New York University

Tandon School of Engineering

Department of Computer Science & Engineering

Deep Learning

CS-GY 6953 / ECE-GY 7123

Spring 26

Professor Gustavo Sandoval

Wednesday 11:00 am – 1:30

Deep Learning

https://github.com/nidhish1/DL_HW/tree/master

Nidhish Gautam

Net ID: ng3483

Problem 1: Warmup — Softmax + Matrix Gradients

Part 1: Temperature-scaled softmax Jacobian (6 pts)

Let $z \in \mathbb{R}^n$ and $\tau > 0$. Define

$$y_i = \mathrm{softmax}_\tau(z)_i = \frac{e^{z_i/\tau}}{\sum_{j=1}^n e^{z_j/\tau}}.$$

We want the Jacobian J with $J_{ik} = \frac{\partial y_i}{\partial z_k}$. Write $S = \sum_{j=1}^n e^{z_j/\tau}$ so that $y_i = e^{z_i/\tau}/S$. Then

$$\frac{\partial y_i}{\partial z_k} = \frac{1}{S}\frac{\partial}{\partial z_k}\big(e^{z_i/\tau}\big) \; - \; \frac{e^{z_i/\tau}}{S^2}\frac{\partial S}{\partial z_k}.$$

We have $\frac{\partial}{\partial z_k}(e^{z_i/\tau}) = \frac{1}{\tau}e^{z_i/\tau}\,\mathbf{1}[i=k]$ and $\frac{\partial S}{\partial z_k} = \frac{1}{\tau}e^{z_k/\tau}$. Substituting and using $y_i = e^{z_i/\tau}/S$ and $y_k = e^{z_k/\tau}/S$,

$$\frac{\partial y_i}{\partial z_k} = \frac{1}{\tau}\,y_i\,\mathbf{1}[i=k] \; - \; \frac{1}{\tau}\,y_i\,y_k \; = \; \frac{1}{\tau}\big(y_i\,\mathbf{1}[i=k] - y_i y_k\big).$$

So the (i,k) entry of the Jacobian is $\frac{1}{\tau}(y_i\delta_{ik} - y_i y_k)$, which is exactly the (i,k) entry of $\frac{1}{\tau}\big(\mathrm{diag}(y) - yy^\top\big)$. Hence

$$\boxed{\frac{\partial y}{\partial z} = \frac{1}{\tau}\big(\mathrm{diag}(y) - yy^\top\big).}$$

Part 2: Gradient w.r.t. A and b in affine least-squares (6 pts)

Let $A \in \mathbb{R}^{d\times m}$, $b \in \mathbb{R}^d$, $x \in \mathbb{R}^m$, and $t \in \mathbb{R}^d$ (fixed). Define

$$f(A,b) = \frac{1}{2}\|Ax + b - t\|_2^2.$$

Write the residual as $r = Ax + b - t \in \mathbb{R}^d$. Then $f = \frac{1}{2}r^\top r$, so $\frac{\partial f}{\partial r} = r^\top$, i.e. $\nabla_r f = r$.

f depends on A only through $r = Ax + b - t$. For the (i,j) entry of A, we have $\frac{\partial r}{\partial A_{ij}} = x_j \cdot e_i$ (the ith standard basis vector), because $(Ax)_i = \sum_\ell A_{i\ell}x_\ell$, so $\frac{\partial (Ax)_i}{\partial A_{ij}} = x_j$. By the chain rule,

$$\frac{\partial f}{\partial A_{ij}} = (\nabla_r f)^\top \frac{\partial r}{\partial A_{ij}} = r^\top (x_j e_i) = r_i x_j.$$

So the gradient of f with respect to A has (i,j) entry $r_i x_j$, which is the (i,j) entry of $rx^\top$. Thus

$$\boxed{\nabla_A f = (Ax + b - t)\,x^\top.}$$

f depends on b through $r = Ax + b - t$, and $\frac{\partial r}{\partial b} = I_d$, so $\nabla_b f = \nabla_r f = r$. Hence

$$\boxed{\nabla_b f = Ax + b - t.}$$

Part 3: Softmax vs. sigmoid: when and why? (3 pts)

Softmax, not sigmoid: Multi-class classification with a single label per example (e.g. image classification into one of many classes, or next-token prediction over a vocabulary). The classes are mutually exclusive, so we model a single categorical distribution over classes. Softmax outputs a probability vector that sums to 1 and is the correct parameterization for that distribution; we typically use cross-entropy loss between the one-hot label and the softmax output. Sigmoid would give one probability per class that does not sum to 1 and does not represent a proper categorical distribution, so it is not appropriate here.

Sigmoid, not softmax: Multi-label classification (e.g. tagging documents with multiple topics, or detecting multiple objects in an image). Each label is a binary outcome (present or not), and labels are not mutually exclusive. We model each label independently with a Bernoulli, so the right output is one probability per label via sigmoid. Cross-entropy (or binary cross-entropy) is applied per label. Softmax would force the outputs to sum to 1 and imply mutual exclusivity, which is wrong when several labels can be on at once.

In short: softmax + cross-entropy for a single categorical (mutually exclusive) choice; sigmoid for independent binary labels. The choice of activation and loss follows from whether we assume one-of-K or independent binary labels.

Problem 2: Linear Regression with Huber Loss

Setup

Residuals: $r_i(w) = y_i - w^\top x_i$, with $w \in \mathbb{R}^d$, $x_i \in \mathbb{R}^d$, $y_i \in \mathbb{R}$.

Huber loss (with parameter $\delta > 0$):

$$\ell_\delta(r) = \begin{cases} \frac{1}{2}r^2 & |r| \le \delta, \\ \delta\left(|r| - \frac{1}{2}\delta\right) & |r| > \delta. \end{cases}$$

Objective:

$$L_\delta(w) = \sum_{i=1}^{n} \ell_\delta(r_i(w)).$$

Part 1: Gradient derivation (6 pts)

We need $\nabla_w L_\delta(w)$. The loss is a sum over samples, so the gradient is the sum of the gradients of $\ell_\delta(r_i(w))$. By the chain rule, the contribution from sample i is

$$\nabla_w \left[\ell_\delta(r_i(w))\right] = \frac{\mathrm{d}\ell_\delta}{\mathrm{d}r}(r_i) \cdot \nabla_w r_i(w).$$

The residual is $r_i(w) = y_i - w^\top x_i$, so

$$\nabla_w r_i(w) = -x_i.$$

The derivative of the Huber loss with respect to r is piecewise: for $|r| \le \delta$ we have $\ell_\delta(r) = \frac{1}{2}r^2$, so $\frac{\mathrm{d}\ell_\delta}{\mathrm{d}r} = r$; for $|r| > \delta$ we have $\ell_\delta(r) = \delta(|r| - \frac{1}{2}\delta)$, so $\frac{\mathrm{d}\ell_\delta}{\mathrm{d}r} = \delta \cdot \operatorname{sign}(r)$. Thus

$$\frac{\mathrm{d}\ell_\delta}{\mathrm{d}r}(r) = \begin{cases} r & |r| \le \delta, \\ \delta \operatorname{sign}(r) & |r| > \delta. \end{cases}$$

Substituting into the chain rule and summing over i,

$$\boxed{\nabla_w L_\delta(w) = -\sum_{i=1}^{n} \frac{\mathrm{d}\ell_\delta}{\mathrm{d}r}(r_i(w))\, x_i,}$$

with the piecewise rule above applied to each $r_i(w)$. In explicit piecewise form:

$$\nabla_w L_\delta(w) = -\sum_{i\colon |r_i(w)| \le \delta} r_i(w)\, x_i \;-\; \sum_{i\colon |r_i(w)| > \delta} \delta \operatorname{sign}(r_i(w))\, x_i.$$

Part 2: Optimal w: closed form or not? (4 pts)

Minimizing $L_\delta(w)$ does **not** admit a single closed-form solution like ordinary least squares (OLS).

In OLS we minimize $\frac{1}{2}\sum_i(y_i - w^\top x_i)^2$. Setting the gradient to zero gives the normal equations $X^\top Xw = X^\top y$, which are linear in w, so we get an explicit solution $w^* = (X^\top X)^{-1}X^\top y$ (when $X^\top X$ is invertible). The key is that the objective is quadratic in w, so $\nabla_w L$ is linear in w.

For Huber loss, the gradient we derived is

$$\nabla_w L_\delta(w) = -\sum_i \psi(r_i(w))\, x_i, \qquad \psi(r) = \begin{cases} r & |r| \le \delta, \\ \delta\,\operatorname{sign}(r) & |r| > \delta. \end{cases}$$

Setting $\nabla_w L_\delta = 0$ therefore gives

$$\sum_i \psi(y_i - w^\top x_i)\, x_i = 0.$$

The function ψ is nonlinear (piecewise linear with a kink). The residuals $r_i(w) = y_i - w^\top x_i$ depend on w, and which branch of ψ applies to each i depends on w. So we get a **nonlinear** system in w: we cannot rearrange this into a single linear system $Aw = b$ with a fixed matrix A. That is what breaks the normal-equations approach.

In practice we minimize $L_\delta(w)$ numerically. One standard approach is **iteratively reweighted least squares (IRLS)**: at each iteration we treat the current residuals as fixed, approximate the Huber objective by a weighted least-squares problem (with weights that depend on the current r_i), solve that for the next w, and repeat. Another standard approach is to use a general-purpose optimizer such as **gradient descent** or **L-BFGS** on $L_\delta(w)$, using the gradient we derived.

Part 3: Add L2 regularization (3 pts)

Define

$$\tilde{L}(w) = \sum_{i=1}^{n} \ell_\delta(r_i(w)) + \frac{\lambda}{2}\|w\|_2^2.$$

The gradient of $\frac{\lambda}{2}\|w\|_2^2$ is λw. So

$$\boxed{\nabla_w \tilde{L}(w) = -\sum_{i=1}^{n} \frac{\mathrm{d}\ell_\delta}{\mathrm{d}r}(r_i(w))\, x_i \; + \; \lambda w,}$$

with the same piecewise definition of $\frac{\mathrm{d}\ell_\delta}{\mathrm{d}r}$ as in Part 1.

How λ changes the solution: Larger λ penalizes large $\|w\|$, so the solution is shrunk toward zero and has smaller norm. That reduces overfitting in high-dimensional settings (fewer samples than dimensions or many correlated features) by favoring simpler models. Under outliers, the quadratic part of Huber already limits their influence; adding $\lambda\|w\|^2$ further stabilizes the solution by preventing the fit from being pulled too strongly by any subset of points, so the regularized minimizer tends to be more robust than an unregularized one when the data are noisy or contain outliers.

Part 4: ML pipeline reflection (2 pts)

In the 3-step recipe (define objective $\rightarrow$ optimize $\rightarrow$ evaluate), Huber changes both optimization and evaluation compared to squared loss.

Optimization: The objective is no longer quadratic in w, so we cannot solve for w in closed form. We rely on iterative methods (e.g. gradient descent or IRLS). The gradient is still straightforward to compute and is continuous (subgradient at $|r| = \delta$ can be defined consistently), so optimization is well behaved.

Evaluation robustness: Squared loss heavily penalizes large residuals, so a few outliers can dominate the loss and distort both the learned w and the reported loss value. Huber loss behaves like squared loss for small $|r|$ and like $|r|$ for large $|r|$, so outliers have bounded influence on the objective and on the fit. So (i) optimization requires an iterative procedure but remains tractable, and (ii) evaluation (and the fitted model) are more robust to outliers than with squared loss alone.

Problem 3: Adam vs. SGD with Momentum

Setup

Adam (scalar parameter w) at step t:

$$\begin{aligned} g_t &= \nabla \ell_t(w_t), \\ m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t, \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2, \\ \hat{m}_t &= \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}, \\ w_{t+1} &= w_t - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}. \end{aligned}$$

Momentum SGD:

$$u_t = \mu u_{t-1} + g_t, \qquad w_{t+1} = w_t - \alpha u_t.$$

Assume $m_0 = 0$, $v_0 = 0$.

Part 1 (3.1): First two Adam updates (6 pts)

We assume $m_0 = 0$, $v_0 = 0$, and keep everything in g_1, g_2, α, β_1, β_2, ε.

