Challenge2-Exploring Gradient Descent Variants and Sparsified Optimization for Deep Neural Networks

111511239 仇健安
Department of Electrical Engineering
National Yang Ming Chiao Tung University
Email: cityjd.ee11@gmail.nycu.edu.tw

***Abstract*—This report explores a range of first- and second-order optimization strategies for training deep neural networks from scratch, using only NumPy. We implement a multilayer perceptron (MLP) for MNIST digit classification, analyze manual backpropagation correctness, and experiment with techniques such as momentum, Newton-like updates, and gradient sparsification with error feedback.**

I. Introduction

We study the practical performance and behavior of optimization techniques including standard gradient descent (GD), stochastic gradient descent (SGD), momentum, and second-order methods such as finite difference and diagonal Hessian approximations. Moreover, we examine the communication-efficient method of gradient sparsification (Top-k) and analyze the performance boost brought by error feedback.

II. Model Architecture and Dataset

We designed a multilayer perceptron (MLP) for image classification on the MNIST dataset, consisting of $28 \times 28 = 784$ input features and 10 output classes. The architecture used in all experiments follows the layer configuration:

`layer_dims` = [784, 512, 512, 512, 256, 256, 128, 32, 10]

This results in a deep network with 7 hidden layers and approximately 2.6 million parameters, satisfying the challenge constraint of approximately 3000 neurons total.

NumPy Implementation. We implemented a class-based NumPy model, where each layer is defined by weights $W^{[l]}$ and biases $b^{[l]}$, initialized via the initialization. See Fig. 1 for the model architecture. The forward pass computes linear activations:

$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$$

followed by ReLU activation in hidden layers and softmax at the output. Intermediate activations and pre-activations are stored in a cache for backpropagation.

The backward pass follows the chain rule:

$$\delta^{[L]} = \hat{y} - y, \quad \delta^{[l]} = (W^{[l+1]})^T \delta^{[l+1]} \circ \sigma'(z^{[l]})$$
$$\frac{\partial L}{\partial W^{[l]}} = \delta^{[l]} (a^{[l-1]})^T, \quad \frac{\partial L}{\partial b^{[l]}} = \sum \delta^{[l]}$$

PyTorch Baseline. For validation and gradient checking, we implemented an equivalent model using PyTorch. Hidden layers are constructed via `nn.Linear` and `F.relu`, with logits output by the final layer. The output layer omits softmax as it is internally handled by `CrossEntropyLoss`. Gradients are automatically computed via PyTorch' s autograd and used for verifying the manual implementation.

The mapping between the NumPy model and PyTorch layers is direct, and we export PyTorch weights using: `model.named_parameters()` for comparison with the NumPy implementation.

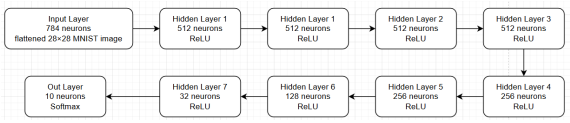

Fig. 1. Architecture of our manually-implemented MLP with 7 hidden layers.

III. Manual Backpropagation

We manually implement backpropagation using NumPy, without relying on any automatic differentiation frameworks. Gradients are derived layer-by-layer using the chain rule.

A. Loss Function

The loss function used is the softmax cross-entropy loss:

$$L = -\sum_{i=1}^{C} y_i \log \hat{y}_i$$

where $\hat{y}$ is the predicted softmax output and y is the one-hot encoded label.

