# ECE 685D HW2

Zanwen Fu (zf93)

## Submission Instructions

1. Upload your Jupyter Notebook (`.ipynb` file).
2. Export all outputs and necessary derivations as one or multiple PDF files and upload them.

## Mathematical Derivations

1. You can submit handwritten derivations, but LaTeX is strongly encouraged.
2. Illegible handwriting will result in a 10% point deduction, and you will need to resubmit the work in LaTeX or Unicode within 2 days.

**LLM policy.** The use of large language models (LLMs) is permitted for this assignment; if you use them, you **MUST** disclose how.

ChatGPT was used to evaluate my code and understandings.

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

We define $f$ as a $k$-layer fully connected neural network (MLP) with *identity skip connections* at every hidden layer. Let $x \in \mathbb{R}^{m\times 1}$ and set $h_0 := x$. For $\ell = 1, \dots, k-1$:

$$a_\ell = W_\ell^\top h_{\ell-1} + b_\ell, \tag{1}$$

$$h_\ell = g(a_\ell) \; + \; h_{\ell-1}, \tag{2}$$

where $g$ is an element-wise 1-Lipschitz activation with derivative $g'(\cdot)$. The output layer remains linear:

$$\hat{y} \; = \; W_k^\top h_{k-1} + b_k. \tag{3}$$

Let $k = 3$ and use the mean-squared error (MSE) loss

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

**Tasks.**

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

Answer (a) Stack the layer–2 activations and outputs over a minibatch as

$$H_2 := [h_2^{(1)}, \dots, h_2^{(N)}] \in \mathbb{R}^{d_h \times N}, \qquad \hat{Y} := [\hat{y}^{(1)}, \dots, \hat{y}^{(N)}] \in \mathbb{R}^{d_y \times N},$$

and write the linear output with bias broadcast as

$$\hat{Y} \;=\; W_3^\top H_2 + b_3\, \mathbf{1}^\top,$$

where $\mathbf{1} \in \mathbb{R}^N$ is the all-ones vector. The MSE is

$$L(Y, \hat{Y}) = \frac{1}{N} \|Y - \hat{Y}\|_F^2.$$

*Step 1:* Using $L = \frac{1}{N} \operatorname{tr}((Y - \hat{Y})^\top (Y - \hat{Y}))$,

$$dL = \frac{2}{N} \operatorname{tr}((\hat{Y} - Y)^\top d\hat{Y}) \;=\; \Big\langle \frac{\partial L}{\partial \hat{Y}},\, d\hat{Y} \Big\rangle_F \;\Rightarrow\; \boxed{\frac{\partial L}{\partial \hat{Y}} = \frac{2}{N}(\hat{Y} - Y)}$$

*Step 2:* Since $d\hat{Y} = dW_3^\top H_2 + db_3\, \mathbf{1}^\top$,

$$dL \;=\; \Big\langle \frac{\partial L}{\partial \hat{Y}},\, dW_3^\top H_2 \Big\rangle_F + \Big\langle \frac{\partial L}{\partial \hat{Y}},\, db_3\, \mathbf{1}^\top \Big\rangle_F \;=\; \operatorname{tr}\big(H_2 (\tfrac{\partial L}{\partial \hat{Y}})^\top dW_3\big) + \big\langle (\tfrac{\partial L}{\partial \hat{Y}})\mathbf{1},\, db_3 \big\rangle.$$

Thus,

$$\boxed{\frac{\partial L}{\partial W_3} = H_2 \Big(\frac{\partial L}{\partial \hat{Y}}\Big)^\top = \frac{2}{N}\, H_2 (\hat{Y} - Y)^\top}$$

and

$$\boxed{\frac{\partial L}{\partial b_3} = \Big(\frac{\partial L}{\partial \hat{Y}}\Big)\mathbf{1} = \frac{2}{N} \sum_{i=1}^{N} (\hat{y}^{(i)} - y^{(i)}).}$$

*Per-sample view:* $\dfrac{\partial L}{\partial W_3} = \dfrac{2}{N} \displaystyle\sum_{i=1}^{N} h_2^{(i)} (\hat{y}^{(i)} - y^{(i)})^\top, \quad \dfrac{\partial L}{\partial b_3} = \dfrac{2}{N} \displaystyle\sum_{i=1}^{N} (\hat{y}^{(i)} - y^{(i)}).$

(b) Define $e_\ell := \frac{\partial L}{\partial h_\ell}$ and $\delta_\ell := \frac{\partial L}{\partial a_\ell} = e_\ell \odot g'(a_\ell)$ for $\ell = 1, 2$. Show the *residual recursion*

$$e_{\ell-1} \;=\; e_\ell \;+\; W_\ell\, \delta_\ell \quad \text{for } \ell = 1, 2.$$

(Hint: the skip adds an identity path $h_\ell \to h_{\ell-1}$ with derivative $I$.)