First update (after we see g_1)

By definition, the first moment is $m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$. For $t = 1$ with $m_0 = 0$ we get $m_1 = (1-\beta_1) g_1$. The second moment is $v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$; for $t = 1$ with $v_0 = 0$ we get $v_1 = (1-\beta_2) g_1^2$.

$$\boxed{m_1 = (1-\beta_1) g_1}, \qquad \boxed{v_1 = (1-\beta_2) g_1^2}.$$

Because we started at zero, the raw m_1 and v_1 are biased downward. The algorithm therefore divides them by $1-\beta_1^t$ and $1-\beta_2^t$ respectively. For $t = 1$ that is $1-\beta_1$ and $1-\beta_2$, which cancel the same factors in m_1 and v_1, so the bias-corrected moments are just the gradient and its square:

$$\boxed{\hat{m}_1 = g_1}, \qquad \boxed{\hat{v}_1 = g_1^2}.$$

The update rule $w_{t+1} = w_t - \alpha \hat{m}_t / (\sqrt{\hat{v}_t} + \varepsilon)$ then gives, after substituting $\hat{m}_1$ and $\hat{v}_1$:

$$\boxed{w_2 = w_1 - \alpha \frac{g_1}{\sqrt{g_1^2} + \varepsilon}.}$$

Second update (after we see g_2)

We now have m_1 and v_1. The recurrence for the first moment blends the previous m_1 with the new gradient g_2: $m_2 = \beta_1 m_1 + (1-\beta_1)g_2$. Substituting $m_1 = (1-\beta_1)g_1$ and factoring out $(1-\beta_1)$ gives a weighted combination of g_1 and g_2. The same logic for the second moment uses $v_2 = \beta_2 v_1 + (1-\beta_2)g_2^2$ and $v_1 = (1-\beta_2)g_1^2$:

$$\boxed{m_2 = (1-\beta_1)(\beta_1 g_1 + g_2)}, \qquad \boxed{v_2 = (1-\beta_2)(\beta_2 g_1^2 + g_2^2)}.$$

Again we bias-correct by dividing by $1-\beta_1^t$ and $1-\beta_2^t$; for $t=2$ that is $1-\beta_1^2$ and $1-\beta_2^2$. No cancellation this time, so we keep the fractions:

$$\boxed{\hat{m}_2 = \frac{(1-\beta_1)(\beta_1 g_1 + g_2)}{1-\beta_1^2}}, \qquad \boxed{\hat{v}_2 = \frac{(1-\beta_2)(\beta_2 g_1^2 + g_2^2)}{1-\beta_2^2}}.$$

Applying the same update rule with these corrected moments yields:

$$\boxed{w_3 = w_2 - \alpha\frac{\hat{m}_2}{\sqrt{\hat{v}_2}+\varepsilon}.}$$

Part 2: Compare to momentum SGD (5 pts)

1. **Effective step size and gradient magnitude:**

   We want to see how the *magnitude* of the parameter update depends on the scale of the gradients. So we look at the update formulas and ask: if gradients were uniformly larger or smaller, how would the step size change?

   In momentum SGD the update is

   $$\Delta w_t = -\alpha u_t, \qquad u_t = \mu u_{t-1} + g_t.$$

   So the step magnitude is

   $$|\Delta w_t| = \alpha |u_t|.$$

   Since u_t is an EMA of the g_τ, we have $u_t \propto$ (scale of gradients) (modulo the decay μ). So doubling all gradient magnitudes roughly doubles $|u_t|$ and hence doubles the step: **effective step size scales directly with gradient magnitude**; bigger gradients $\Rightarrow$ bigger steps.

   In Adam the update is

   $$\Delta w_t = -\alpha\frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\varepsilon}.$$

   Here $\hat{m}_t$ is a bias-corrected first moment (EMA of g_t) and $\hat{v}_t$ is a bias-corrected second moment (EMA of g_t^2), so $\sqrt{\hat{v}_t}$ approximates the *root-mean-square* of recent gradients. Thus the step has the form $\alpha \cdot (\text{direction from } \hat{m}_t)/(\text{RMS scale} + \varepsilon)$: the numerator and the scale in the denominator both grow with gradient magnitude, so they partially cancel. The *effective step magnitude* is

   $$\alpha\frac{|\hat{m}_t|}{\sqrt{\hat{v}_t}+\varepsilon}.$$

   If we scale all g_t by c, then $\hat{m}_t \propto c$ and $\hat{v}_t \propto c^2$, so $|\hat{m}_t|/\sqrt{\hat{v}_t}$ is scale-invariant (see (b)); hence the step size is **much less sensitive** to raw gradient magnitude. Moreover each parameter has its own $\hat{v}_t$, so adaptation is **per-parameter**: a coordinate with consistently large $|g|$ gets a larger $\hat{v}$ and thus a smaller effective step; one with small or rare gradients gets a smaller $\hat{v}$ and a relatively larger step. So Adam reduces dependence on gradient magnitude via **RMS normalization**.

2. **Scaling gradients by a constant c:**

   We ask: if every gradient g_t is replaced by cg_t, how do the two algorithms' updates change? This tells us whether the optimizer is sensitive to the global scale of the loss/gradients (e.g. loss scaling or batch size).

   In momentum SGD, $u_t = \mu u_{t-1} + g_t$ is linear in g_t, so under $g_t \mapsto cg_t$ we get
   $$u_t \mapsto cu_t.$$
   The step is $\Delta w_t = -\alpha u_t$, so the step scales by c: **momentum SGD is not scale-invariant**; changing gradient scale changes step size proportionally.

   In Adam, $\hat{m}_t$ is linear in the g_τ (bias-corrected EMA of g_t), so $\hat{m}_t \mapsto c\hat{m}_t$. The second moment v_t is an EMA of g_t^2, so $g_t^2 \mapsto c^2 g_t^2$ implies $\hat{v}_t \mapsto c^2\hat{v}_t$, hence
   $$\sqrt{\hat{v}_t} \mapsto |c|\sqrt{\hat{v}_t}.$$
   The ratio in the update is $\hat{m}_t/(\sqrt{\hat{v}_t}+\varepsilon)$. So under scaling: numerator $\mapsto c\hat{m}_t$, denominator $\mapsto |c|\sqrt{\hat{v}_t}+\varepsilon$. For $c > 0$,
   $$\frac{\hat{m}_t}{\sqrt{\hat{v}_t}} \mapsto \frac{c\hat{m}_t}{c\sqrt{\hat{v}_t}} = \frac{\hat{m}_t}{\sqrt{\hat{v}_t}}:$$
   magnitude of the ratio is invariant. For $c < 0$, the direction of $\hat{m}_t$ flips (we move the opposite way), which is correct. So when $\sqrt{\hat{v}_t}$ dominates the denominator, Adam is **approximately scale-invariant**. The caveat: when $\varepsilon > 0$ and $\hat{v}_t$ is small (e.g. early steps or rarely updated parameters), $\sqrt{\hat{v}_t}+\varepsilon \approx \varepsilon$, so the denominator does not scale with c; then the step $\propto c\hat{m}_t$ and scale-invariance is **slightly broken**.

3. **Role of ε:**

   The update divides by $\sqrt{\hat{v}_t}$; we need to avoid division by zero and uncontrolled updates when $\hat{v}_t$ is very small. So we ask what happens when $\hat{v}_t \to 0$ and how ε fixes it.

   The Adam step is
   $$\Delta w_t = -\alpha\frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\varepsilon}.$$
   Without ε, when $\hat{v}_t = 0$ (e.g. $t = 1$ with a single zero gradient, or a parameter that has never received a non-zero gradient) we would divide by zero. More generally, when $\hat{v}_t \ll 1$, $\sqrt{\hat{v}_t}$ is tiny and $1/\sqrt{\hat{v}_t}$ can be huge, so a single large $\hat{m}_t$ would produce an **exploding update**. Adding ε in the denominator gives
   $$\sqrt{\hat{v}_t}+\varepsilon \geq \varepsilon > 0,$$
   so the denominator is **lower-bounded**: the step magnitude is at most
   $$\frac{\alpha|\hat{m}_t|}{\varepsilon},$$
   which is bounded for bounded $\hat{m}_t$. So ε (1) prevents division by zero, (2) lower-bounds the denominator and prevents exploding updates when $\hat{v}_t$ is tiny (early in training or for rarely updated coordinates), and (3) ensures numerical stability. In practice ε is small (e.g. 10^{-8}) so that when $\hat{v}_t$ is already large, the denominator is hardly affected.

Part 3: Noisy gradients and sparse features (4 pts)

(a) **Learning-rate adaptation across parameters:**

We ask how each algorithm sets the *effective* step size per parameter. In high dimensions, different coordinates can have very different gradient scales; an optimizer that adapts per coordinate may behave better.

In Adam, the effective step for a parameter is

$$\Delta w_t = -\alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}.$$

Each parameter has its *own* m_t, v_t (and thus $\hat{m}_t$, $\hat{v}_t$) from its own gradient history. So $\sqrt{\hat{v}_t}$ is the RMS of that coordinate's recent gradients. A coordinate with consistently large $|g_t|$ has large $\hat{v}_t$, so the denominator is large and the **effective step is smaller**; one with small or rarely non-zero gradients has small $\hat{v}_t$, so the denominator is smaller and the **effective step is relatively larger**. So Adam adapts the learning rate **per parameter** using $\hat{v}_t$. In momentum SGD, the update is $-\alpha u_t$ with a single global α; u_t smooths gradients via

$$u_t = \mu u_{t-1} + g_t,$$

but there is no per-parameter rescaling by a second-moment term. So momentum SGD uses one global α and does **not** adapt step size per parameter based on gradient scale.

(b) **Noisy gradients and sparse features:**

With noisy gradients, we want to avoid overreacting to a few large spurious values. With sparse features, some parameters are updated rarely; we want them to still get meaningful updates when they do get a gradient. We compare how Adam's structure (especially $\hat{v}_t$) addresses these versus momentum.

Noisy gradients: In Adam, the step is normalized by $\sqrt{\hat{v}_t}$, which is an EMA of g_t^2:

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}.$$

So $\hat{v}_t$ reflects the *typical squared magnitude* of recent gradients. A single sporadic large gradient increases $\hat{v}_t$ only slightly (EMA smooths it), and the denominator $\sqrt{\hat{v}_t} + \varepsilon$ is already of the order of the typical scale; so the **RMS normalization** down-weights the impact of that large gradient. In other words, directions with high variance (noise) get a larger denominator and smaller effective steps, reducing the effect of spurious large gradients and stabilizing updates. In momentum SGD, u_t is an EMA of g_t, so one large g_t can still move u_t noticeably; there is no second-moment normalization, so noisy spikes can cause larger, less stable steps.

Sparse features: For a coordinate that is rarely non-zero (e.g. a sparse feature), most g_t are zero. In Adam, v_t (and $\hat{v}_t$) for that coordinate stays **small** because most $g_t^2 = 0$. When a non-zero gradient finally appears, the denominator $\sqrt{\hat{v}_t} + \varepsilon$ is still small, so the **effective step is relatively large**: that parameter gets a meaningful update and can "catch up." In momentum SGD, the same coordinate has small u_t (mostly zeros), so when a gradient appears the step is α times that gradient with no automatic boost from a small denominator; sparse features may therefore learn more slowly. Combined with noise, momentum can be less stable because there is no per-parameter scaling by $\sqrt{\hat{v}_t}$.