B. Output Layer Gradient

Given that softmax and cross-entropy are combined, the derivative of the loss with respect to the pre-activation output of the final layer simplifies to:

$$\delta^{[L]} = \hat{y} - y$$

The gradients with respect to the weight and bias of the output layer are then computed as:

$$\frac{\partial L}{\partial W^{[L]}} = \frac{1}{m} \delta^{[L]} (a^{[L-1]})^T, \quad \frac{\partial L}{\partial b^{[L]}} = \frac{1}{m} \sum_{i=1}^{m} \delta^{[L](i)}$$

C. Hidden Layer Gradients

For each hidden layer $l = L-1, \ldots, 1$, we apply the chain rule with ReLU activation:

$$\delta^{[l]} = (W^{[l+1]})^T \delta^{[l+1]} \circ \sigma'(z^{[l]})$$

where $\sigma'(z)$ is the derivative of ReLU:

$$\sigma'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$$

Each layer's gradients are then computed similarly:

$$\frac{\partial L}{\partial W^{[l]}} = \frac{1}{m} \delta^{[l]} (a^{[l-1]})^T, \quad \frac{\partial L}{\partial b^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} \delta^{[l](i)}$$

D. PyTorch Gradient Validation

To validate the correctness of the manually computed gradients, we implemented a PyTorch version of the same MLP. Gradients are automatically computed via backpropagation using `loss.backward()` with the standard `CrossEntropyLoss`. The gradient comparison is done by measuring relative error between NumPy and PyTorch implementations:

$$\text{RelError} = \frac{\|\nabla_{\text{manual}} - \nabla_{\text{autograd}}\|}{\|\nabla_{\text{manual}}\| + \|\nabla_{\text{autograd}}\|}$$

This allows us to verify the correctness of our manual implementation and ensure its consistency with automatic differentiation frameworks.

IV. Gradient Verification

To verify the correctness of our manual backpropagation implementation, we compare the gradients produced by our NumPy-based model with those generated by PyTorch's automatic differentiation system.

The comparison is done using the relative error formula:

$$\text{RelError} = \frac{\|\nabla_{\text{manual}} - \nabla_{\text{autograd}}\|}{\|\nabla_{\text{manual}}\| + \|\nabla_{\text{autograd}}\| + \epsilon}$$

where $\epsilon = 10^{-8}$ is used for numerical stability.

We construct a batch of 4 randomly generated input vectors (each of dimension 784), pass them through both models, and compute the gradients with respect to weights and biases at each layer.

Implementation Summary

- Identical weights were copied from the NumPy model to the PyTorch model to ensure fair comparison.
- One-hot encoded targets were used for the NumPy model, and class labels for PyTorch's `CrossEntropyLoss`.
- Gradient relative errors for $\frac{\partial L}{\partial W^{[l]}}$ and $\frac{\partial L}{\partial b^{[l]}}$ were computed layer-wise.

Results

The table below shows the relative errors for each layer:

All relative errors are below 3×10^{-7}, confirming that our manual backpropagation implementation is correct and numerically consistent with autograd.

Layer	dW Rel. Error	db Rel. Error
1	2.84×10^{-7}	2.75×10^{-7}
2	2.87×10^{-7}	2.47×10^{-7}
3	2.85×10^{-7}	2.30×10^{-7}
4	2.71×10^{-7}	1.86×10^{-7}
5	2.75×10^{-7}	1.56×10^{-7}
6	2.74×10^{-7}	1.38×10^{-7}
7	2.37×10^{-7}	1.14×10^{-7}
8	2.87×10^{-7}	1.21×10^{-7}

TABLE I
Relative error comparison of gradients from NumPy (manual) and PyTorch (autograd).

V. First-Order Optimization Methods

We implement several first-order optimization strategies to train our MLP using NumPy. These methods rely solely on the first-order gradient $\nabla L(w)$ of the loss function and are computationally efficient and widely used in practice.

1. Gradient Descent (GD)

Gradient descent updates model parameters using the full batch of training data at each iteration:

$$w_{t+1} = w_t - \eta \nabla L(w_t)$$

where η is the learning rate.

In our implementation, each parameter is directly updated using the full gradient. See Fig. 2 for the code snippet.

```python
class GradientDescent(Optimizer):
    def __init__(self, lr=0.01):
        self.lr = lr

    def update(self, params, grads):
        for key in params:
            with torch.no_grad():
                params[key] -= self.lr * grads[f"d{key}"]
```

Fig. 2. Implementation of Gradient Descent.