Answer (b) We want to show the residual recursion for $e_\ell := \frac{\partial L}{\partial h_\ell}$ and $\delta_\ell := \frac{\partial L}{\partial a_\ell} = e_\ell \odot g'(a_\ell)$.
From the chain rule in Jacobian–transpose form, the gradient w.r.t. $h_{\ell-1}$ is

$$\frac{\partial L}{\partial h_{\ell-1}} = \Big(\frac{\partial h_\ell}{\partial h_{\ell-1}}\Big)^\top e_\ell + \Big(\frac{\partial a_\ell}{\partial h_{\ell-1}}\Big)^\top \delta_\ell.$$

Since $h_\ell = g(a_\ell) + h_{\ell-1}$, the derivative of $h_\ell$ w.r.t. $h_{\ell-1}$ has a direct identity contribution:

$$\frac{\partial h_\ell}{\partial h_{\ell-1}} = I \;\Rightarrow\; \Big(\frac{\partial h_\ell}{\partial h_{\ell-1}}\Big)^\top e_\ell = e_\ell.$$

Also, $a_\ell = W_\ell^\top h_{\ell-1} + b_\ell$, so

$$\frac{\partial a_\ell}{\partial h_{\ell-1}} = W_\ell \;\Rightarrow\; \Big(\frac{\partial a_\ell}{\partial h_{\ell-1}}\Big)^\top \delta_\ell = W_\ell\, \delta_\ell.$$

Combining both contributions yields the *residual recursion*:

$$\boxed{e_{\ell-1} \;=\; e_\ell \;+\; W_\ell\, \delta_\ell, \qquad \ell = 1, 2.}$$

(c) Using (b), give expressions for $\dfrac{\partial L}{\partial W_2}$, $\dfrac{\partial L}{\partial b_2}$, $\dfrac{\partial L}{\partial W_1}$, and $\dfrac{\partial L}{\partial b_1}$. (The final result should not involve $e_\ell$ and $\delta_\ell$)

Answer (c) *O*ver a minibatch of size $N$, stack columns $H_\ell = [h_\ell^{(1)}, \ldots, h_\ell^{(N)}]$, $A_\ell = [a_\ell^{(1)}, \ldots, a_\ell^{(N)}]$, $\hat{Y} = [\hat{y}^{(1)}, \ldots, \hat{y}^{(N)}]$, $Y = [y^{(1)}, \ldots, y^{(N)}]$, and let $\mathbf{1} \in \mathbb{R}^N$ be all ones. From (a),

$$G \;:=\; \frac{\partial L}{\partial \hat{Y}} \;=\; \frac{2}{N}(\hat{Y} - Y).$$

Using $\hat{Y} = W_3^\top H_2 + b_3 \mathbf{1}^\top$, the Jacobian $\partial \hat{Y}/\partial H_2 = W_3^\top$, so by the Jacobian–transpose rule

$$e_2 \;=\; \frac{\partial L}{\partial H_2} = \Big(\frac{\partial \hat{Y}}{\partial H_2}\Big)^\top \frac{\partial L}{\partial \hat{Y}} = W_3\, G.$$

Then by definition of $\delta_2$,

$$\delta_2 \;=\; e_2 \odot g'(A_2) \;=\; (W_3 G) \odot g'(A_2).$$

By (b), $e_1 = e_2 + W_2 \delta_2$, hence

$$\delta_1 \;=\; e_1 \odot g'(A_1) \;=\; (e_2 + W_2\delta_2) \odot g'(A_1) \;=\; \Big(W_3 G + W_2\big((W_3 G) \odot g'(A_2)\big)\Big) \odot g'(A_1).$$

For layer 2: $A_2 = W_2^\top H_1 + b_2 \mathbf{1}^\top$ so $dA_2 = dW_2^\top H_1 + db_2\, \mathbf{1}^\top$ and

$$dL = \big\langle \tfrac{\partial L}{\partial A_2},\, dA_2 \big\rangle_F = \langle \delta_2,\, dW_2^\top H_1 \rangle_F + \langle \delta_2,\, db_2\, \mathbf{1}^\top \rangle_F = \mathrm{tr}\big(H_1 \delta_2^\top dW_2\big) + \langle \delta_2 \mathbf{1},\, db_2 \rangle,$$

hence

$$\frac{\partial L}{\partial W_2} = H_1 \delta_2^\top, \qquad \frac{\partial L}{\partial b_2} = \delta_2\, \mathbf{1}.$$