Problem 4: BloodMNIST (Data Loading and Exploration)

Q4.1: Data Loading and Exploration (2 points)

Tasks:

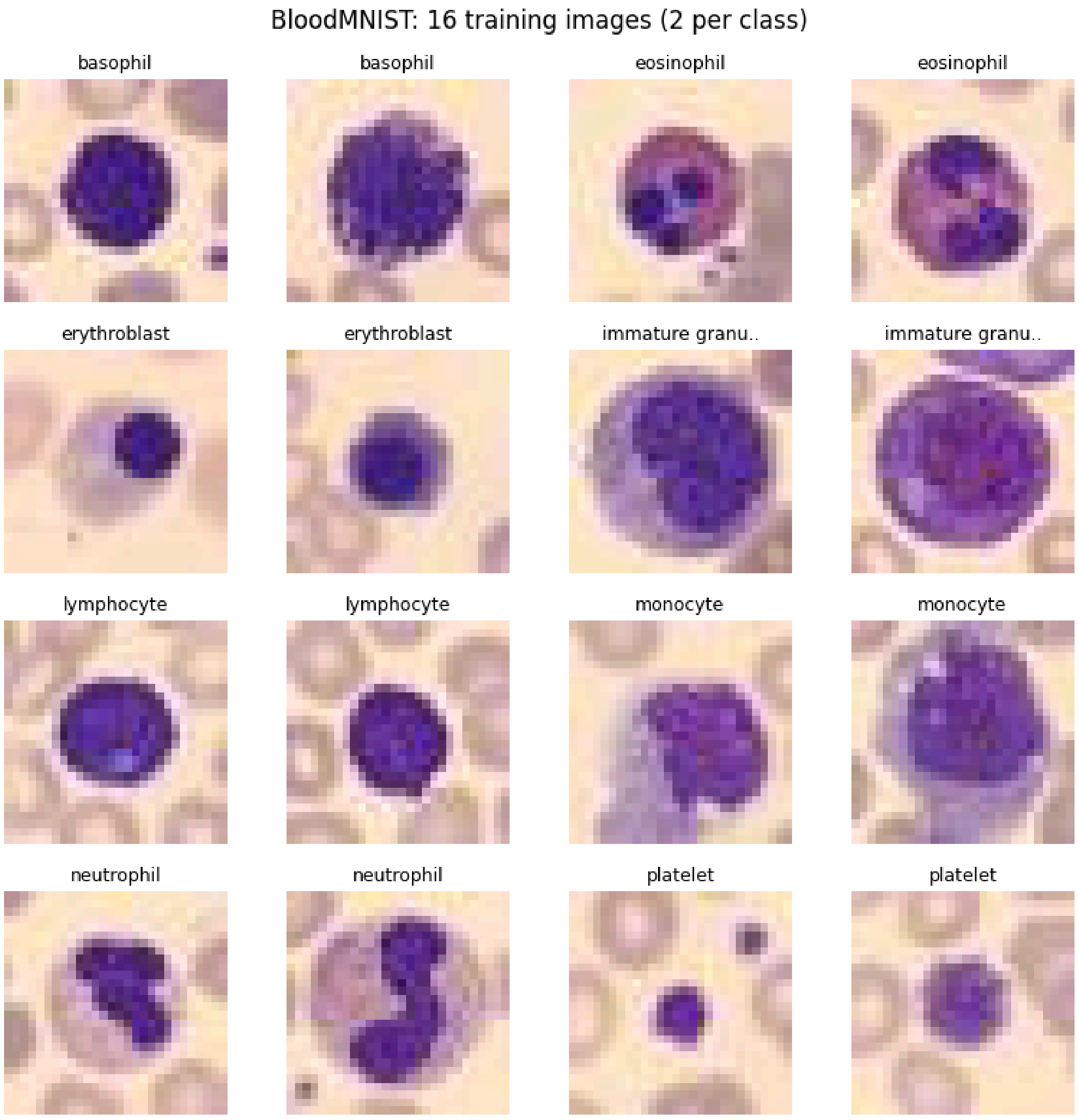
1. Install and load BloodMNIST using the `medmnist` package.
2. Report the number of samples in train/val/test splits.
3. Display a grid of 16 random training images (2 per class) with their class labels.
4. Plot the class distribution (bar chart) for the training set and comment on whether the dataset is balanced.

In [7]:
```python
# --- Task 1: Install and load BloodMNIST using the medmnist package ---

import os
import numpy as np
import matplotlib.pyplot as plt
from medmnist import INFO
from torchvision import transforms

data_dir = "./data"
os.makedirs(data_dir, exist_ok=True)

info = INFO["bloodmnist"]
BloodMNIST = getattr(medmnist, info["python_class"])
transform = transforms.ToTensor()

train_dataset = BloodMNIST(root=data_dir, split="train", transform=transform
val_dataset   = BloodMNIST(root=data_dir, split="val",   transform=transform
test_dataset  = BloodMNIST(root=data_dir, split="test",  transform=transform

num_classes = len(info["label"])
class_names = [info["label"][str(i)] for i in range(num_classes)]
```

In [8]:
```python
# --- Task 2: Report the number of samples in train/val/test splits ---
n_train = len(train_dataset)
n_val   = len(val_dataset)
n_test  = len(test_dataset)
print(f"Training samples:   {n_train}")
print(f"Validation samples: {n_val}")
print(f"Test samples:       {n_test}")
```

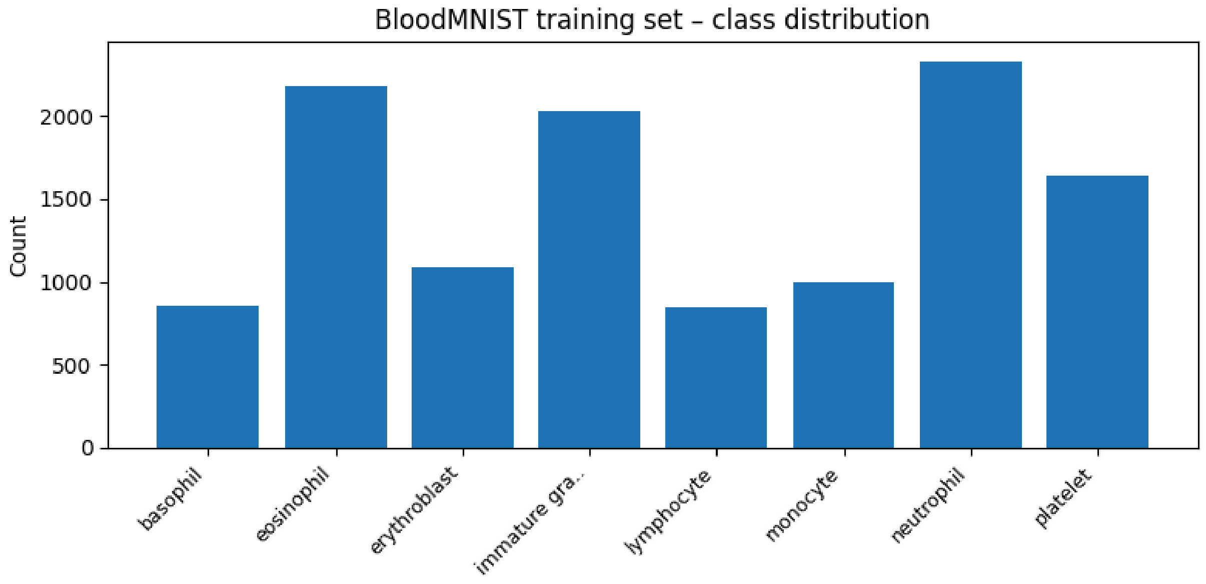
```
Training samples:   11959
Validation samples: 1712
Test samples:       3421
```

In [11]:
```python
# --- Task 3: Display a grid of 16 random training images (2 per class) with
indices_per_class = {i: [] for i in range(num_classes)}
for idx, (img, target) in enumerate(train_dataset):
    y = int(target.squeeze() if hasattr(target, "squeeze") else target)
    if len(indices_per_class[y]) < 2:
        indices_per_class[y].append(idx)
    if all(len(v) == 2 for v in indices_per_class.values()):
        break

selected = [idx for c in range(num_classes) for idx in indices_per_class[c]]
fig, axes = plt.subplots(4, 4, figsize=(8, 8))
for ax, idx in zip(axes.flatten(), selected):
    img, target = train_dataset[idx]
    y = int(target.squeeze() if hasattr(target, "squeeze") else target)
    disp = img.permute(1, 2, 0).numpy() if img.dim() == 3 else img.squeeze()
    ax.imshow(disp, cmap="gray" if disp.ndim == 2 else None)
    title = class_names[y][:14] + (".." if len(class_names[y]) > 14 else "")
    ax.set_title(title, fontsize=9)
    ax.axis("off")
plt.suptitle("BloodMNIST: 16 training images (2 per class)")
plt.tight_layout()
plt.show()
```



In [12]:
```
# --- Task 4: Plot class distribution (bar chart) for training set; comment
class_counts = np.zeros(num_classes, dtype=int)
for _, target in train_dataset:
    y = int(target.squeeze() if hasattr(target, "squeeze") else target)
    class_counts[y] += 1

plt.figure(figsize=(8, 4))
plt.bar(range(num_classes), class_counts)
# Shorten long class names for x-axis so the 4th class (e.g. immature granul
short_names = [s[:12] + ".." if len(s) > 12 else s for s in class_names]
plt.xticks(range(num_classes), short_names, rotation=45, ha="right", fontsiz
plt.ylabel("Count")
plt.title("BloodMNIST training set - class distribution")
plt.tight_layout()
plt.show()

# Distribution analysis
print("Class distribution (train):")
```

```
for i in range(num_classes):
    print(f"  {class_names[i]:25s}: {class_counts[i]:4d} ({100*class_counts[
print(f"\nMax: {class_counts.max()} ({class_names[class_counts.argmax()]}),
print("The dataset is not balanced: some classes have many more samples thar
```



```
Class distribution (train):
  basophil                 :  852 (7.1%)
  eosinophil               : 2181 (18.2%)
  erythroblast             : 1085 (9.1%)
  immature granulocytes(myelocytes, metamyelocytes and promyelocytes): 2026
(16.9%)
  lymphocyte               :  849 (7.1%)
  monocyte                 :  993 (8.3%)
  neutrophil               : 2330 (19.5%)
  platelet                 : 1643 (13.7%)

Max: 2330 (neutrophil), Min: 849 (lymphocyte)
The dataset is not balanced: some classes have many more samples than others
(e.g. neutrophil vs basophil).
```

Q4.2: Build and Train an MLP (5 points)

1. Flatten each 28×28×3 RGB image into a **2,352-dimensional** input vector
2. Build a 3-layer MLP: **Input 2352 → Hidden 1: 256 (ReLU) → Hidden 2: 128 (ReLU) → Output: 8**
3. **Adam** optimizer, lr=1e-3, **CrossEntropyLoss**
4. Train for **30 epochs**, batch size **64**
5. Plot **training loss** and **validation loss** (same figure)
6. Plot **training accuracy** and **validation accuracy** (same figure)

In [13]:
```
# --- Q4.2: DataLoaders and device ---
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
```

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
BATCH_SIZE = 64
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True
val_loader   = DataLoader(val_dataset,   batch_size=BATCH_SIZE, shuffle=Fals
```

In [14]:
```
# --- Q4.2: 3-layer MLP (2352 → 256 ReLU → 128 ReLU → 8) ---
class BloodMLP(nn.Module):
    def __init__(self):
        super(BloodMLP, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(28 * 28 * 3, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 8)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.flatten(x)
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

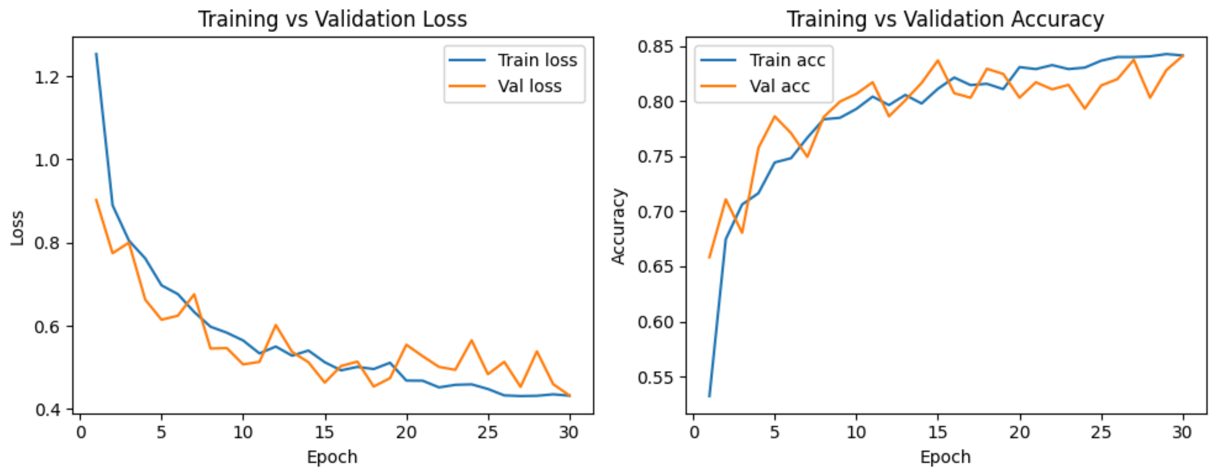
In [15]:
```
# --- Q4.2: Model, optimizer, loss ---
model = BloodMLP().to(device)
optimizer = optim.Adam(model.parameters(), lr=1e-3)
criterion = nn.CrossEntropyLoss()
```

