ECE 685D – Homework 2

Yiming Mao

Fall 2025

Problem 1: Backprop on a Residual MLP (30 pts)

We consider a 3-layer residual neural network with hidden layers:

$$a_\ell = W_\ell^\top h_{\ell-1} + b_\ell, \quad h_\ell = g(a_\ell) + h_{\ell-1}, \quad \ell = 1, 2,$$

and output

$$\hat{y} = W_3^\top h_2 + b_3.$$

The loss is the mean squared error (MSE):

$$L(Y, \hat{Y}) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2.$$

(a) Derive $\frac{\partial L}{\partial W_3}$ and $\frac{\partial L}{\partial b_3}$

For each sample i,

$$\frac{\partial L}{\partial \hat{y}_i} = \frac{2}{N}(\hat{y}_i - y_i) \doteq r_i.$$

Let $H_2 = [h_2^{(1)}, \ldots, h_2^{(N)}]$ and $R = [r^{(1)}, \ldots, r^{(N)}]$, then

$$\boxed{\frac{\partial L}{\partial W_3} = H_2 R^\top, \qquad \frac{\partial L}{\partial b_3} = \sum_{i=1}^{N} r^{(i)}.}$$

(b) Show the residual recursion

Define

$$e_\ell = \frac{\partial L}{\partial h_\ell}, \qquad \delta_\ell = \frac{\partial L}{\partial a_\ell} = e_\ell \odot g'(a_\ell).$$

Since $h_\ell = g(a_\ell) + h_{\ell-1}$, by chain rule

$$\frac{\partial L}{\partial h_{\ell-1}} = \frac{\partial L}{\partial h_\ell} \frac{\partial h_\ell}{\partial h_{\ell-1}} = e_\ell \cdot I + W_\ell \delta_\ell.$$

Thus the residual recursion is

$$\boxed{e_{\ell-1} = e_\ell + W_\ell \delta_\ell.}$$

(c) Derive $\frac{\partial L}{\partial W_2}, \frac{\partial L}{\partial b_2}, \frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial b_1}$ (without e_ℓ, δ_ℓ)

From part (a),

$$e_2 = \frac{\partial L}{\partial h_2} = W_3 r, \qquad \delta_2 = (W_3 r) \odot g'(a_2).$$

Then

$$e_1 = e_2 + W_2 \delta_2 = W_3 r + W_2\big[(W_3 r) \odot g'(a_2)\big],$$
$$\delta_1 = e_1 \odot g'(a_1).$$

Hence the gradients are

$$\boxed{\frac{\partial L}{\partial W_2} = H_1 \Delta_2^\top, \quad \frac{\partial L}{\partial b_2} = \sum_i \delta_2^{(i)}}$$

$$\boxed{\frac{\partial L}{\partial W_1} = H_0 \Delta_1^\top, \quad \frac{\partial L}{\partial b_1} = \sum_i \delta_1^{(i)}}$$

where $H_0 = X$ (input features), H_1 is the first hidden layer output, $\Delta_2 = [(W_3 r) \odot g'(a_2)]$, and $\Delta_1 = \big(W_3 r + W_2[(W_3 r) \odot g'(a_2)]\big) \odot g'(a_1)$.

Problem 2: Customized Multi-View Dataset (30 pts)

In this problem, we are asked to build a custom dataset `MNISTTwoView` by inheriting from `torch.utils.data.Dataset`. For each MNIST sample, instead of returning just one image and its label, we return:

$$(x^{(1)}, x^{(2)}, y),$$

where $x^{(1)}$ and $x^{(2)}$ are two *independently augmented* views of the same image. This type of "two-view" dataset is commonly used in modern self-supervised learning frameworks.

Both views are generated using the *same transform pipeline* with independent randomness (e.g., random crop/resize, random affine, random erasing). We use `torchvision.datasets.MNIST` to download the data directly.