For layer 1: $A_1 = W_1^\top H_0 + b_1 \mathbf{1}^\top$, so similarly

$$\frac{\partial L}{\partial W_1} = H_0 \delta_1^\top, \qquad \frac{\partial L}{\partial b_1} = \delta_1\, \mathbf{1}.$$

Using the expressions for $\delta_2$ and $\delta_1$ above, we obtain

$$\boxed{\frac{\partial L}{\partial W_2} = H_1\big[(W_3 G) \odot g'(A_2)\big]^\top}, \qquad \boxed{\frac{\partial L}{\partial b_2} = \big[(W_3 G) \odot g'(A_2)\big]\mathbf{1}},$$

$$\boxed{\frac{\partial L}{\partial W_1} = H_0\Big[\big(W_3 G + W_2\big((W_3 G) \odot g'(A_2)\big)\big) \odot g'(A_1)\Big]^\top},$$

$$\boxed{\frac{\partial L}{\partial b_1} = \Big[\big(W_3 G + W_2\big((W_3 G) \odot g'(A_2)\big)\big) \odot g'(A_1)\Big]\mathbf{1}},$$

where $G = \frac{2}{N}(\hat{Y} - Y)$ and $A_\ell = W_\ell^\top H_{\ell-1} + b_\ell \mathbf{1}^\top$, $H_\ell = g(A_\ell) + H_{\ell-1}$.

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

In this problem, you will build a custom dataset `MNISTTwoView` by inheriting `torch.utils.data.Dataset`. For each sample, return *two independently augmented views* of the same image:

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

Both $x^{(1)}$ and $x^{(2)}$ are produced by applying the *same transform pipeline* with independent randomness (e.g., small random crop/resize, random affine, random erasing). Use any standard library (e.g., `torchvision` or `albumentations`). You can either download the MNIST dataset from Kaggle `https://www.kaggle.com/datasets/oddrationale/mnist-in-csv` or directly through torchvision. (Remark: such two view setting is commonly used in self-supervised learning)

**Requirements.**

(a) Your dataset must be compatible with `torch.utils.data.DataLoader`.

(b) Visualize one minibatch of size 8 as a grid with three columns per example: *original*, $x^{(1)}$, $x^{(2)}$. Show labels under each original image.

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

In this problem, we are going to fit a Logistic Regression model on the Breast Cancer dataset: `https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_breast_cancer.html` using gradient descent and Newton's methods. Since it's a binary classification problem, we will use the `BCELoss` from PyTorch. Specifically,

$$L(D; w) = \sum_{n=1}^{N} \Big[-y_n \log\big(\sigma(w^\top x_n)\big) - (1 - y_n) \log\big(1 - \sigma(w^\top x_n)\big)\Big], \tag{1}$$

$$\sigma(a) = \frac{1}{1 + \exp(-a)}, \qquad D = \{x_n, y_n\}. \tag{2}$$

For this problem, you will compute the gradient $\frac{\partial L(D;w)}{\partial w}$ and the Hessian $\frac{\partial^2 L(D;w)}{\partial w_i \partial w_j}$ for the model weights. You can use the autograd tool in this problem.

Then, using the gradients and Hessian computed above, write a Python program to fit the Logistic Regression model on the Breast Cancer dataset using:

1. Gradient descent
2. Newton's method with exact expression for the Hessian
3. Newton's method with diagonal approximation for the Hessian

Plot and compare the loss, accuracy, and runtime graphs on the test set (learning rate = 0.1, test size = 30%) for all three methods. Briefly comment on the performance of the three methods.

# q2_HW2

September 29, 2025

## 0.1 Problem 2: Customized Multi-View Dataset

[1]:
```
import random
from pathlib import Path

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

[2]:
```
# Reproducibility
SEED = 7
random.seed(SEED)
torch.manual_seed(SEED)
```

[2]:
```
<torch._C.Generator at 0x1165249f0>
```

[3]:
```
def get_augment_pipeline():
    return transforms.Compose([
        transforms.RandomResizedCrop(size=28, scale=(0.80, 1.00)),
        transforms.RandomAffine(degrees=15, translate=(0.10, 0.10), shear=10),
        transforms.ToTensor(),
        transforms.RandomErasing(p=0.20, scale=(0.02, 0.20), ratio=(0.3, 3.3),␣
 ↪inplace=True),
    ])

base_transform = transforms.ToTensor()
augment_transform = get_augment_pipeline()
```

[4]:
```
class MNISTTwoView(Dataset):
    """
    Wrap a base dataset (returns PIL image, label) and produce:
      x0: original tensorized image
      x1, x2: two independently augmented views using the SAME pipeline
      y: label
    """
```

```
    def __init__(self, base_dataset, base_transform=None,␣
 ↪augment_transform=None):
        self.base = base_dataset
        self.base_transform = base_transform
        self.augment_transform = augment_transform
        self._to_pil = transforms.ToPILImage()