In [16]:
```
# --- Q4.2: Train for 30 epochs, record train/val loss and accuracy ---
num_epochs = 30
train_losses, val_losses = [], []
train_accs, val_accs = [], []

for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    correct, total = 0, 0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        labels = labels.squeeze().long()
        optimizer.zero_grad()
        logits = model(images)
        loss = criterion(logits, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
        correct += (logits.argmax(dim=1) == labels).sum().item()
        total += labels.size(0)

    train_losses.append(running_loss / len(train_loader))
    train_accs.append(correct / total)

    model.eval()
    val_loss = 0.0
    val_correct, val_total = 0, 0
    with torch.no_grad():
        for images, labels in val_loader:
            images, labels = images.to(device), labels.to(device)
            labels = labels.squeeze().long()
            logits = model(images)
            val_loss += criterion(logits, labels).item()
            val_correct += (logits.argmax(dim=1) == labels).sum().item()
            val_total += labels.size(0)

    val_losses.append(val_loss / len(val_loader))
    val_accs.append(val_correct / val_total)
    print(f"Epoch {epoch+1}/{num_epochs}  Train Loss: {train_losses[-1]:.4f}
```

```
Epoch 1/30  Train Loss: 1.2534 Acc: 0.5322  |  Val Loss: 0.9025 Acc: 0.6583
Epoch 2/30  Train Loss: 0.8898 Acc: 0.6747  |  Val Loss: 0.7746 Acc: 0.7109
Epoch 3/30  Train Loss: 0.8050 Acc: 0.7062  |  Val Loss: 0.8001 Acc: 0.6805
Epoch 4/30  Train Loss: 0.7621 Acc: 0.7164  |  Val Loss: 0.6630 Acc: 0.7576
Epoch 5/30  Train Loss: 0.6974 Acc: 0.7444  |  Val Loss: 0.6148 Acc: 0.7862
Epoch 6/30  Train Loss: 0.6763 Acc: 0.7482  |  Val Loss: 0.6244 Acc: 0.7710
Epoch 7/30  Train Loss: 0.6333 Acc: 0.7669  |  Val Loss: 0.6758 Acc: 0.7494
Epoch 8/30  Train Loss: 0.5980 Acc: 0.7836  |  Val Loss: 0.5456 Acc: 0.7856
Epoch 9/30  Train Loss: 0.5835 Acc: 0.7849  |  Val Loss: 0.5466 Acc: 0.7996
Epoch 10/30  Train Loss: 0.5648 Acc: 0.7930  |  Val Loss: 0.5073 Acc: 0.8067
Epoch 11/30  Train Loss: 0.5337 Acc: 0.8042  |  Val Loss: 0.5134 Acc: 0.8172
Epoch 12/30  Train Loss: 0.5503 Acc: 0.7965  |  Val Loss: 0.6021 Acc: 0.7862
Epoch 13/30  Train Loss: 0.5283 Acc: 0.8057  |  Val Loss: 0.5374 Acc: 0.8008
Epoch 14/30  Train Loss: 0.5407 Acc: 0.7979  |  Val Loss: 0.5125 Acc: 0.8166
Epoch 15/30  Train Loss: 0.5125 Acc: 0.8113  |  Val Loss: 0.4635 Acc: 0.8370
Epoch 16/30  Train Loss: 0.4931 Acc: 0.8215  |  Val Loss: 0.5035 Acc: 0.8072
Epoch 17/30  Train Loss: 0.5013 Acc: 0.8146  |  Val Loss: 0.5140 Acc: 0.8032
Epoch 18/30  Train Loss: 0.4962 Acc: 0.8158  |  Val Loss: 0.4541 Acc: 0.8294
Epoch 19/30  Train Loss: 0.5113 Acc: 0.8109  |  Val Loss: 0.4743 Acc: 0.8248
Epoch 20/30  Train Loss: 0.4685 Acc: 0.8309  |  Val Loss: 0.5546 Acc: 0.8032
Epoch 21/30  Train Loss: 0.4682 Acc: 0.8292  |  Val Loss: 0.5271 Acc: 0.8172
Epoch 22/30  Train Loss: 0.4521 Acc: 0.8328  |  Val Loss: 0.5013 Acc: 0.8107
Epoch 23/30  Train Loss: 0.4582 Acc: 0.8292  |  Val Loss: 0.4943 Acc: 0.8148
Epoch 24/30  Train Loss: 0.4594 Acc: 0.8305  |  Val Loss: 0.5652 Acc: 0.7932
Epoch 25/30  Train Loss: 0.4482 Acc: 0.8368  |  Val Loss: 0.4837 Acc: 0.8143
Epoch 26/30  Train Loss: 0.4328 Acc: 0.8401  |  Val Loss: 0.5135 Acc: 0.8201
Epoch 27/30  Train Loss: 0.4313 Acc: 0.8401  |  Val Loss: 0.4533 Acc: 0.8376
Epoch 28/30  Train Loss: 0.4319 Acc: 0.8407  |  Val Loss: 0.5386 Acc: 0.8032
Epoch 29/30  Train Loss: 0.4354 Acc: 0.8428  |  Val Loss: 0.4598 Acc: 0.8283
Epoch 30/30  Train Loss: 0.4322 Acc: 0.8415  |  Val Loss: 0.4325 Acc: 0.8411
```

In [17]:
```
# --- Q4.2: Plot training vs validation loss and accuracy ---
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))
epochs = range(1, num_epochs + 1)
ax1.plot(epochs, train_losses, label="Train loss")
ax1.plot(epochs, val_losses, label="Val loss")
ax1.set_xlabel("Epoch")
ax1.set_ylabel("Loss")
ax1.set_title("Training vs Validation Loss")
ax1.legend()

ax2.plot(epochs, train_accs, label="Train acc")
ax2.plot(epochs, val_accs, label="Val acc")
```

```
ax2.set_xlabel("Epoch")
ax2.set_ylabel("Accuracy")
ax2.set_title("Training vs Validation Accuracy")
ax2.legend()
plt.tight_layout()
plt.show()
```



Q4.3: Evaluation and Analysis (5 points)

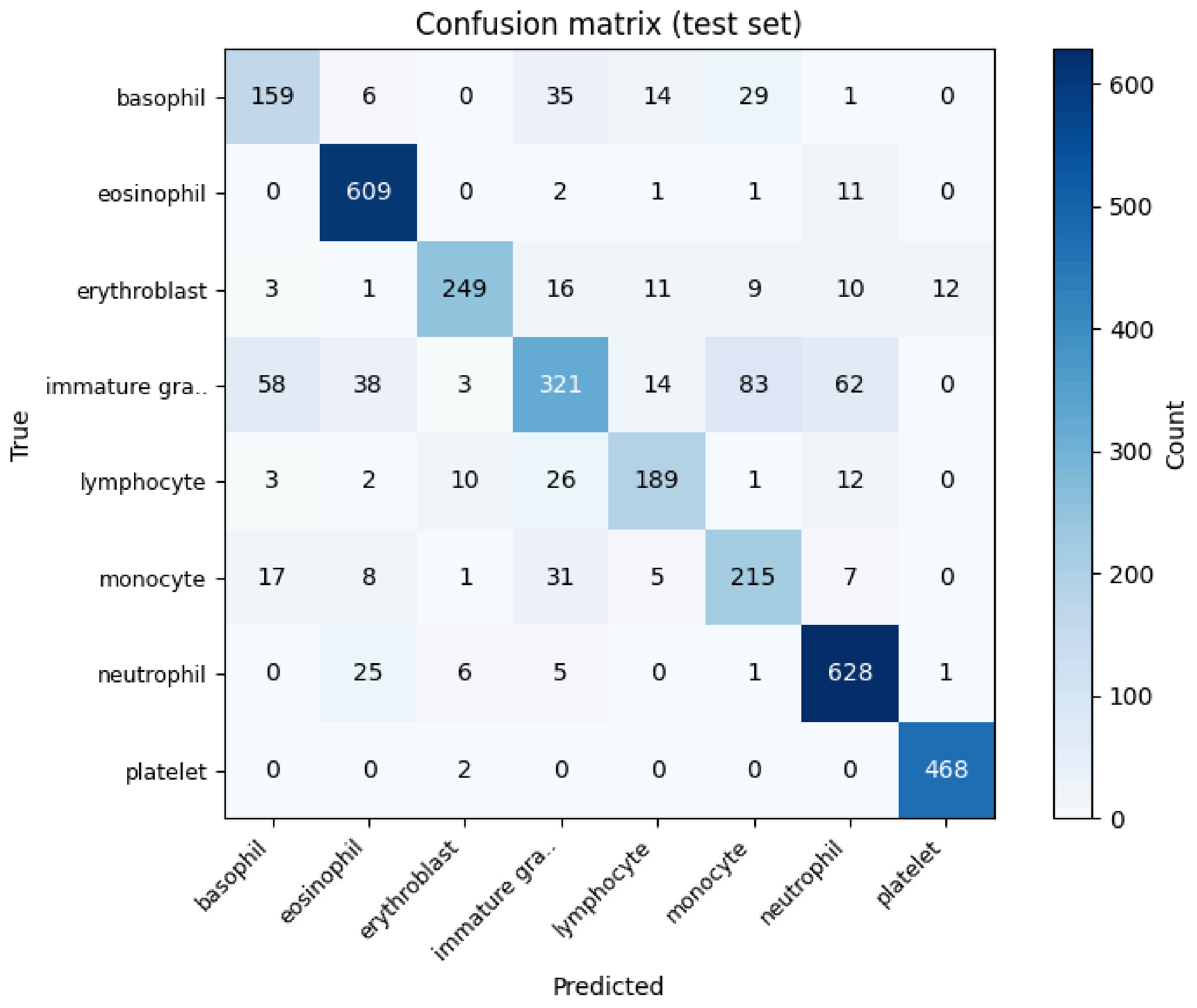
1. Report the **final test accuracy** (as a percentage)
2. Generate a **confusion matrix** (8×8) for the test set using `sklearn.metrics.confusion_matrix` and visualize it as a **heatmap** with class labels
3. Identify the **two most confused cell type pairs** (highest off-diagonal values)
4. Compute and report **per-class precision and recall** using `sklearn.metrics.classification_report`
5. Which cell type has the **lowest recall**? Examine **5 misclassified examples** of this cell type and hypothesize why the model struggles with it.