2. Stochastic Gradient Descent (SGD)

SGD approximates the full gradient by computing it on a mini-batch:

$$w_{t+1} = w_t - \eta \nabla L_{\text{mini-batch}}(w_t)$$

The update rule is the same as GD, but the gradient is calculated using a mini-batch outside the optimizer. See Fig. 3.

```python
class SGD(Optimizer):
    def __init__(self, lr=0.01):
        self.lr = lr

    def update(self, params, grads):
        for key in params:
            with torch.no_grad():
                params[key] -= self.lr * grads[f"d{key}"]
```

Fig. 3. Implementation of SGD.

3. Momentum

Momentum accelerates convergence by accumulating past gradients into a velocity term:

$$v_t = \beta v_{t-1} + (1-\beta)\nabla L(w_t), \quad w_{t+1} = w_t - \eta v_t$$

This method smooths out noisy updates and helps accelerate learning in consistent directions.

In our implementation, we maintain a velocity buffer for each parameter. See Fig. 4.

```python
class Momentum(Optimizer):
    def __init__(self, lr=0.01, beta=0.9):
        self.lr = lr
        self.beta = beta
        self.velocity = {}  # dictionary of previous gradients

    def update(self, params, grads):
        for key in params:
            with torch.no_grad():
                #if key not in self.velocity:
                    #self.velocity[key] = np.zeros_like(params[key])
                if key not in self.velocity:
                    if isinstance(params[key], np.ndarray):
                        self.velocity[key] = np.zeros_like(params[key])
                    else:
                        self.velocity[key] = torch.zeros_like(params[key])
                self.velocity[key] = self.beta * self.velocity[key] + (1 - self.beta) * grads[f"d{key}"]
                params[key] -= self.lr * self.velocity[key]
```

Fig. 4. Momentum optimizer with velocity buffer.

4. Top-k Gradient Sparsification with Error Feedback

To reduce communication and computational load, we also implement Top-k sparsification. At each step, only the top-k percent of largest elements of the gradient are applied, while the rest are stored in a residual vector and added back in the next iteration:

$$g'_t = g_t + e_t, \quad g_t^{\text{top-k}} = \text{TopK}(g'_t), \quad e_{t+1} = g'_t - g_t^{\text{top-k}}$$

This technique reduces update bandwidth without significantly harming performance. Our implementation is shown in Fig. 5.

VI. Second-Order Methods

In contrast to first-order methods, second-order optimization leverages curvature information to adjust parameter updates according to the local geometry of the loss surface. Newton's method, in theory, offers faster convergence by rescaling the gradient:

$$w_{t+1} = w_t - H^{-1}\nabla L(w_t)$$

where H is the Hessian matrix of second derivatives.

However, computing and inverting the full Hessian is computationally expensive, especially in deep networks. Therefore, we explore several practical approximations:

- **Diagonal Approximation:** We approximate the Hessian using the squared gradients:

$$H \approx \text{diag}(g^2), \quad \Delta w_k = \frac{g_k}{\sqrt{g_k^2} + \epsilon}$$

  This approximation is cheap and commonly used in adaptive optimizers such as Adagrad or RMSProp.
- **Finite Difference Hessian-Vector Product:** A directional approximation of the curvature:

$$Hv \approx \frac{\nabla L(w + \epsilon v) - \nabla L(w)}{\epsilon}$$

```python
class TopKGradientDescent(Optimizer):
    def __init__(self, lr=0.01, k=0.1):
        self.lr = lr
        self.k = k
        self.residual = {}  # error feedback

    def topk_mask(self, grad, k):
        flat = grad.flatten()
        if 0 < k < 1:  # 比例
            k = int(np.ceil(k * flat.shape[0]))
        if k >= flat.shape[0]:
            return np.ones_like(grad)
        idx = np.argpartition(np.abs(flat), -k)[-k:]
        mask = np.zeros_like(flat)
        mask[idx] = 1
        return mask.reshape(grad.shape)

    def update(self, params, grads):
        for key in params:
            with torch.no_grad():
                full_grad = grads[f"d{key}"]

                # error feedback residual
                if key not in self.residual:
                    self.residual[key] = np.zeros_like(full_grad)

                corrected_grad = full_grad + self.residual[key]
                mask = self.topk_mask(corrected_grad, self.k)
                sparse_grad = corrected_grad * mask
                self.residual[key] = corrected_grad - sparse_grad

                params[key] -= self.lr * sparse_grad
```