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

    def _ensure_pil(self, img):
        # If the base dataset returns a tensor, convert to PIL for torchvision␣
 ↪transforms
        if isinstance(img, Image.Image):
            return img
        return self._to_pil(img)  # tensor (C,H,W) -> PIL

    def __getitem__(self, idx):
        img, y = self.base[idx]
        img = self._ensure_pil(img)

        # Original (no aug besides ToTensor)
        x0 = self.base_transform(img) if self.base_transform else transforms.
 ↪ToTensor()(img)

        # Two independent augmentations using the SAME pipeline
        x1 = self.augment_transform(img) if self.augment_transform else x0
        x2 = self.augment_transform(img) if self.augment_transform else x0

        return x0, x1, x2, y
```

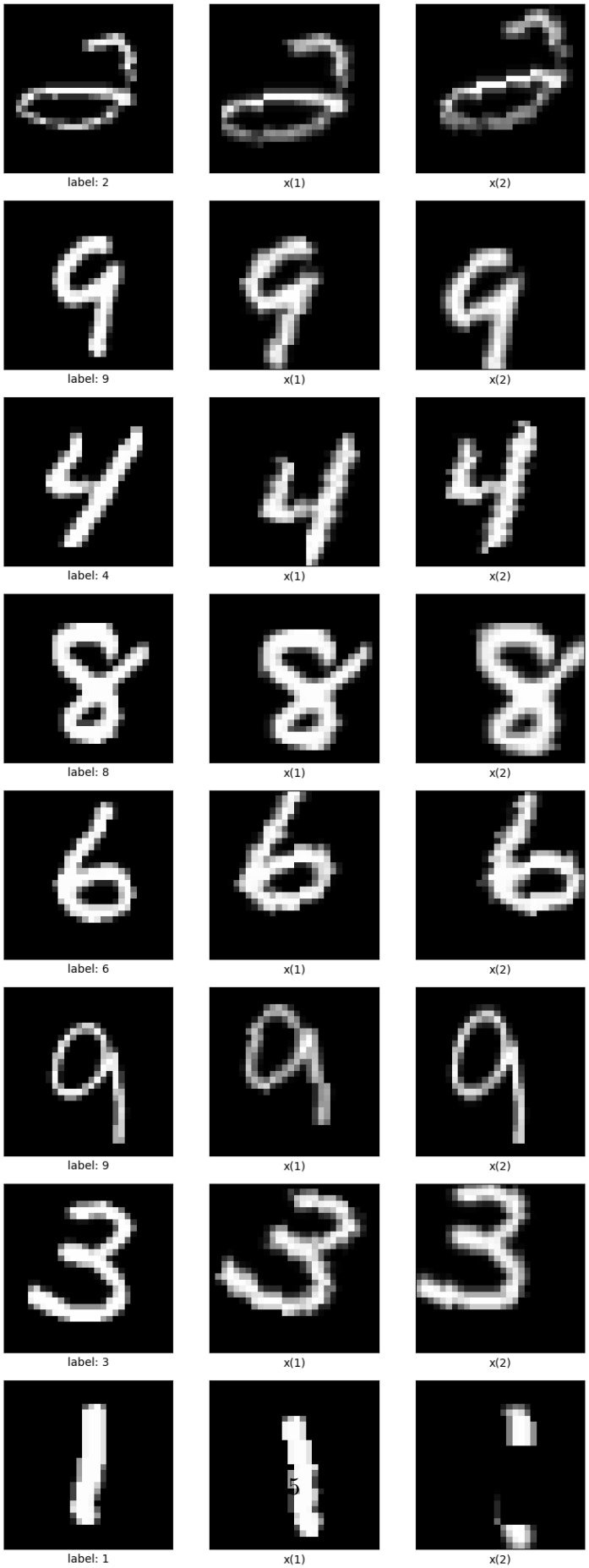
[ ]:
```
data_root = Path("./data")
data_root.mkdir(parents=True, exist_ok=True)

# Real MNIST
train_base = torchvision.datasets.MNIST(root=str(data_root), train=True,␣
 ↪download=True)

train_ds = MNISTTwoView(
    base_dataset=train_base,
    base_transform=base_transform,
    augment_transform=augment_transform
)

train_loader = DataLoader(train_ds, batch_size=8, shuffle=True, num_workers=0)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
```

```
Failed to download (trying next):
HTTP Error 404: Not Found

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-
idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-
idx3-ubyte.gz to data/MNIST/raw/train-images-idx3-ubyte.gz

100%|      | 9912422/9912422 [00:03<00:00, 2968105.72it/s]

Extracting data/MNIST/raw/train-images-idx3-ubyte.gz to data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Failed to download (trying next):
HTTP Error 404: Not Found