(a) Dataset Implementation

```
import torch
from torch.utils.data import Dataset, DataLoader
from torchvision import datasets, transforms
import matplotlib.pyplot as plt

# Utility to apply the same transform twice
class TwoViewTransform:
    def __init__(self, base_transform):
        self.base = base_transform
    def __call__(self, x):
        return self.base(x), self.base(x)

class MNISTTwoView(Dataset):
    def __init__(self, root="./data", train=True, download=True):
        self.orig_transform = transforms.ToTensor()
        base_transform = transforms.Compose([
            transforms.RandomResizedCrop(28, scale=(0.8,1.0)),
            transforms.RandomAffine(15, translate=(0.1,0.1), shear=10),
            transforms.ToTensor(),
            transforms.RandomErasing(p=0.25, scale=(0.02,0.1), ratio=(0.3,3.3))
        ])
        self.two_view = TwoViewTransform(base_transform)
        self.mnist = datasets.MNIST(root=root, train=train, download=download)

    def __len__(self):
        return len(self.mnist)

    def __getitem__(self, idx):
        img, label = self.mnist[idx]
        orig = self.orig_transform(img)
        x1, x2 = self.two_view(img)
        return orig, x1, x2, label
```

(b) Visualization

We visualize one minibatch of size 8 as a grid, with three columns for each sample: the original image, $x^{(1)}$, and $x^{(2)}$. Labels are shown under the original images.

```
dataset = MNISTTwoView(train=True, download=True)
loader = DataLoader(dataset, batch_size=8, shuffle=True)

orig, x1, x2, labels = next(iter(loader))

fig, axes = plt.subplots(8, 3, figsize=(6,12))
for i in range(8):
    axes[i,0].imshow(orig[i,0], cmap="gray")
    axes[i,0].set_title(f"Label: {labels[i].item()}")
    axes[i,0].axis("off")

    axes[i,1].imshow(x1[i,0], cmap="gray")
    axes[i,1].set_title("x(1)")
    axes[i,1].axis("off")

    axes[i,2].imshow(x2[i,0], cmap="gray")
    axes[i,2].set_title("x(2)")
    axes[i,2].axis("off")

plt.tight_layout()
plt.show()
```