In [18]:
```
# --- Q4.3.1: Final test accuracy ---
from torch.utils.data import DataLoader
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

model.eval()
all_preds, all_labels = [], []
with torch.no_grad():
    for images, labels in test_loader:
        images = images.to(device)
        labels = labels.squeeze().long()
        logits = model(images)
        all_preds.append(logits.argmax(dim=1).cpu())
        all_labels.append(labels.cpu())

all_preds = torch.cat(all_preds).numpy()
all_labels = torch.cat(all_labels).numpy()
```

```
test_acc = 100.0 * (all_preds == all_labels).mean()
print(f"Final test accuracy: {test_acc:.2f}%")
```

```
Final test accuracy: 82.96%
```

In [20]:
```
# --- Q4.3.2: Confusion matrix (8×8) heatmap with class labels ---
from sklearn.metrics import confusion_matrix

cm = confusion_matrix(all_labels, all_preds)
fig, ax = plt.subplots(figsize=(8, 6))
im = ax.imshow(cm, cmap="Blues")
plt.colorbar(im, ax=ax, label="Count")
ax.set_xticks(range(8))
ax.set_yticks(range(8))
short_names = [s[:12] + '..' if len(s) > 12 else s for s in class_names]
ax.set_xticklabels(short_names, rotation=45, ha="right", fontsize=9)
ax.set_yticklabels(short_names, fontsize=9)
ax.set_xlabel("Predicted")
ax.set_ylabel("True")
for i in range(8):
    for j in range(8):
        ax.text(j, i, cm[i, j], ha="center", va="center", color="black" if c
plt.title("Confusion matrix (test set)")
plt.tight_layout()
plt.show()
```



True \ Predicted	basophil	eosinophil	erythroblast	immature gra..	lymphocyte	monocyte	neutrophil	platelet
basophil	159	6	0	35	14	29	1	0
eosinophil	0	609	0	2	1	1	11	0
erythroblast	3	1	249	16	11	9	10	12
immature gra..	58	38	3	321	14	83	62	0
lymphocyte	3	2	10	26	189	1	12	0
monocyte	17	8	1	31	5	215	7	0
neutrophil	0	25	6	5	0	1	628	1
platelet	0	0	2	0	0	0	0	468

In [21]:
```
# --- Q4.3.3: Two most confused cell type pairs (highest off-diagonal) ---
# Copy cm and zero out diagonal to get only off-diagonal
cm_off = cm.copy()
for i in range(8):
    cm_off[i, i] = 0

# Find indices of 2 largest off-diagonal values (flat index)
flat = cm_off.flatten()
top2_flat = np.argsort(flat)[-2:][::-1]
for idx in top2_flat:
    i, j = idx // 8, idx % 8
    print(f"  {class_names[i]} predicted as {class_names[j]}: {cm[i, j]} tim
print("(Two most confused pairs above.)")
```

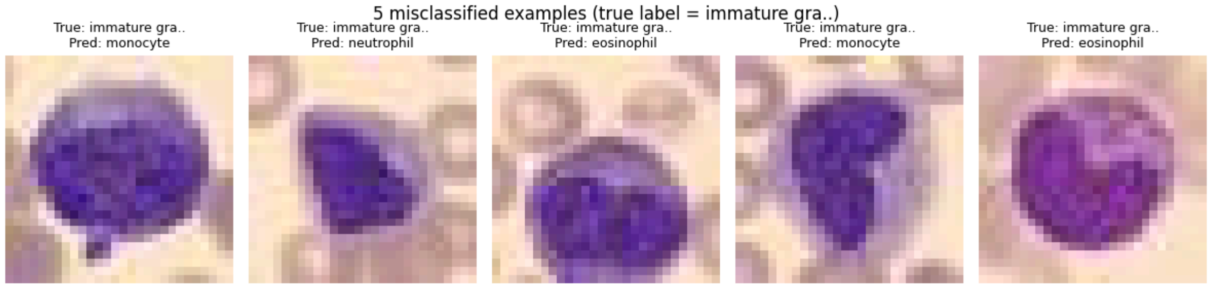
```
  immature granulocytes(myelocytes, metamyelocytes and promyelocytes) predic
ted as monocyte: 83 times
  immature granulocytes(myelocytes, metamyelocytes and promyelocytes) predic
ted as neutrophil: 62 times
(Two most confused pairs above.)
```

In [24]:
```
# --- Q4.3.4: Per-class precision and recall (classification_report) ---
from sklearn.metrics import classification_report

print(classification_report(all_labels, all_preds, target_names=class_names,
```

```
                                                                     precisi
on    recall  f1-score   support


                                                       basophil      0.6
62     0.652     0.657       244
                                                     eosinophil      0.8
84     0.976     0.928       624
                                                  erythroblast      0.9
19     0.801     0.856       311
immature granulocytes(myelocytes, metamyelocytes and promyelocytes)      0.7
36     0.554     0.633       579
                                                    lymphocyte      0.8
08     0.778     0.792       243
                                                      monocyte      0.6
34     0.757     0.690       284
                                                    neutrophil      0.8
59     0.943     0.899       666
                                                      platelet      0.9
73     0.996     0.984       470


                                                      accuracy
0.830      3421
                                                     macro avg      0.8
09     0.807     0.805      3421
                                                  weighted avg      0.8
28     0.830     0.825      3421
```

In [25]:
```
# --- Q4.3.5: Cell type with lowest recall; show 5 misclassified examples --
from sklearn.metrics import precision_recall_fscore_support
precision, recall, _, _ = precision_recall_fscore_support(all_labels, all_pr
lowest_recall_class = int(np.argmin(recall))
print(f"Cell type with lowest recall: {class_names[lowest_recall_class]} (re

# Indices in test set where true=lowest_recall_class but predicted != lowest
mis_mask = (all_labels == lowest_recall_class) & (all_preds != lowest_recall
mis_indices = np.where(mis_mask)[0][:5]

short_names = [s[:12] + '..' if len(s) > 12 else s for s in class_names]
fig, axes = plt.subplots(1, 5, figsize=(12, 3))
for ax, idx in zip(axes, mis_indices):
    img, _ = test_dataset[idx]
    true_lab = all_labels[idx]
    pred_lab = all_preds[idx]
    disp = img.permute(1, 2, 0).numpy() if img.dim() == 3 else img.squeeze()
    ax.imshow(disp, cmap="gray" if disp.ndim == 2 else None)
    ax.set_title(f"True: {short_names[true_lab]}\nPred: {short_names[pred_la
    ax.axis("off")
plt.suptitle(f"5 misclassified examples (true label = {short_names[lowest_re
plt.tight_layout()
plt.show()
```

```
Cell type with lowest recall: immature granulocytes(myelocytes, metamyelocyt
es and promyelocytes) (recall = 0.554)
```



Hypothesis (why the model struggles with this cell type):
The class with lowest recall is often a minority class or one that is visually similar to others (e.g. immature granulocytes vs neutrophils). The model may confuse it with the most confused pair identified above, or the training set may have few examples. Inspect the 5 misclassified images: shared appearance (size, shape, color) with the predicted class can explain the errors.

Q4.4: Prediction Confidence Analysis (3 points)

Categorize predictions into four quadrants by **confidence** (max softmax prob) and **correctness**:

- **High confidence**: max prob > 0.9
- **Low confidence**: max prob < 0.6

	Correct	Incorrect
High confidence	✓ Confident	× Confident
Low confidence	✓ Uncertain	× Uncertain

1. Find and display **2 examples from each quadrant** (8 images total) with: image, true label, predicted label, prediction confidence.
2. **Written analysis (3–4 sentences)**: What distinguishes "Incorrect but Confident" examples? Why might the model be overconfident?

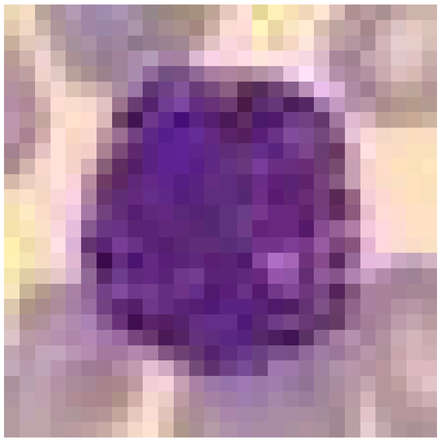
In [26]:
```
# --- Q4.4: Get max softmax probability for each test sample ---
model.eval()
all_probs = []
with torch.no_grad():
    for images, _ in test_loader:
        images = images.to(device)
        logits = model(images)
        probs = torch.softmax(logits, dim=1)
        max_prob, _ = probs.max(dim=1)
        all_probs.append(max_prob.cpu().numpy())
all_probs = np.concatenate(all_probs)

correct = (all_preds == all_labels)
high_conf = (all_probs > 0.9)
low_conf = (all_probs < 0.6)

# Four quadrants: (correct, high), (correct, low), (incorrect, high), (incor
quadrants = [
    ("Correct, High conf", np.where(correct & high_conf)[0]),
    ("Correct, Low conf",  np.where(correct & low_conf)[0]),
    ("Incorrect, High conf", np.where(~correct & high_conf)[0]),
    ("Incorrect, Low conf",  np.where(~correct & low_conf)[0]),
]
for name, idx in quadrants:
    print(f"  {name}: {len(idx)} samples")
```

```
Correct, High conf: 1927 samples
Correct, Low conf: 287 samples
Incorrect, High conf: 53 samples
Incorrect, Low conf: 302 samples
```

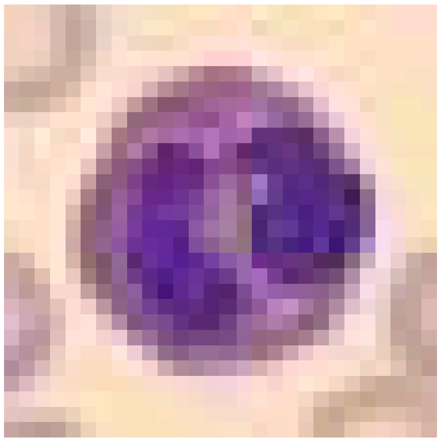
In [27]:
```
# --- Q4.4: Display 2 examples from each quadrant (8 images total) ---
short_names = [s[:12] + ".." if len(s) > 12 else s for s in class_names]
fig, axes = plt.subplots(4, 2, figsize=(8, 12))
for row, (quad_name, indices) in enumerate(quadrants):
    chosen = indices[:2]
    for col, idx in enumerate(chosen):
        ax = axes[row, col]
        img, _ = test_dataset[idx]
        disp = img.permute(1, 2, 0).numpy() if img.dim() == 3 else img.squee
        ax.imshow(disp, cmap="gray" if disp.ndim == 2 else None)
        true_lab = all_labels[idx]
        pred_lab = all_preds[idx]
        conf = all_probs[idx]
        ax.set_title(f"True: {short_names[true_lab]}\nPred: {short_names[pre
        ax.axis("off")
    axes[row, 0].set_ylabel(quad_name, fontsize=10)
plt.suptitle("Q4.4: 2 examples per quadrant (Correct/Incorrect × High/Low co
plt.tight_layout()
plt.show()
```

Q4.4: 2 examples per quadrant (Correct/Incorrect × High/Low confidence)

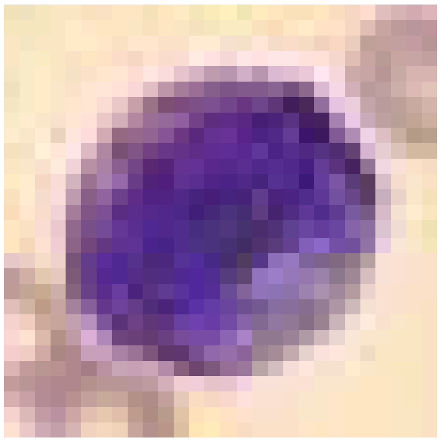
True: basophil
Pred: basophil
Conf: 0.945



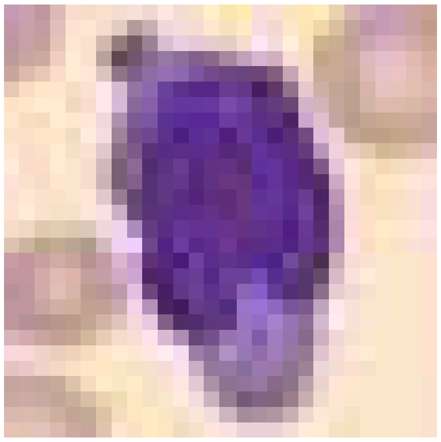
True: eosinophil
Pred: eosinophil
Conf: 0.995



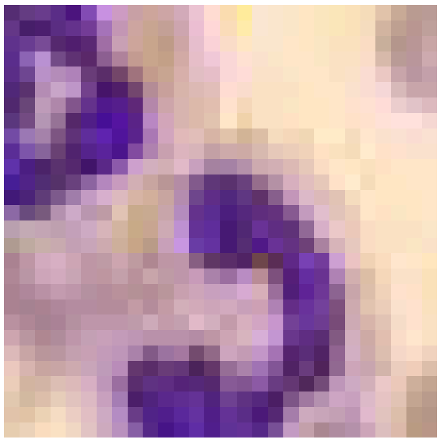
True: immature gra..
Pred: immature gra..
Conf: 0.459



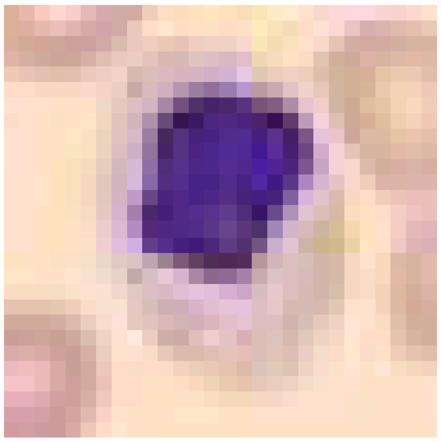
True: monocyte
Pred: monocyte
Conf: 0.453



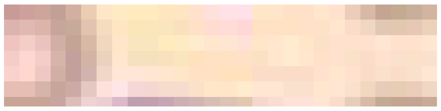
True: neutrophil
Pred: eosinophil
Conf: 0.939



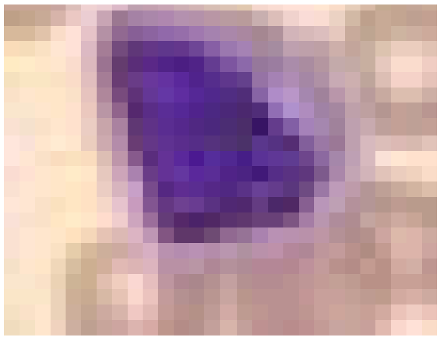
True: neutrophil
Pred: erythroblast
Conf: 0.916



True: immature gra..
Pred: neutrophil
Conf: 0.455



True: eosinophil
Pred: immature gra..
Conf: 0.436



Written analysis (3–4 sentences):
The "Incorrect but Confident" examples are cases where the model assigns high probability (>0.9) to the wrong class. Often these are cell types that look visually similar to the predicted class (e.g. granulocytes vs neutrophils, or similar staining and shape), so the model has learned features that confidently but wrongly distinguish them. The model may be overconfident on these because the training data or the 28×28 resolution does not capture the subtle differences that a pathologist would use, so the classifier relies on strong but misleading cues and assigns high confidence to the wrong class.