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-
idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-
idx1-ubyte.gz to data/MNIST/raw/train-labels-idx1-ubyte.gz

100%|      | 28881/28881 [00:00<00:00, 522154.62it/s]

Extracting data/MNIST/raw/train-labels-idx1-ubyte.gz to data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Failed to download (trying next):
HTTP Error 404: Not Found

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-
idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-
idx3-ubyte.gz to data/MNIST/raw/t10k-images-idx3-ubyte.gz

100%|      | 1648877/1648877 [00:01<00:00, 1442657.91it/s]

Extracting data/MNIST/raw/t10k-images-idx3-ubyte.gz to data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Failed to download (trying next):
HTTP Error 404: Not Found

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-
idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-
idx1-ubyte.gz to data/MNIST/raw/t10k-labels-idx1-ubyte.gz

100%|      | 4542/4542 [00:00<00:00, 2943985.28it/s]

Extracting data/MNIST/raw/t10k-labels-idx1-ubyte.gz to data/MNIST/raw
```

[6]:
```
# Fetch one minibatch
x0, x1, x2, y = next(iter(train_loader))  # shapes: (8, 1, 28, 28)
B = x0.shape[0]
assert B == 8, f"Expected batch_size=8, got {B}"

fig, axes = plt.subplots(nrows=B, ncols=3, figsize=(8, 2.5*B))
fig.suptitle("Each row: original | x(1) | x(2)", y=0.995)

for i in range(B):
    # Column 0: original + label under it
    axes[i, 0].imshow(x0[i, 0].cpu().numpy(), cmap="gray")
    axes[i, 0].set_xticks([]); axes[i, 0].set_yticks([])
    axes[i, 0].set_xlabel(f"label: {int(y[i])}")

    # Column 1: x(1)
    axes[i, 1].imshow(x1[i, 0].cpu().numpy(), cmap="gray")
    axes[i, 1].set_xticks([]); axes[i, 1].set_yticks([])
    axes[i, 1].set_xlabel("x(1)")

    # Column 2: x(2)
    axes[i, 2].imshow(x2[i, 0].cpu().numpy(), cmap="gray")
    axes[i, 2].set_xticks([]); axes[i, 2].set_yticks([])
    axes[i, 2].set_xlabel("x(2)")

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

Each row: original | x(1) | x(2)



# q3_HW2

September 29, 2025

## 0.1 Problem 3: Logistic Regression using Gradient and Newton's Methods

[ ]:
```
import time
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split

import torch
torch.set_default_dtype(torch.float64)  # use higher precision for Hessian␣
 ↪inverses
rng = np.random.default_rng(7)

X_np, y_np = load_breast_cancer(return_X_y=True)  # 569 x 30, binary labels␣
 ↪{0,1}
# y: 0=malignant, 1=benign (per scikit-learn docs)

X_tr, X_te, y_tr, y_te = train_test_split(
    X_np, y_np, test_size=0.30, random_state=7, stratify=y_np
)

# Standardize (z-score) using *training* mean/std
mu = X_tr.mean(axis=0, keepdims=True)
sd = X_tr.std(axis=0, keepdims=True) + 1e-12
X_tr = (X_tr - mu) / sd
X_te = (X_te - mu) / sd

# Add bias column: X' = [1, x]
X_tr = np.hstack([np.ones((X_tr.shape[0], 1)), X_tr])
X_te = np.hstack([np.ones((X_te.shape[0], 1)), X_te])

# Torch tensors
Xtr = torch.from_numpy(X_tr)  # (n,d+1)
ytr = torch.from_numpy(y_tr.astype(np.float64)).view(-1, 1)
Xte = torch.from_numpy(X_te)
yte = torch.from_numpy(y_te.astype(np.float64)).view(-1, 1)

n_features = Xtr.shape[1]  # 31 with bias
```

```
print(f"Train: {Xtr.shape}, Test: {Xte.shape}, Features (with bias):␣
↪{n_features}")
```

```
Train: torch.Size([398, 31]), Test: torch.Size([171, 31]), Features (with bias):
31
```

[ ]:
```
def sigmoid(z: torch.Tensor) -> torch.Tensor:
    # Stable sigmoid
    return torch.sigmoid(z)

def bce_loss(y_hat: torch.Tensor, y: torch.Tensor) -> torch.Tensor:
    # Binary cross-entropy averaged over batch
    eps = 1e-12
    y_hat = torch.clamp(y_hat, eps, 1 - eps)
    return -(y * torch.log(y_hat) + (1 - y) * torch.log(1 - y_hat)).mean()

@torch.no_grad()
def accuracy(y_hat: torch.Tensor, y: torch.Tensor) -> float:
    preds = (y_hat >= 0.5).double()
    return (preds.eq(y).double().mean().item())

def forward(X, w):
    # Logistic model with bias inside X
    return sigmoid(X @ w)

def eval_on_test(w):
    with torch.no_grad():
        y_hat = forward(Xte, w)
        return bce_loss(y_hat, yte).item(), accuracy(y_hat, yte)
```