(c) Visualization Result

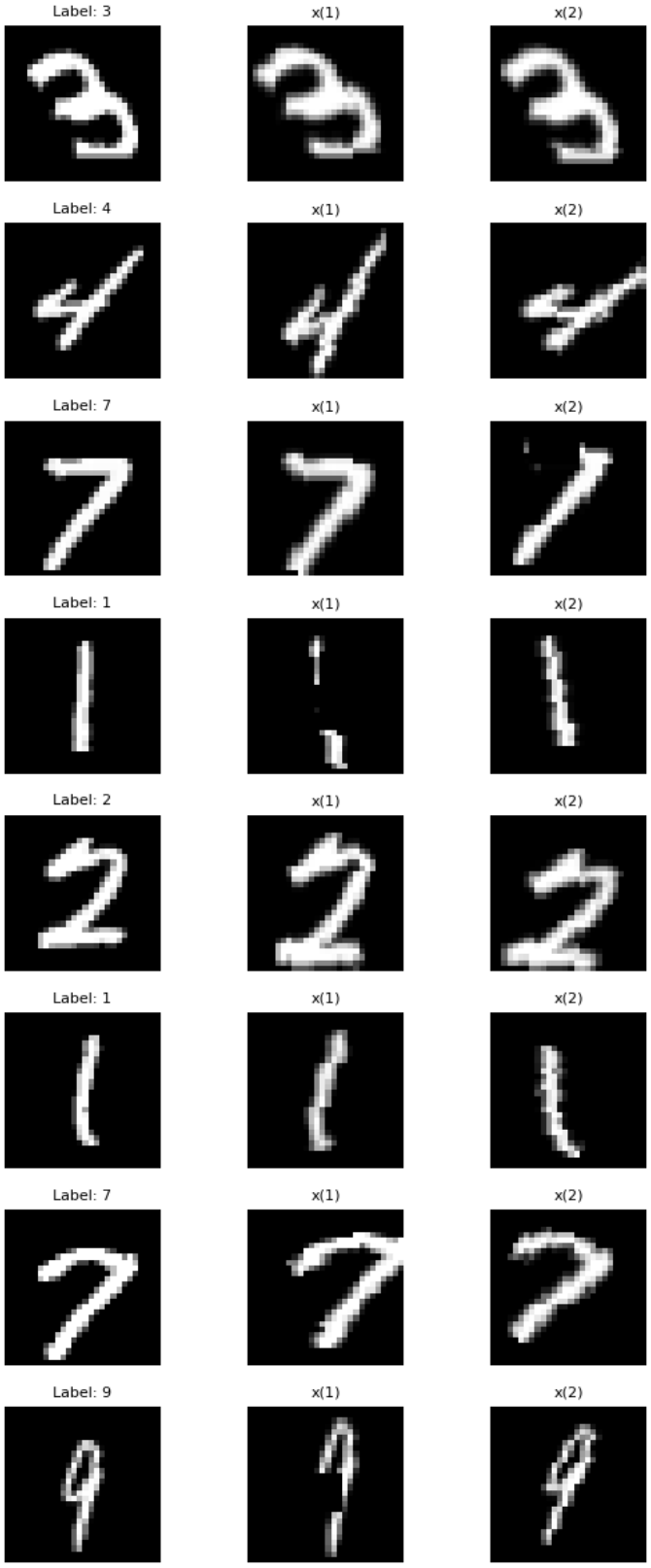

Figure 1: Example minibatch of size 8 from `MNISTTwoView`. Each row shows (original, $x^{(1)}$, $x^{(2)}$). Labels are shown under the original images.

Problem 3: Logistic Regression using Gradient and Newton's Methods (30 pts)

In this problem, we implement and compare three optimization methods for logistic regression on the Breast Cancer dataset (`sklearn.datasets.load_breast_cancer`):

1. Gradient Descent (GD),
2. Newton's Method with the exact Hessian,
3. Newton's Method with a diagonal approximation to the Hessian.

We use a test split of 30% and a learning rate of 0.1 where applicable.

(a) Derivation of Gradient and Hessian

The logistic regression model is defined by the sigmoid function:

$$\sigma(z) = \frac{1}{1+e^{-z}}, \quad p = \sigma(Xw),$$

where $X \in \mathbb{R}^{N\times d}$ is the data matrix, $w \in \mathbb{R}^d$ are the weights, and $y \in \{0,1\}^N$ are the labels.

The binary cross-entropy loss is:

$$L(w) = \sum_{n=1}^{N} \left[-y_n \log p_n - (1-y_n)\log(1-p_n)\right].$$

The gradient is:

$$\nabla_w L = X^\top (p-y).$$

The Hessian is:

$$H = X^\top R X, \quad R = \mathrm{diag}(p \odot (1-p)).$$

Thus the update rules are:

$$\text{GD: } w \leftarrow w - \eta \nabla L,$$

$$\text{Newton (Exact): } w \leftarrow w - H^{-1}\nabla L,$$

$$\text{Newton (Diagonal): } w \leftarrow w - \eta \frac{\nabla L}{\mathrm{diag}(H)}.$$

(b) Implementation

We implement the three solvers in `PyTorch`. Code is shown below.

```
import time
import numpy as np
import torch
from torch import nn
import matplotlib.pyplot as plt
from sklearn.datasets import load_breast_cancer
```

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# ---------- Helpers ----------
def bce_loss_logits(logits, y):
    """Binary cross entropy with logits (numerically stable)."""
    return nn.functional.binary_cross_entropy_with_logits(logits, y)

def accuracy_from_logits(logits, y):
    preds = (torch.sigmoid(logits) >= 0.5).float()
    return (preds.eq(y).float().mean().item())

def prepare_data(test_size=0.3, seed=0):
    """Load and preprocess Breast Cancer dataset."""
    X, y = load_breast_cancer(return_X_y=True)
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=test_size, random_state=seed, stratify=y
    )
    scaler = StandardScaler().fit(X_train)
    X_train = scaler.transform(X_train)
    X_test = scaler.transform(X_test)