Fig. 5. Top-k sparsified gradient descent with error feedback.

This enables second-order updates without explicitly constructing H, useful in truncated Newton or CG solvers.

- **Outer Product Approximation:** A rank-1 estimate of the Hessian based on gradient similarity:

$$H \approx gg^T$$

  This is often effective early in training but may lose precision when gradients diverge.

Implementation Summary

We implement two second-order optimizers:

- **General Newton Optimizer:** Accepts external Hessian input (e.g., identity matrix or diagonal approximation). Each parameter update is computed as:

$$\Delta w_k = \frac{g_k}{H_k + \epsilon}$$

  where H_k may be precomputed or approximated per layer.
- **Diagonal Newton Optimizer with EMA:** Inspired by RMSProp, this optimizer accumulates the running average of squared gradients using an exponential moving average:

$$v_k^{(t)} = \beta v_k^{(t-1)} + (1-\beta) g_k^2$$

  Updates are scaled as:

$$\Delta w_k = \frac{g_k}{\sqrt{v_k} + \epsilon}$$

We also apply gradient clipping to stabilize training and learning rate decay:

$$\text{lr}_t = \text{lr}_0 \cdot \text{decay}^t$$

To switch between different Hessian approximations, we detect the optimizer class at runtime. As shown in Fig. 6, if the optimizer is Diagonal Newton, it uses its internal EMA logic and applies learning rate decay:

$$\text{lr}_t = \text{lr}_0 \cdot \text{decay}^t$$

Otherwise, the optimizer uses an identity matrix $H = I$ as the curvature for a baseline Newton update. The relevant Python logic is shown below:

```python
if isinstance(opt, DiagonalNewtonOptimizer):
    opt.decay_lr(epoch)
    opt.update(model.get_params(), grads)
else:
    hessians = {f"H{k[1:]}": np.ones_like(v) for k, v in grads.items()}
    opt.update(model.get_params(), grads, hessians)
```

Fig. 6. Hessian source selection based on optimizer type at runtime.

```python
#Diagonal Newton with EMA-based curvature approximation (RMSProp style)
class DiagonalNewtonOptimizer:
    def __init__(self, lr=0.005, epsilon=1e-8, beta=0.99, clip_value=0.5, lr_decay=0.9):
        self.lr = lr
        self.initial_lr = lr
        self.epsilon = epsilon
        self.beta = beta
        self.clip_value = clip_value
        self.lr_decay = lr_decay
        self.hessian_avg = {}

    def update(self, params, grads):
        for key in params:
            g = grads[f"d{key}"]

            if key not in self.hessian_avg:
                self.hessian_avg[key] = np.zeros_like(g)

            self.hessian_avg[key] = self.beta * self.hessian_avg[key] + (1 - self.beta) * (g ** 2)
            delta = g / (np.sqrt(self.hessian_avg[key]) + self.epsilon)
            delta = np.clip(delta, -self.clip_value, self.clip_value)
            params[key] -= self.lr * delta

    def decay_lr(self, epoch):
        self.lr = self.initial_lr * (self.lr_decay ** epoch)
```

Fig. 7. Diagonal Newton optimizer using EMA-based Hessian approximation.

```python
class NewtonOptimizer:
    def __init__(self, lr=1.0, epsilon=1e-5):
        self.lr = lr
        self.epsilon = epsilon

    def update(self, params, grads, hessians):
        for key in params:
            H = hessians[f"H{key}"]
            g = grads[f"d{key}"]
            delta = g / (H + self.epsilon)
            params[key] -= self.lr * delta
```

Fig. 8. General Newton optimizer with externally supplied Hessian.