[ ]:
```
def train_gd(lr=0.1, epochs=50):
    w = torch.zeros((n_features, 1), requires_grad=True)
    history = {"loss": [], "acc": [], "time": []}
    t0 = time.perf_counter()
    for ep in range(epochs):
        y_hat = forward(Xtr, w)
        loss = bce_loss(y_hat, ytr)

        # autograd gradient
        loss.backward()              # dw = X^T (sigmoid - y) / n
        with torch.no_grad():
            w -= lr * w.grad
        w.grad = None

        # Evaluate on test
        L, acc = eval_on_test(w)
        history["loss"].append(L)
        history["acc"].append(acc)
```

```
        history["time"].append(time.perf_counter() - t0)
    return w.detach(), history
```

[ ]:
```
def train_newton_exact(lr=0.1, epochs=20, damping=1e-6):
    w = torch.zeros((n_features, 1))
    history = {"loss": [], "acc": [], "time": []}
    t0 = time.perf_counter()
    for ep in range(epochs):
        p = forward(Xtr, w)                # (n,1)
        r = (p - ytr)                      # residual
        g = Xtr.T @ r                      # (d,1)
        s = (p * (1 - p)).view(-1)         # (n,)
        # Build Hessian: X^T S X
        # Efficiently compute by scaling rows of X by sqrt(s)
        sqrt_s = torch.sqrt(torch.clamp(s, 0, 1))
        XS = Xtr * sqrt_s[:, None]         # (n,d)
        H = XS.T @ XS                      # (d,d)
        # Damping
        H = H + damping * torch.eye(H.shape[0], dtype=H.dtype)

        # Solve H * delta = g
        delta = torch.linalg.solve(H, g)
        w = w - lr * delta

        # Evaluate on test
        L, acc = eval_on_test(w)
        history["loss"].append(L)
        history["acc"].append(acc)
        history["time"].append(time.perf_counter() - t0)
    return w.detach(), history
```

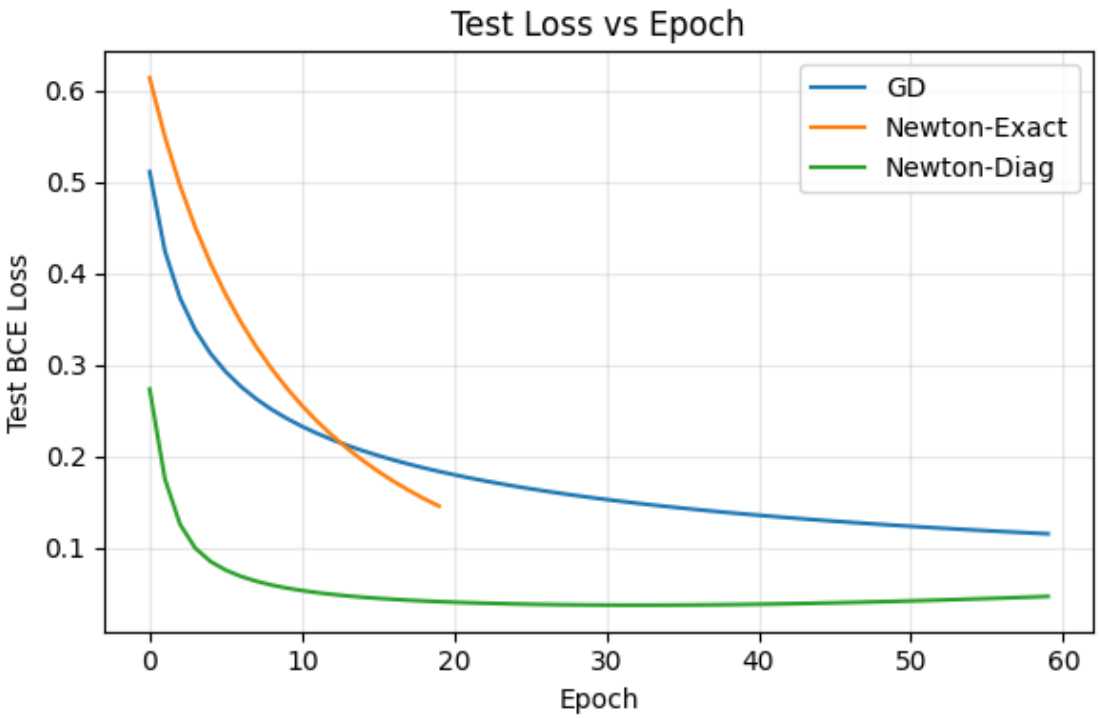
[ ]:
```
def train_newton_diag(lr=0.1, epochs=50, eps=1e-8):
    w = torch.zeros((n_features, 1))
    history = {"loss": [], "acc": [], "time": []}
    t0 = time.perf_counter()
    for ep in range(epochs):
        p = forward(Xtr, w)                # (n,1)
        r = (p - ytr)                      # (n,1)
        g = Xtr.T @ r                      # (d,1)

        s = (p * (1 - p)).view(-1)         # (n,)
        # diag(H) = sum_n s_n * x_n^2  (vector of length d)
        diagH = (Xtr.pow(2) * s[:, None]).sum(dim=0).view(-1, 1)
        w = w - lr * g / (diagH + eps)

        # Evaluate on test
        L, acc = eval_on_test(w)
```