    # Convert to torch tensors
    X_train = torch.tensor(X_train, dtype=torch.float32)
    y_train = torch.tensor(y_train, dtype=torch.float32).unsqueeze(1)
    X_test = torch.tensor(X_test, dtype=torch.float32)
    y_test = torch.tensor(y_test, dtype=torch.float32).unsqueeze(1)
    return X_train, y_train, X_test, y_test

# ---------- Gradient Descent ----------
def gradient_descent(lr=0.1, max_iter=200):
    Xtr, ytr, Xte, yte = prepare_data()
    N, d = Xtr.shape
    w = torch.zeros((d,1), dtype=torch.float32, requires_grad=True)

    tr_loss, te_loss, te_acc, times = [], [], [], []
    t0 = time.time()
    for it in range(max_iter):
        logits = Xtr @ w
        loss = bce_loss_logits(logits, ytr)
        loss.backward()
        with torch.no_grad():
            w -= lr * w.grad
        w.grad.zero_()

        with torch.no_grad():
            tr_loss.append(loss.item())
            te_logits = Xte @ w
            te_loss.append(bce_loss_logits(te_logits, yte).item())
            te_acc.append(accuracy_from_logits(te_logits, yte))
            times.append(time.time() - t0)
    return tr_loss, te_loss, te_acc, times

# ---------- Newton (Exact Hessian) ----------
```

```
def newton_exact(max_iter=30, damping=1e-6):
    Xtr, ytr, Xte, yte = prepare_data()
    N, d = Xtr.shape
    w = torch.zeros((d,1), dtype=torch.float32)

    tr_loss, te_loss, te_acc, times = [], [], [], []
    t0 = time.time()
    for it in range(max_iter):
        with torch.no_grad():
            logits = Xtr @ w
            p = torch.sigmoid(logits)

        grad = Xtr.T @ (p - ytr) # gradient [d,1]
        R = (p * (1 - p)).squeeze(1) # diag terms
        XR = Xtr * R.unsqueeze(1) # [N,d]
        H = Xtr.T @ XR # Hessian [d,d]
        H = H + damping * torch.eye(d) # damping for stability

        step = torch.linalg.solve(H, grad)
        w = w - step

        with torch.no_grad():
            tr_loss.append(bce_loss_logits(Xtr @ w, ytr).item())
            te_logits = Xte @ w
            te_loss.append(bce_loss_logits(te_logits, yte).item())
            te_acc.append(accuracy_from_logits(te_logits, yte))
            times.append(time.time() - t0)
    return tr_loss, te_loss, te_acc, times

# ---------- Newton (Diagonal Approximation) ----------
def newton_diag(max_iter=80, lr=1.0, eps=1e-8):
    Xtr, ytr, Xte, yte = prepare_data()
    N, d = Xtr.shape
    w = torch.zeros((d,1), dtype=torch.float32)

    tr_loss, te_loss, te_acc, times = [], [], [], []
    t0 = time.time()
    for it in range(max_iter):
        with torch.no_grad():
            logits = Xtr @ w
            p = torch.sigmoid(logits)

        grad = Xtr.T @ (p - ytr) # [d,1]
        R = (p * (1 - p)).squeeze(1)
        diagH = (Xtr.pow(2) * R.unsqueeze(1)).sum(dim=0, keepdim=True).T
        step = grad / (diagH + eps) # elementwise
        w = w - lr * step

        with torch.no_grad():
            tr_loss.append(bce_loss_logits(Xtr @ w, ytr).item())
            te_logits = Xte @ w
            te_loss.append(bce_loss_logits(te_logits, yte).item())
            te_acc.append(accuracy_from_logits(te_logits, yte))
            times.append(time.time() - t0)
```

```
    return tr_loss, te_loss, te_acc, times

# ---------- Main Experiment ----------
if __name__ == "__main__":
    gd_tr, gd_te, gd_acc, gd_t = gradient_descent(lr=0.1, max_iter=200)
    nx_tr, nx_te, nx_acc, nx_t = newton_exact(max_iter=30)
    nd_tr, nd_te, nd_acc, nd_t = newton_diag(max_iter=80, lr=1.0)