Problem 5: Weight Initialization and Training Dynamics (15 points)

Q5.1: Implement Initialization Schemes (3 points)

In [2]:
```python
import math
import torch
```

In [3]:
```python
def initialize_weights(shape, method):
    """
    Args:
        shape: tuple of (fan_in, fan_out)
        method: 'zero', 'small_random', 'xavier', 'he'
    Returns:
        torch.Tensor of initialized weights
    """

    if len(shape) != 2:
        raise ValueError("Shape must be a tuple of (fan_in, fan_out)")

    fan_in, fan_out = shape


    if method == 'zero':
        return torch.zeros(shape)
    elif method == 'small_random':
        return torch.randn(shape) * 0.01

    elif method == 'xavier':
        sigma = math.sqrt(2 / (fan_in + fan_out))
        return torch.randn(shape) * sigma
    elif method == 'he':
        sigma = math.sqrt(2 / fan_in)
        return torch.randn(shape) * sigma
    else:
        raise ValueError("Unknown method")
```

In [4]:
```python
print(initialize_weights((10, 10), 'zero'))
```

```
tensor([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

In [5]:
```python
print(initialize_weights((10, 10), 'small_random'))
```

```
tensor([[-7.3728e-03,  2.0317e-02, -9.1637e-03, -7.7929e-03, -1.6863e-02,
          1.9904e-03, -2.0977e-03,  8.7249e-03,  4.7539e-03, -5.0325e-05],
        [ 2.1910e-02, -8.2438e-03,  1.8353e-03,  1.9136e-02,  2.1032e-02,
          2.6045e-02,  8.7373e-03,  4.9453e-03, -5.2550e-04, -9.2108e-03],
        [ 9.8186e-06,  1.1029e-02,  7.2815e-03,  9.2429e-03,  1.6524e-02,
         -2.0089e-02,  1.2951e-02,  1.1343e-02, -1.2502e-03,  1.1907e-02],
        [ 8.5119e-04,  1.0017e-02, -5.3816e-03, -4.1842e-03,  2.4173e-03,
          1.5439e-02, -1.2378e-03,  8.5695e-03, -8.1867e-03,  7.1069e-03],
        [-1.0179e-02, -9.4137e-03,  3.2180e-03,  7.6749e-03, -1.2479e-02,
         -1.1913e-03, -6.4261e-03,  1.7171e-02,  1.0931e-02,  4.0554e-04],
        [-5.6521e-03, -1.6655e-02,  8.7556e-03, -1.1121e-03,  8.4691e-03,
         -2.4977e-02, -1.4781e-02,  3.5186e-03, -3.5608e-03,  3.5375e-03],
        [ 2.7990e-03, -4.6826e-03, -2.6930e-02,  8.3600e-03, -2.8309e-03,
          3.4185e-03, -1.8099e-03,  2.3390e-03, -2.3730e-02,  1.5198e-02],
        [-1.3285e-02, -5.3940e-03, -1.4910e-03,  1.0418e-02, -2.1128e-03,
          4.3383e-03, -1.3918e-02, -7.9023e-03,  1.2211e-02, -1.4695e-02],
        [ 2.1684e-02,  5.5362e-04, -6.7632e-03,  1.8570e-02, -1.7721e-03,
         -1.8883e-02, -1.6094e-02, -1.4597e-02, -1.1398e-02, -1.2175e-02],
        [ 8.6801e-03, -3.2961e-03, -3.3023e-05, -7.8670e-03,  8.1508e-03,
         -7.3199e-03,  6.3486e-03, -1.0573e-02, -6.3033e-03, -5.6826e-04]])
```

```
In [6]: print(initialize_weights((10, 10), 'xavier'))
```

```
tensor([[-1.9522e-01,  4.6267e-01, -3.9077e-03,  3.1891e-01, -8.1778e-02,
         -1.0004e-01, -1.6565e-01, -7.0138e-02,  2.3415e-01, -4.3308e-01],
        [-4.9960e-01, -2.1628e-02,  3.5275e-01,  7.1366e-02,  1.9426e-01,
          1.8613e-02, -2.0105e-01,  4.0095e-01, -3.6415e-01, -5.7032e-01],
        [-7.4455e-01, -3.5765e-01,  9.0227e-02, -8.2679e-02,  6.4100e-02,
          4.1779e-01,  5.2513e-01, -4.5788e-02,  2.4235e-01,  8.4194e-02],
        [ 2.3923e-01,  1.2945e-01,  2.3440e-01, -6.0221e-01, -2.0571e-01,
          3.0181e-01, -6.6088e-01, -4.8119e-01, -3.7612e-01,  3.0595e-01],
        [ 6.0871e-01,  3.8558e-01, -4.7836e-01,  7.9041e-02,  3.2196e-01,
          7.2231e-01,  1.3444e-01, -2.1802e-01, -6.0108e-02, -2.9329e-01],
        [ 1.0758e-01,  4.0476e-01,  9.9169e-02, -4.6255e-02, -3.3340e-01,
         -4.6265e-01,  5.3194e-02, -2.7121e-01,  4.1252e-04, -2.3865e-01],
        [ 2.3185e-01, -6.1036e-02,  3.5239e-01,  4.2175e-01,  2.6114e-01,
          1.1305e-01, -4.8434e-02, -3.7686e-01, -2.8912e-01,  2.1219e-01],
        [-4.5900e-01, -3.6447e-01,  3.8754e-02, -2.6771e-01, -4.6433e-02,
         -1.0833e-01,  1.3636e-01,  3.5308e-01,  8.5155e-02, -3.8674e-01],
        [ 9.0305e-02,  2.7343e-02,  2.6054e-01,  1.1212e-02, -1.6176e-01,
          2.0786e-02,  4.4808e-01,  9.2547e-02,  1.5002e-01,  2.2513e-01],
        [-2.8721e-01,  3.4034e-01, -2.3397e-01,  1.0475e-01, -1.5091e-01,
         -1.3869e-01, -4.2362e-01, -2.0444e-01, -6.9800e-01, -4.3488e-01]])
```

```
In [7]: print(initialize_weights((10, 10), 'he'))
```

```
tensor([[-0.4747,  0.2276,  0.6711, -0.0488,  0.3201,  0.2678, -0.2571,  0.7
028,
         -0.1859, -0.0471],
        [-0.5096, -0.1429, -0.4816,  0.4405,  0.1535, -0.0602, -0.3839, -0.2
299,
          0.0330,  0.1493],
        [-0.4028, -0.5051, -0.6254, -0.7087, -0.5010,  0.0422, -0.4240,  0.3
202,
         -0.3374,  1.3728],
        [-0.4379,  0.1670,  0.0026,  0.0277,  0.4402, -0.0052,  0.4097,  0.5
598,
          0.8233, -0.2221],
        [-0.8151,  0.1944,  0.3852,  0.7843,  0.2673,  0.8377,  0.2311, -0.1
750,
         -0.2108,  0.8659],
        [-0.8962, -0.3528, -0.2630, -0.0697,  0.1602, -0.8806,  0.6894, -0.4
559,
         -0.7360,  0.3275],
        [ 0.4181,  0.1856,  0.0891,  0.6988,  0.2787,  0.0607,  0.1608, -0.4
034,
         -0.1173,  0.7348],
        [-0.2997, -0.2791, -0.5637,  0.0288,  0.5149, -0.1368,  0.0627, -0.1
159,
         -0.1855,  0.1012],
        [ 0.1542, -0.6843,  0.9115, -0.0068,  0.1636, -0.3744,  0.0169,  0.1
341,
         -0.2064,  0.9942],
        [-0.1578, -0.4001,  0.5640, -0.0671, -0.2171,  0.3364, -0.1550, -0.4
566,
         -0.1502, -0.3381]])
```

```
In [ ]:
```

Problem 5.2: Activation Statistics Before Training (5 points)

Q5.2: Activation Statistics Before Training (5 points)

Build a 6-layer MLP for MNIST classification:

- Architecture: 784 → 256 → 256 → 256 → 256 → 256 → 10
- Use Tanh activations for all hidden layers

For each of the 4 initialization methods:

1. Initialize the network (do **NOT** train yet)
2. Forward pass a batch of 256 random MNIST images
3. Record the mean and standard deviation of activations at each of the 5 hidden layers
4. Create a figure with 2 subplots:
    - Subplot 1: Mean activation vs. layer depth (4 lines, one per init method)
    - Subplot 2: Std of activation vs. layer depth (4 lines, one per init method)

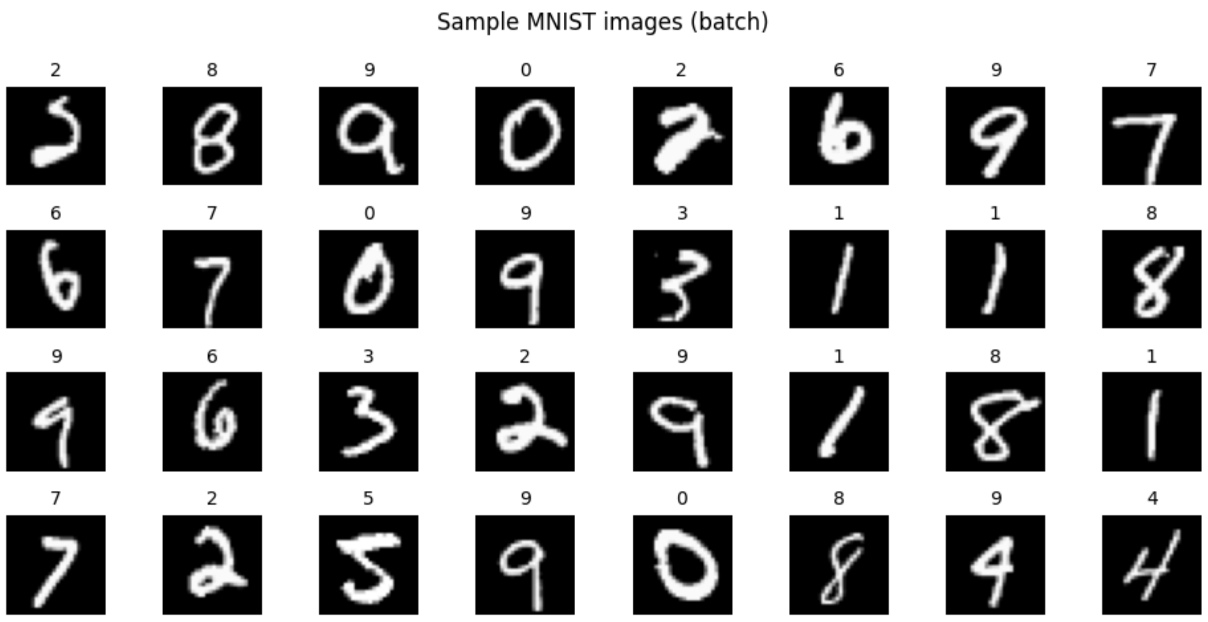
In [39]:
```
import torch.nn as nn
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

transform = transforms.ToTensor()