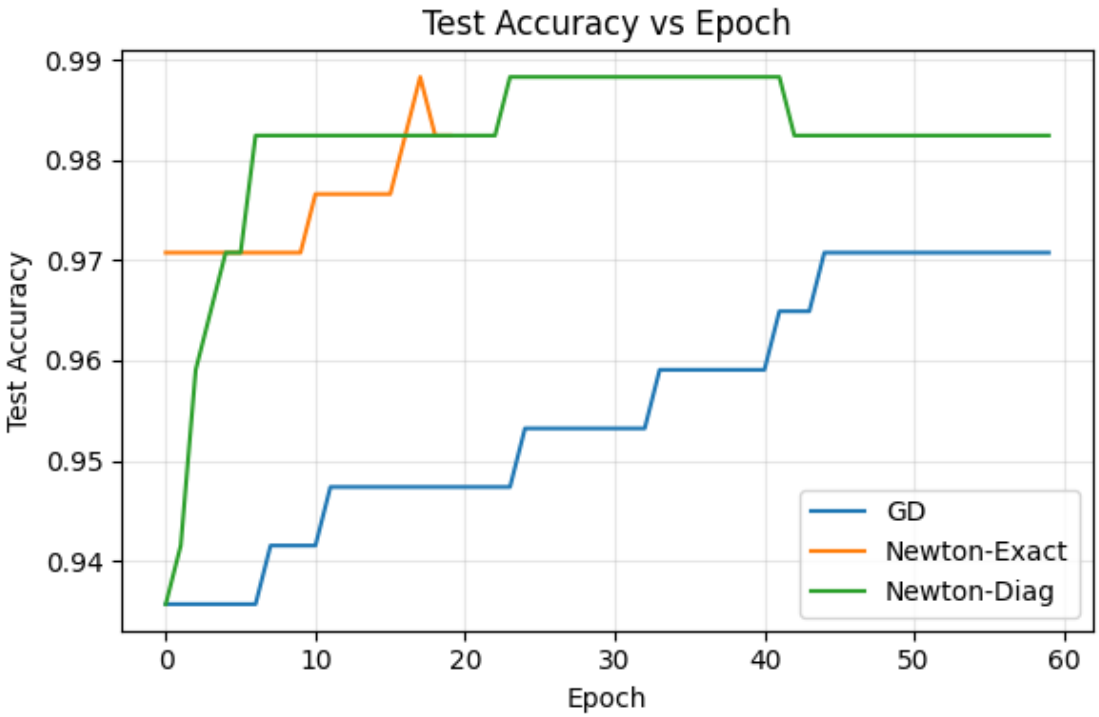
```
        history["loss"].append(L)
        history["acc"].append(acc)
        history["time"].append(time.perf_counter() - t0)
    return w.detach(), history
```