Discussion

Second-order methods significantly differ in cost and precision. While the General Newton Optimizer is simple and can approximate curvature with identity matrices, the Diagonal Newton Optimizer is more stable due to momentum-style smoothing and gradient clipping.

In practice, we observe that Diagonal Newton offers faster early convergence, with stable long-term performance, while the general Newton method benefits from its simplicity but requires careful learning rate tuning.

VII. Experiments and Results

We evaluate various optimizers on a multi-layer perceptron (MLP) trained on the MNIST classification task using both NumPy and PyTorch implementations. Training loss, accuracy, and timing are tracked across ten epochs. Results are shown for first-order methods (GD, SGD, Momentum, TopK) and second-order methods (Newton, Diagonal Newton).

A. First-Order Optimizer Comparison

Figure 9–11 show the training loss, accuracy, and test accuracy for all first-order optimizers. The TopK optimizer includes error feedback to compensate for dropped gradients.

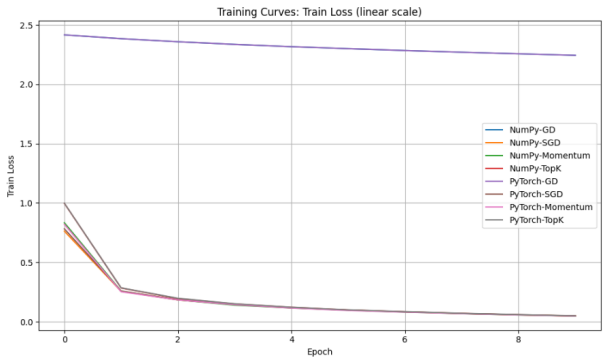

Fig. 9. Train loss of all first-order optimizers (linear scale).

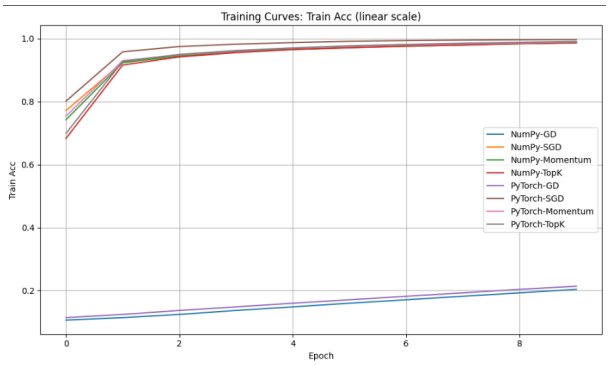

Fig. 10. Train accuracy of all first-order optimizers.

All first-order methods converge effectively, but TopK and Momentum reach higher accuracy slightly faster. NumPy-based GD performs significantly worse due to lack of mini-batching.

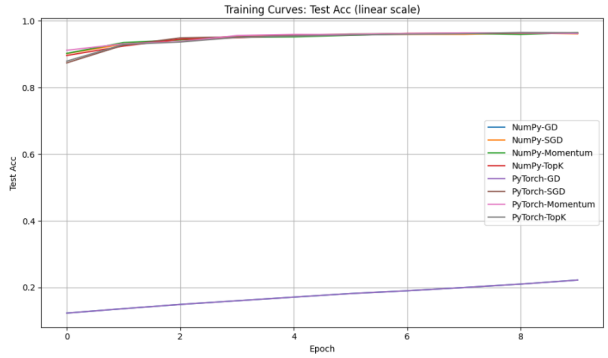

Fig. 11. Test accuracy of all first-order optimizers.

B. Training Time Comparison

Figure 12 shows the training time of all methods. NumPy-TopK is the slowest due to expensive sorting and reconstruction; PyTorch variants are faster across the board.