# Load MNIST
train_dataset = datasets.MNIST(root="./data", train=True, download=True, tra

# DataLoader: batch of 256 random MNIST images
batch_size = 256
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True

# Get one batch for activation statistics (Q5.2)
batch_images, batch_labels = next(iter(train_loader))
print(f"Batch images shape: {batch_images.shape}")
print(f"Batch labels shape: {batch_labels.shape}")
```

```
Batch images shape: torch.Size([256, 1, 28, 28])
Batch labels shape: torch.Size([256])
```

Visualizing some of the images from the dataset

In [40]:
```
import matplotlib.pyplot as plt

# Visualize a grid of images from the batch
n_rows, n_cols = 4, 8
n_show = n_rows * n_cols
```

```python
fig, axes = plt.subplots(n_rows, n_cols, figsize=(n_cols * 1.2, n_rows * 1.2
for i, ax in enumerate(axes.flat):
    if i < n_show:
        img = batch_images[i].squeeze()
        ax.imshow(img, cmap="gray")
        ax.set_title(int(batch_labels[i]), fontsize=10)
    ax.axis("off")
plt.suptitle("Sample MNIST images (batch)", fontsize=12)
plt.tight_layout()
plt.show()
```



In [41]:
```python
import math
import torch
import torch.nn as nn


def initialize_weights(shape, method):
    """
    Args:
        shape: tuple of (fan_in, fan_out)
        method: 'zero', 'small_random', 'xavier', 'he'
    Returns:
        torch.Tensor of initialized weights
    """
    if len(shape) != 2:
        raise ValueError("Shape must be a tuple of (fan_in, fan_out)")
    fan_in, fan_out = shape
    if method == 'zero':
        return torch.zeros(shape)
    elif method == 'small_random':
        return torch.randn(shape) * 0.01
    elif method == 'xavier':
        sigma = math.sqrt(2 / (fan_in + fan_out))
        return torch.randn(shape) * sigma
    elif method == 'he':
        sigma = math.sqrt(2 / fan_in)
        return torch.randn(shape) * sigma
    else:
        raise ValueError("Unknown method")


class MLPLayer(nn.Module):
    """
    6-layer MLP for MNIST: 784 → 256 → 256 → 256 → 256 → 256 → 10
    Tanh activations on all hidden layers.
    """
    def __init__(self, init_method="xavier"):
        super().__init__()
        self.init_method = init_method

        self.fc1 = nn.Linear(784, 256)
        self.fc2 = nn.Linear(256, 256)
        self.fc3 = nn.Linear(256, 256)
        self.fc4 = nn.Linear(256, 256)
        self.fc5 = nn.Linear(256, 256)
        self.fc6 = nn.Linear(256, 10)

        self.act = nn.Tanh()

        self._init_weights(init_method)

    ## reusing the initialize_weights function from Q5.1
    def _init_weights(self, method):
        for m in [self.fc1, self.fc2, self.fc3, self.fc4, self.fc5, self.fc6
            w = initialize_weights((m.in_features, m.out_features), method)
            m.weight.data = w.t()
            m.bias.data.zero_()

    def forward(self, x):
        # x: (batch, 784) — flatten in caller if x is (batch, 1, 28, 28)
        h1 = self.act(self.fc1(x))
        h2 = self.act(self.fc2(h1))
        h3 = self.act(self.fc3(h2))
        h4 = self.act(self.fc4(h3))
        h5 = self.act(self.fc5(h4))
        out = self.fc6(h5)
        # Return logits and hidden activations for Q5.2 stats
        return out, (h1, h2, h3, h4, h5)
```

In [42]:
```
## one batch of 256 random MNIST images
x = batch_images.flatten(start_dim=1)
print(x.shape)
```

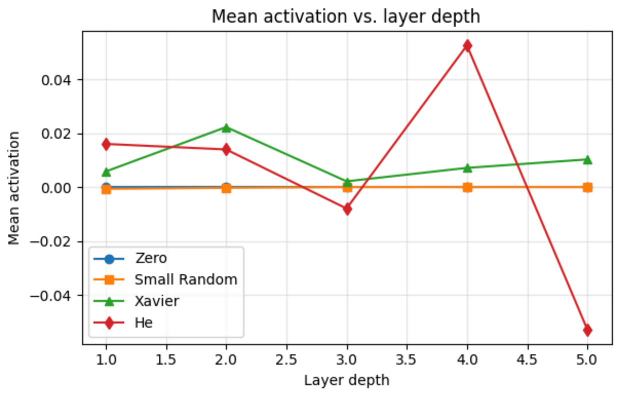
```
torch.Size([256, 784])
```

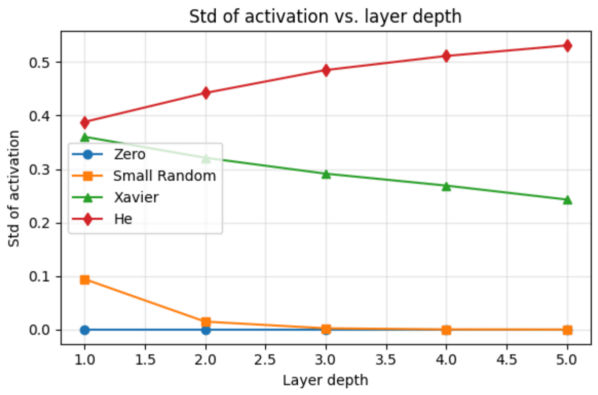
In [ ]:
```
## different models with different initialization methods
model_zero = MLPLayer(init_method="zero")
model_random = MLPLayer(init_method="small_random")
model_xavier = MLPLayer(init_method="xavier")
model_he = MLPLayer(init_method="he")
```

```
def unpack_forward(result):
    if len(result) == 2:
        logits, hiddens = result
        return logits, hiddens
    logits, h1, h2, h3, h4, h5 = result
    return logits, [h1, h2, h3, h4, h5]

mode_zero_results, mode_zero_hidden = unpack_forward(model_zero(x))
mode_random_results, mode_random_hidden = unpack_forward(model_random(x))
mode_xavier_results, mode_xavier_hidden = unpack_forward(model_xavier(x))
mode_he_results, mode_he_hidden = unpack_forward(model_he(x))



## testing the


for i in range(5):
    print("Mean of hidden layer " + str(i+1) + " for mode zero: " + str(mode
    print("Std of hidden layer " + str(i+1) + " for mode zero: " + str(mode_

print("---------------------------------")

for i in range(5):
    print("Mean of hidden layer " + str(i+1) + " for mode random: " + str(mo
    print("Std of hidden layer " + str(i+1) + " for mode random: " + str(moc

print("---------------------------------")

for i in range(5):
    print("Mean of hidden layer " + str(i+1) + " for mode xavier: " + str(mo
    print("Std of hidden layer " + str(i+1) + " for mode xavier: " + str(moc

print("---------------------------------")

for i in range(5):
    print("Mean of hidden layer " + str(i+1) + " for mode he: " + str(mode_h
    print("Std of hidden layer " + str(i+1) + " for mode he: " + str(mode_he
Mean of hidden layer 1 for mode zero: tensor(0., grad_fn=<MeanBackward0>)
Std of hidden layer 1 for mode zero: tensor(0., grad_fn=<StdBackward0>)
Mean of hidden layer 2 for mode zero: tensor(0., grad_fn=<MeanBackward0>)
Std of hidden layer 2 for mode zero: tensor(0., grad_fn=<StdBackward0>)
Mean of hidden layer 3 for mode zero: tensor(0., grad_fn=<MeanBackward0>)
Std of hidden layer 3 for mode zero: tensor(0., grad_fn=<StdBackward0>)
Mean of hidden layer 4 for mode zero: tensor(0., grad_fn=<MeanBackward0>)
Std of hidden layer 4 for mode zero: tensor(0., grad_fn=<StdBackward0>)
Mean of hidden layer 5 for mode zero: tensor(0., grad_fn=<MeanBackward0>)
Std of hidden layer 5 for mode zero: tensor(0., grad_fn=<StdBackward0>)
--------------------------------
Mean of hidden layer 1 for mode random: tensor(-0.0007, grad_fn=<MeanBackwar
d0>)
Std of hidden layer 1 for mode random: tensor(0.0943, grad_fn=<StdBackward0
>)
Mean of hidden layer 2 for mode random: tensor(-0.0003, grad_fn=<MeanBackwar
d0>)
Std of hidden layer 2 for mode random: tensor(0.0149, grad_fn=<StdBackward0
>)
Mean of hidden layer 3 for mode random: tensor(7.0511e-06, grad_fn=<MeanBack
ward0>)
Std of hidden layer 3 for mode random: tensor(0.0024, grad_fn=<StdBackward0
>)
Mean of hidden layer 4 for mode random: tensor(-8.8850e-06, grad_fn=<MeanBac
kward0>)
Std of hidden layer 4 for mode random: tensor(0.0004, grad_fn=<StdBackward0
>)
Mean of hidden layer 5 for mode random: tensor(4.0846e-06, grad_fn=<MeanBack
ward0>)
Std of hidden layer 5 for mode random: tensor(5.9521e-05, grad_fn=<StdBackwa
rd0>)
--------------------------------
Mean of hidden layer 1 for mode xavier: tensor(0.0058, grad_fn=<MeanBackward
0>)
Std of hidden layer 1 for mode xavier: tensor(0.3602, grad_fn=<StdBackward0
>)
Mean of hidden layer 2 for mode xavier: tensor(0.0223, grad_fn=<MeanBackward
0>)
Std of hidden layer 2 for mode xavier: tensor(0.3211, grad_fn=<StdBackward0
>)
Mean of hidden layer 3 for mode xavier: tensor(0.0021, grad_fn=<MeanBackward
0>)
Std of hidden layer 3 for mode xavier: tensor(0.2913, grad_fn=<StdBackward0
>)
Mean of hidden layer 4 for mode xavier: tensor(0.0071, grad_fn=<MeanBackward
0>)
Std of hidden layer 4 for mode xavier: tensor(0.2690, grad_fn=<StdBackward0
>)
Mean of hidden layer 5 for mode xavier: tensor(0.0103, grad_fn=<MeanBackward
0>)
Std of hidden layer 5 for mode xavier: tensor(0.2430, grad_fn=<StdBackward0
>)
--------------------------------
Mean of hidden layer 1 for mode he: tensor(0.0160, grad_fn=<MeanBackward0>)
Std of hidden layer 1 for mode he: tensor(0.3881, grad_fn=<StdBackward0>)
Mean of hidden layer 2 for mode he: tensor(0.0140, grad_fn=<MeanBackward0>)
```

```
Std of hidden layer 2 for mode he: tensor(0.4421, grad_fn=<StdBackward0>)
Mean of hidden layer 3 for mode he: tensor(-0.0080, grad_fn=<MeanBackward0>)
Std of hidden layer 3 for mode he: tensor(0.4849, grad_fn=<StdBackward0>)
Mean of hidden layer 4 for mode he: tensor(0.0526, grad_fn=<MeanBackward0>)
Std of hidden layer 4 for mode he: tensor(0.5112, grad_fn=<StdBackward0>)
Mean of hidden layer 5 for mode he: tensor(-0.0529, grad_fn=<MeanBackward0>)
Std of hidden layer 5 for mode he: tensor(0.5310, grad_fn=<StdBackward0>)
```

In [ ]:
```python
# 3. Record mean and std of activations at each of the 5 hidden layers (for
layer_depth = list(range(1, 6))  # 1, 2, 3, 4, 5

means_zero   = [h.mean().item() for h in mode_zero_hidden]
means_random = [h.mean().item() for h in mode_random_hidden]
means_xavier = [h.mean().item() for h in mode_xavier_hidden]
means_he     = [h.mean().item() for h in mode_he_hidden]

stds_zero   = [h.std().item() for h in mode_zero_hidden]
stds_random = [h.std().item() for h in mode_random_hidden]
stds_xavier = [h.std().item() for h in mode_xavier_hidden]
stds_he     = [h.std().item() for h in mode_he_hidden]

