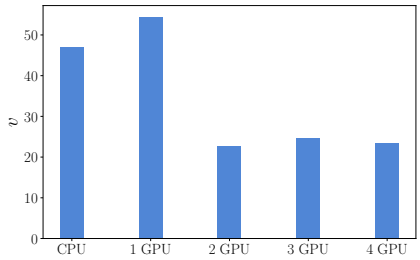


Figure 6: Comparison on v

5 Discussion

The experimental results highlight important distinctions between the performance of Stochastic Gradient Descent and Adam optimizers when applied to the LeNet architecture trained on the MNIST dataset. From the accuracy and loss comparison across different learning rates and batch sizes, it is clear that Adam consistently performs better in terms of stability and convergence speed. Adam is less sensitive to the learning rate, with even relatively high learning rates yielding stable and accurate results. In contrast, SGD requires careful tuning, as its performance is more susceptible to changes in learning rate, with higher learning rates leading to faster convergence but at a greater risk of instability.

Metric	SGD	Adam
Sensitivity	High	Low
Convergence speed	Slower, dependent on γ	Faster, stable across η
Stability	Less stable, sensitive to γ	More stable, less sensitive to η
Performance	Risk of instability	Relatively stable
Hyperparameter tuning effort	High	Low
Preferred for	Simpler, well-tuned models	Complex, default settings

Table 1: Comparison between SGD and Adam optimizers

In terms of computational efficiency, our additional experiments demonstrate the trade-offs between using a CPU, a single GPU, and multiple GPUs. While GPU-accelerated training significantly reduces training time compared to CPU-based computations, adding more GPUs beyond a single one does not continue to improve performance due to communication and synchronization overhead.

6 Conclusion

In this report, we have demonstrated that the Adam optimizer outperforms the Stochastic Gradient Descent in terms of robustness and stability when applied to the LeNet model on the MNIST dataset. Adam is less sensitive to learning rate variations, making it more suitable for scenarios where hyperparameter tuning is limited. Additionally, while GPU usage significantly reduces training time compared to CPU, the benefits of adding multiple GPUs diminish due to communication overhead. Thus, for smaller tasks, a single GPU provides the best balance between efficiency and performance.

REFERENCES

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256. JMLR Workshop and Conference Proceedings, 2010.

Diederik P Kingma. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pp. 400–407, 1951.

A USE OF TENSORBOARD

I claim that I use TensorBoard to visualize the experimental results during training. Then I exported the data and used `matplotlib` to generate the plots, ensuring the correct formatting of fonts and mathematical symbols.

Code Block 4: draw.py

```python
def load_tensorboard_data(log_dir):
    event_acc = EventAccumulator(log_dir)
    event_acc.Reload()
    print("Available tags:", event_acc.Tags()['scalars'])
    train_acc = event_acc.Scalars('train_accu')
    train_loss = event_acc.Scalars('train_loss')
    val_acc = event_acc.Scalars('val  _accu')
    val_loss = event_acc.Scalars('val  _loss')

    steps_acc = [x.step for x in train_acc]
    values_acc = [x.value for x in train_acc]
    steps_loss = [x.step for x in train_loss]
    values_loss = [x.value for x in train_loss]
    steps_val_acc = [x.step for x in val_acc]
    values_val_acc = [x.value for x in val_acc]
    steps_val_loss = [x.step for x in val_loss]
    values_val_loss = [x.value for x in val_loss]

    return (steps_acc, values_acc), (steps_loss, values_loss), (
        steps_val_acc, values_val_acc), (steps_val_loss,
        values_val_loss)
```