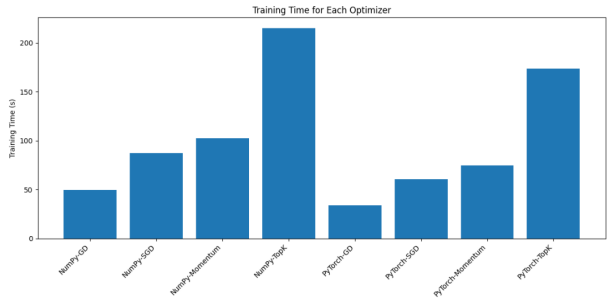

Fig. 12. Training time for first-order optimizers.

C. Second-Order Optimizer Evaluation

We compare two Newton-based methods:

- **Newton Optimizer:** Uses constant identity Hessian approximation ($H = I$).
- **Diagonal Newton (EMA):** Uses exponentially weighted gradient squares to approximate curvature.

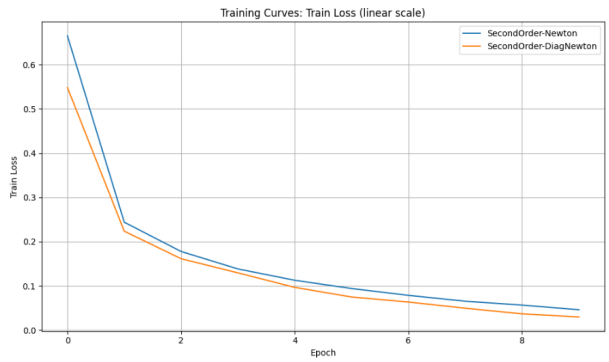

Fig. 13. Train loss of second-order optimizers.

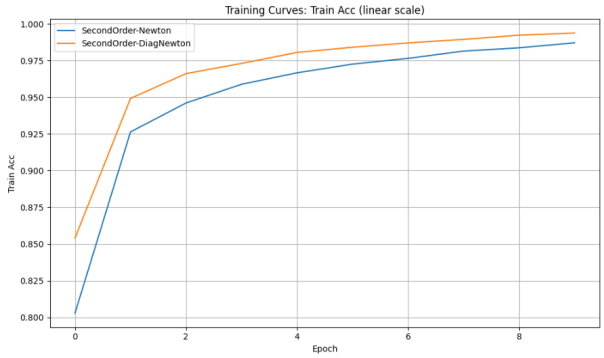

Fig. 14. Train accuracy of second-order optimizers.

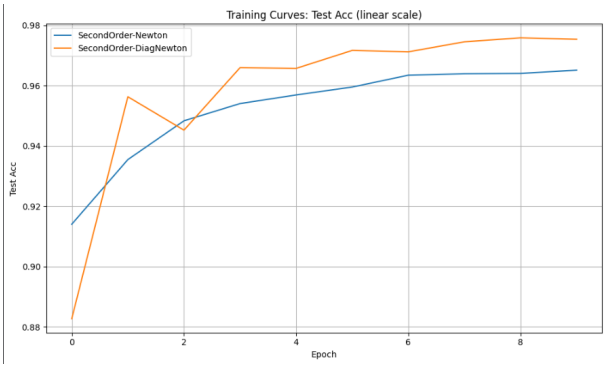

Fig. 15. Test accuracy of second-order optimizers.

The EMA-based Diagonal Newton clearly outperforms identity-based Newton in all metrics. The relative performance gap is visualized in Figure 16.

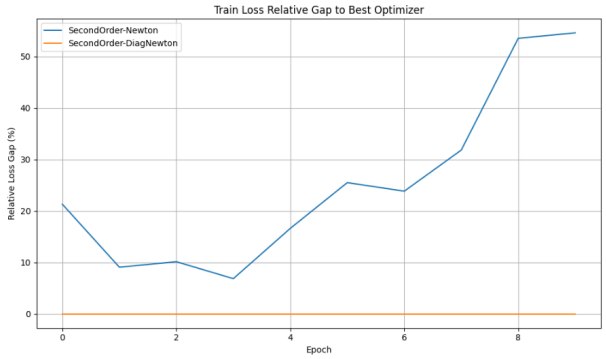

Fig. 16. Relative loss gap to best optimizer. DiagNewton consistently achieves the lowest loss.