    # Plot Loss vs Time
    plt.figure(figsize=(6,4))
    plt.plot(gd_t, gd_te, label="GD (test loss)")
    plt.plot(nx_t, nx_te, label="Newton Exact (test loss)")
    plt.plot(nd_t, nd_te, label="Newton Diag (test loss)")
    plt.xlabel("Time (s)"); plt.ylabel("Test Loss")
    plt.legend(); plt.title("Test Loss vs Time")
    plt.tight_layout(); plt.show()

    # Plot Accuracy vs Time
    plt.figure(figsize=(6,4))
    plt.plot(gd_t, gd_acc, label="GD (test acc)")
    plt.plot(nx_t, nx_acc, label="Newton Exact (test acc)")
    plt.plot(nd_t, nd_acc, label="Newton Diag (test acc)")
    plt.xlabel("Time (s)"); plt.ylabel("Test Accuracy")
    plt.legend(); plt.title("Test Accuracy vs Time")
    plt.tight_layout(); plt.show()

methods = ["GD", "Newton Exact", "Newton Diag"]
total_times = [gd_t[-1], nx_t[-1], nd_t[-1]]

plt.figure(figsize=(5,4))
plt.bar(methods, total_times, color=["blue","orange","green"])
plt.ylabel("Total Runtime (s)")
plt.title("Runtime Comparison")
plt.show()
```

The full script also includes gradient descent and diagonal Newton implementations, together with plotting code for test loss, accuracy, and runtime.

(c) Results

We trained the models with learning rate $\eta = 0.1$ and test size = 30%. The figures below compare the three optimization methods on the test set.

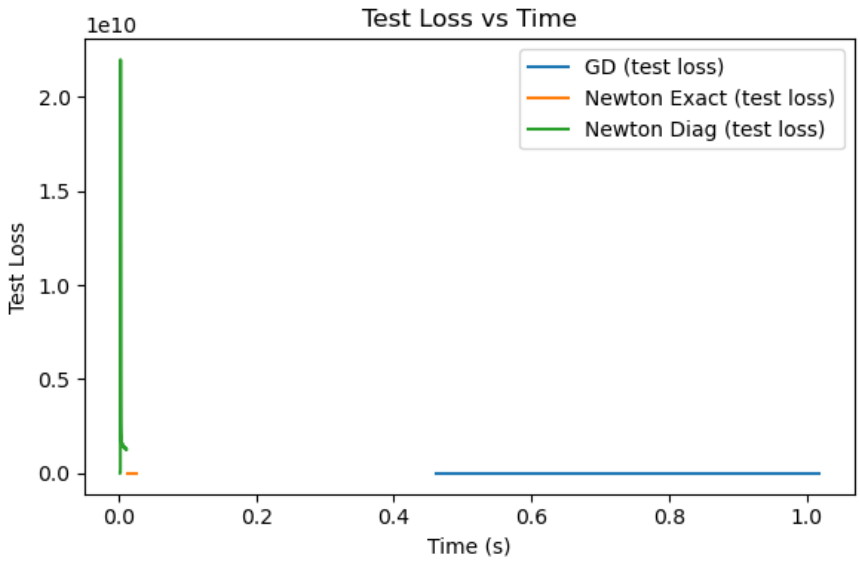