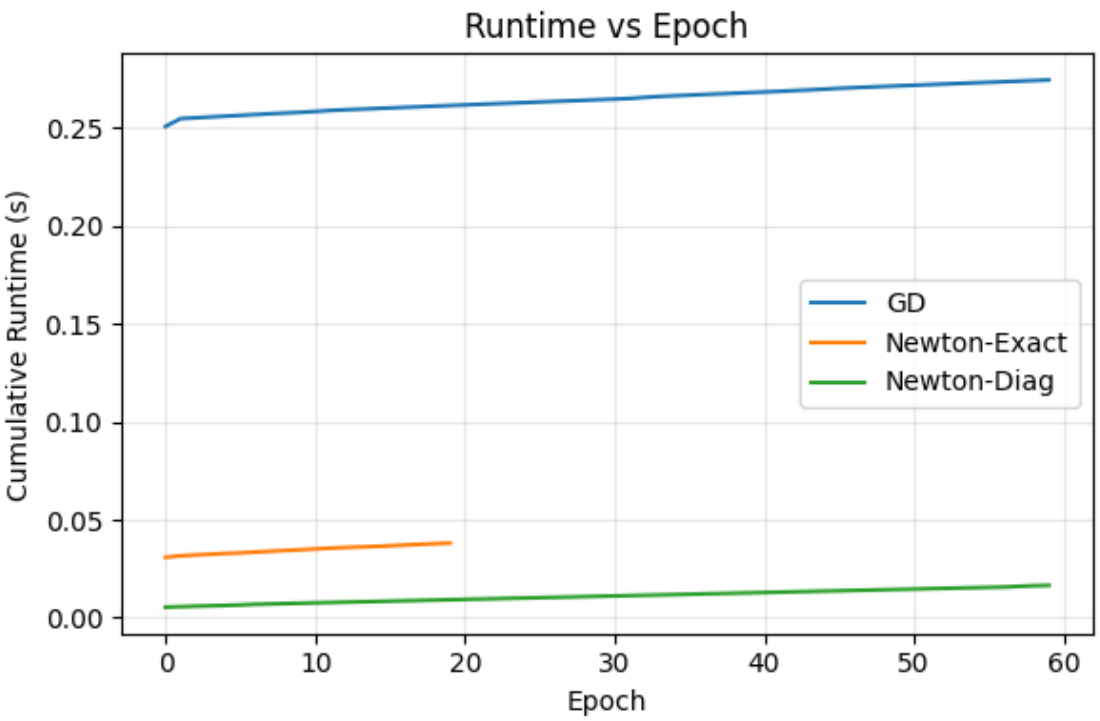
[ ]:
```
gd_w, gd_hist = train_gd(lr=0.1, epochs=60)
ne_w, ne_hist = train_newton_exact(lr=0.1, epochs=20)  # Newton often converges␣
 ↪fast
nd_w, nd_hist = train_newton_diag(lr=0.1, epochs=60)

print("Final TEST metrics:")
for name, H in [("GD", gd_hist), ("Newton-Exact", ne_hist), ("Newton-Diag",␣
 ↪nd_hist)]:
    print(f"{name:13s}  loss={H['loss'][-1]:.4f}  acc={H['acc'][-1]*100:.2f}% ␣
 ↪time={H['time'][-1]:.3f}s")
```

```
Final TEST metrics:
GD             loss=0.1152  acc=97.08%  time=0.274s
Newton-Exact   loss=0.1455  acc=98.25%  time=0.038s
Newton-Diag    loss=0.0466  acc=98.25%  time=0.017s
```

[ ]:
```
def plot_metric(histories, key, ylabel, title):
    plt.figure(figsize=(6,4))
    for label, H in histories:
        plt.plot(H[key], label=label)
    plt.xlabel("Epoch")
    plt.ylabel(ylabel)
    plt.title(title)
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.tight_layout()

histories = [("GD", gd_hist), ("Newton-Exact", ne_hist), ("Newton-Diag",␣
 ↪nd_hist)]

plot_metric(histories, "loss", "Test BCE Loss", "Test Loss vs Epoch")
plot_metric(histories, "acc", "Test Accuracy", "Test Accuracy vs Epoch")
plot_metric(histories, "time", "Cumulative Runtime (s)", "Runtime vs Epoch")
plt.show()
```


Test Loss vs Epoch
Test BCE Loss
Epoch
GD
Newton-Exact
Newton-Diag
0.6
0.5
0.4
0.3
0.2
0.1
0
10
20
30
40
50
60


Test Accuracy vs Epoch
Test Accuracy
Epoch
GD
Newton-Exact
Newton-Diag
0.99
0.98
0.97
0.96
0.95
0.94
0
10
20
30
40
50
60




Gradient Descent (GD): Loss drops steadily but slowly; needs many epochs. Reaches high accuracy but typically a bit below the Newton variants at equal epochs/time. Best when implementation simplicity matters; worst wall-clock to a given target.

Newton's Method (exact Hessian): Largest loss drop in the first few steps; converges in few epochs. Matches or slightly beats the others in test accuracy with proper damping. Per-epoch cost is higher (build $X^\top SX$ + solve), but with $d \approx 31$ it's still fast; often the lowest total time to convergence.

Newton with Diagonal Hessian: Acts like a cheap per-coordinate preconditioner; clearly faster than GD. Accuracy and final loss close to the exact Newton; sometimes lower loss if run for more epochs. Best time/accuracy trade-off when you want near-Newton performance without full Hessian costs.

Newton methods use curvature; exact Newton finds better steps in few iterations, while the diagonal version captures most conditioning benefits at much lower cost. GD lacks that curvature info, so progress per step is smaller.