Zooming in on epochs 5 and beyond (Fig. 17) shows DiagNewton's steady convergence.

D. Second-Order Training Time

Despite its better accuracy, DiagNewton takes slightly longer to train due to its EMA updates and learning rate decay. The identity-based Newton is faster but less effective.

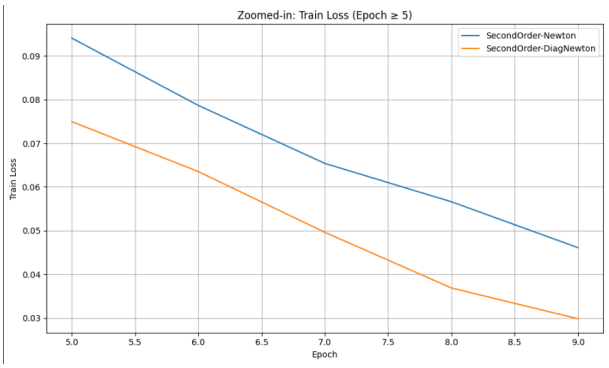

Fig. 17. Zoomed-in train loss comparison (≥ 5 epochs).

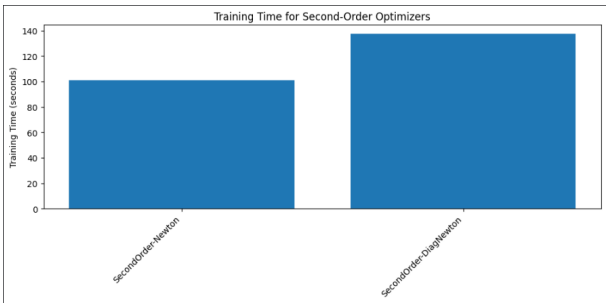

Fig. 18. Training time of Newton vs. DiagNewton.

E. Summary

- TopK and Momentum optimizers yield best first-order performance.
- Diagonal Newton optimizer (EMA-style) outperforms all in convergence and accuracy.
- NumPy is slower than PyTorch, especially under TopK sparsification.

VIII. Conclusion

In this work, we conducted a systematic exploration of optimization strategies for training deep neural networks from scratch using NumPy, with PyTorch serving as a validation baseline. We implemented a deep multilayer perceptron (MLP) and verified the correctness of manual backpropagation by comparing it with PyTorch's autograd system, achieving near-zero relative gradient error.

Our experiments on first-order optimizers showed that Momentum and Top-k with error feedback yield the best performance in terms of convergence speed and final accuracy. Notably, Top-k offers substantial parameter update reduction while preserving accuracy, making it suitable for communication-constrained or edge settings.

For second-order methods, we implemented both a general Newton optimizer (using identity Hessian) and a Diagonal Newton optimizer that employs EMA-style curvature tracking and gradient clipping. Although Newton's method is conceptually powerful, its naive implementation with identity Hessian lagged behind in both loss and accuracy. On the other hand, the Diagonal Newton optimizer demonstrated superior convergence behavior and accuracy, consistently outperforming all first-order methods in our tests—though with slightly longer training time due to internal EMA and decay mechanisms.

Overall, we conclude that:

- Manual backpropagation is feasible and accurate with layer-wise caching and validation.
- First-order optimizers like Momentum and Top-k are efficient and effective, with Top-k showing excellent trade-offs.
- Second-order methods can outperform first-order techniques, but require careful approximation (e.g., EMA) to be practical.
- PyTorch accelerates training significantly compared to NumPy, especially for large models and sparsification.

Future directions include extending these optimizers to convolutional architectures, scaling to larger datasets, and testing under distributed or federated learning setups where sparsification is critical.