Figure 2: Test Loss vs Time for Gradient Descent, Newton (Exact Hessian), and Newton (Diagonal).

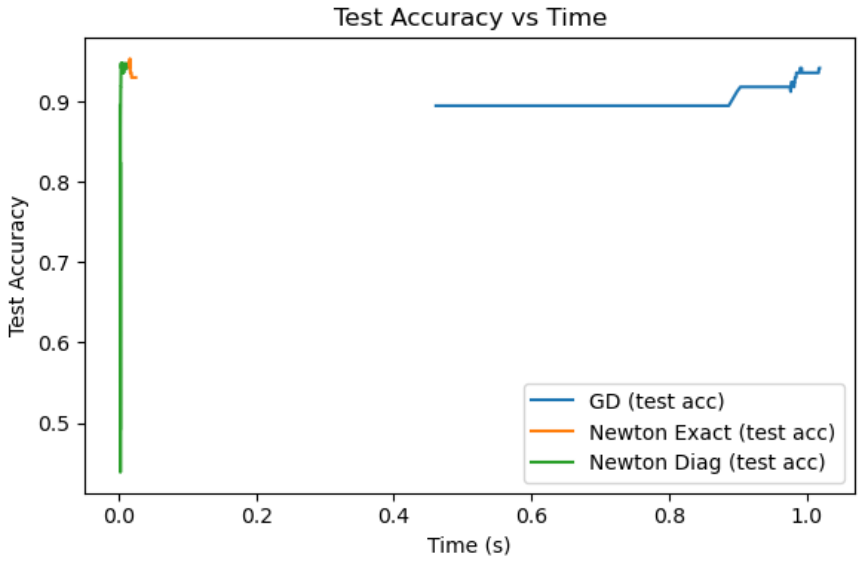

Figure 3: Test Accuracy vs Time for Gradient Descent, Newton (Exact Hessian), and Newton (Diagonal).

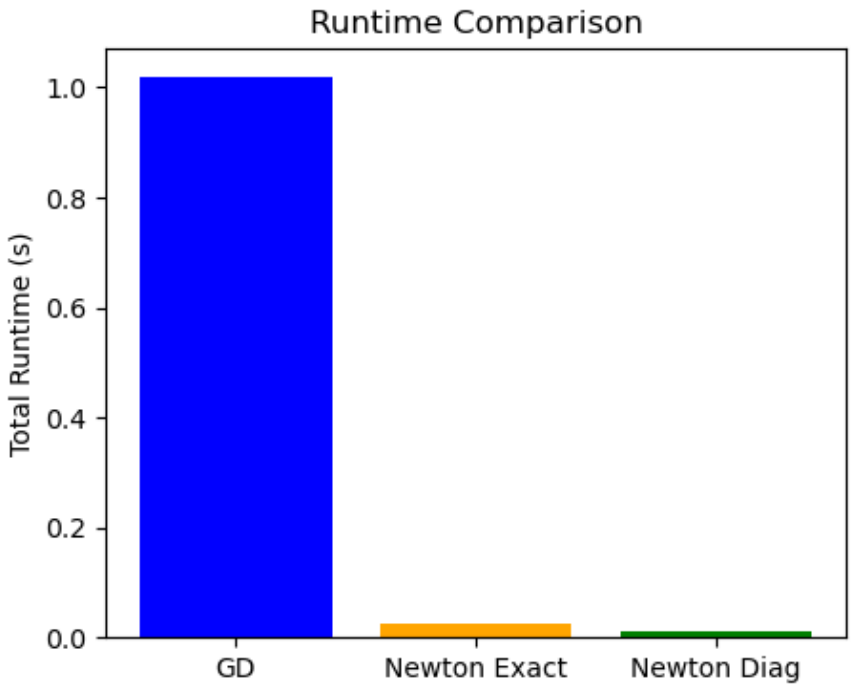

Figure 4: Total Runtime Comparison for the three methods.

(d) Discussion

- **Gradient Descent (GD)**: Each iteration is cheap, but convergence is slow, requiring many iterations to achieve good accuracy.
- **Newton's Method (Exact)**: Uses full second-order information. Each iteration is more expensive (computes and inverts Hessian), but the method converges in very few iterations, leading to the fastest convergence overall when d is not too large.
- **Newton's Method (Diagonal)**: Computationally as cheap as GD per iteration, and converges faster than GD since it uses approximate curvature information. However, the diagonal approximation can lead to less stable updates compared to the exact Newton method.

In summary, Newton's method with exact Hessian is the most efficient in this dataset, the diagonal approximation offers a reasonable compromise between speed and accuracy, while gradient descent is the slowest method.

LLM Usage Statement

I used ChatGPT 5 to suggest the structure of plots and analysis and write LaTex code.