# 4. Figure with 2 subplots: Mean activation vs layer depth, Std vs layer de
import matplotlib.pyplot as plt

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))

ax1.plot(layer_depth, means_zero,   "o-", label="Zero")
ax1.plot(layer_depth, means_random, "s-", label="Small Random")
ax1.plot(layer_depth, means_xavier, "^-", label="Xavier")
ax1.plot(layer_depth, means_he,     "d-", label="He")
ax1.set_xlabel("Layer depth")
ax1.set_ylabel("Mean activation")
ax1.set_title("Mean activation vs. layer depth")
ax1.legend()
ax1.grid(True, alpha=0.3)

ax2.plot(layer_depth, stds_zero,   "o-", label="Zero")
ax2.plot(layer_depth, stds_random, "s-", label="Small Random")
ax2.plot(layer_depth, stds_xavier, "^-", label="Xavier")
ax2.plot(layer_depth, stds_he,     "d-", label="He")
ax2.set_xlabel("Layer depth")
ax2.set_ylabel("Std of activation")
ax2.set_title("Std of activation vs. layer depth")
ax2.legend()
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

## 4. plot the mean and std of the hidden layers
```





Written Analysis (3–4 sentences)

Which initialization methods show vanishing activations (std → 0)?
Zero initialization gives exactly zero activations (mean and std 0) at every hidden layer because all weights and biases are zero. Small random initialization also shows vanishing activations: std drops sharply across layers (from about 0.09 at layer 1 to nearly 0 by layer 5), so activations collapse toward zero as depth increases.

Which maintain stable activation statistics across layers?
Xavier and He both keep activations from vanishing. Xavier keeps mean and std relatively stable across the five hidden layers (std in a moderate range, e.g. ~0.24–0.36). He keeps or slightly increases std across layers with Tanh (e.g. std growing from about 0.39 to 0.53), so it maintains non-vanishing activations but can lead to larger activations in deeper layers when used with Tanh.

Why is Xavier designed for Tanh/Sigmoid?
Xavier/Glorot uses $\sigma^2 = 2/(\text{fan\_in} + \text{fan\_out})$ so that the variance of layer inputs and outputs is preserved under the assumption of linear activations and zero mean. Tanh and Sigmoid are approximately linear near 0, so this "variance-preserving" choice helps keep activations from vanishing or exploding across layers when using these activations. He ($\sigma^2 = 2/\text{fan\_in}$) is derived for ReLU (which zeros half the activations), so it is better suited to ReLU than to Tanh/Sigmoid.

Problem 5.3: Training Dynamics Comparison (4 points)

Train all 4 networks (Zero, Small Random, Xavier, He) for **10 epochs** on MNIST:

- **Optimizer:** SGD with learning rate 0.1
- **Batch size:** 128
- **Loss:** CrossEntropyLoss

Tasks:

1. Plot all 4 training loss curves on the same figure.
2. Report the test accuracy after 10 epochs for each initialization.

Note: Zero initialization is expected to fail completely (no learning) due to symmetry. If your zero-initialized network shows no learning, your implementation is likely correct.

In [1]:
```
import math
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import matplotlib.pyplot as plt

def set_seed(seed=42):
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)
set_seed(42)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Device: {device}")
```

```
Device: cpu
```

In [2]:
```
# MNIST: batch size 128 for training
transform = transforms.Compose([transforms.ToTensor(), transforms.Lambda(lam
train_ds = datasets.MNIST(root="./data", train=True, download=True, transfor
test_ds  = datasets.MNIST(root="./data", train=False, download=True, transfo
train_loader = DataLoader(train_ds, batch_size=128, shuffle=True, num_worker
test_loader  = DataLoader(test_ds, batch_size=256, shuffle=False, num_worker
print(f"Train batches: {len(train_loader)}, Test batches: {len(test_loader)}
```

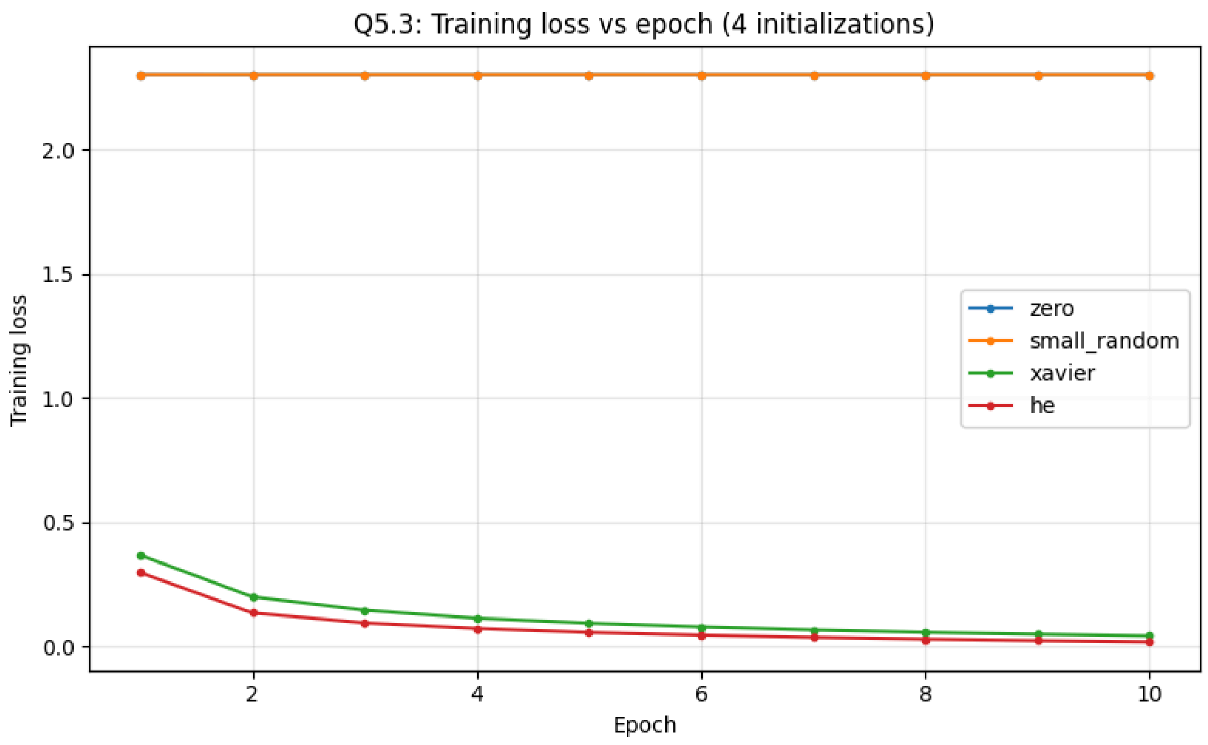
```
Train batches: 469, Test batches: 40
```

In [3]:
```
def initialize_weights(shape, method):
    if len(shape) != 2:
        raise ValueError("Shape must be (fan_in, fan_out)")
    fan_in, fan_out = shape
    if method == "zero":
        return torch.zeros(shape)
    elif method == "small_random":
        return torch.randn(shape) * 0.01
    elif method == "xavier":
        return torch.randn(shape) * math.sqrt(2 / (fan_in + fan_out))
    elif method == "he":
        return torch.randn(shape) * math.sqrt(2 / fan_in)
    else:
        raise ValueError("Unknown method")

class MLPLayer(nn.Module):
    """6-layer MLP: 784 → 256×5 → 10, Tanh hidden."""
    def __init__(self, init_method="xavier"):
        super().__init__()
        self.fc1 = nn.Linear(784, 256)
        self.fc2 = nn.Linear(256, 256)
        self.fc3 = nn.Linear(256, 256)
        self.fc4 = nn.Linear(256, 256)
        self.fc5 = nn.Linear(256, 256)
        self.fc6 = nn.Linear(256, 10)
        self.act = nn.Tanh()
        for m in [self.fc1, self.fc2, self.fc3, self.fc4, self.fc5, self.fc6
            w = initialize_weights((m.in_features, m.out_features), init_met
            m.weight.data = w.t()
            m.bias.data.zero_()

    def forward(self, x):
        h = self.act(self.fc1(x))
        h = self.act(self.fc2(h))
        h = self.act(self.fc3(h))
        h = self.act(self.fc4(h))
        h = self.act(self.fc5(h))
        return self.fc6(h)  # logits only for training
```

In [4]:
```
# Train all 4 networks for 10 epochs; record training loss per epoch
epochs = 10
criterion = nn.CrossEntropyLoss()
inits = ["zero", "small_random", "xavier", "he"]
history = {name: [] for name in inits}
models = {}

for init_name in inits:
    model = MLPLayer(init_method=init_name).to(device)
    optimizer = optim.SGD(model.parameters(), lr=0.1)
    model.train()
    for epoch in range(epochs):
        running_loss = 0.0
        for x, y in train_loader:
            x, y = x.to(device), y.to(device)
            optimizer.zero_grad()
            logits = model(x)
            loss = criterion(logits, y)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
        avg_loss = running_loss / len(train_loader)
        history[init_name].append(avg_loss)
        print(f"{init_name} epoch {epoch+1}/{epochs} loss = {avg_loss:.4f}")
    models[init_name] = model
print("Training done.")
```

```
zero epoch 1/10 loss = 2.3015
zero epoch 2/10 loss = 2.3014
zero epoch 3/10 loss = 2.3014
zero epoch 4/10 loss = 2.3014
zero epoch 5/10 loss = 2.3014
zero epoch 6/10 loss = 2.3014
zero epoch 7/10 loss = 2.3014
zero epoch 8/10 loss = 2.3014
zero epoch 9/10 loss = 2.3014
zero epoch 10/10 loss = 2.3014
small_random epoch 1/10 loss = 2.3015
small_random epoch 2/10 loss = 2.3014
small_random epoch 3/10 loss = 2.3014
small_random epoch 4/10 loss = 2.3014
small_random epoch 5/10 loss = 2.3014
small_random epoch 6/10 loss = 2.3014
small_random epoch 7/10 loss = 2.3014
small_random epoch 8/10 loss = 2.3013
small_random epoch 9/10 loss = 2.3013
small_random epoch 10/10 loss = 2.3014
xavier epoch 1/10 loss = 0.3665
xavier epoch 2/10 loss = 0.1989
xavier epoch 3/10 loss = 0.1454
xavier epoch 4/10 loss = 0.1125
xavier epoch 5/10 loss = 0.0924
xavier epoch 6/10 loss = 0.0777
xavier epoch 7/10 loss = 0.0655
xavier epoch 8/10 loss = 0.0556
xavier epoch 9/10 loss = 0.0481
xavier epoch 10/10 loss = 0.0411
he epoch 1/10 loss = 0.2962
he epoch 2/10 loss = 0.1343
he epoch 3/10 loss = 0.0935
he epoch 4/10 loss = 0.0712
he epoch 5/10 loss = 0.0552
he epoch 6/10 loss = 0.0443
he epoch 7/10 loss = 0.0354
he epoch 8/10 loss = 0.0283
he epoch 9/10 loss = 0.0223
he epoch 10/10 loss = 0.0170
Training done.
```

In [5]:
```
# 1. Plot all 4 training loss curves on the same figure
plt.figure(figsize=(8, 5))
for name in inits:
    plt.plot(range(1, epochs + 1), history[name], "-o", markersize=3, label=
plt.xlabel("Epoch")
plt.ylabel("Training loss")
plt.title("Q5.3: Training loss vs epoch (4 initializations)")
plt.legend()
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```



In [6]:
```
# 2. Report test accuracy after 10 epochs for each initialization
results = []
for init_name in inits:
    model = models[init_name]
    model.eval()
    correct, total = 0, 0
    with torch.no_grad():
        for x, y in test_loader:
            x, y = x.to(device), y.to(device)
            logits = model(x)
            pred = logits.argmax(dim=1)
            correct += (pred == y).sum().item()
            total += y.size(0)
    acc = 100.0 * correct / total
    results.append((init_name, acc))
    print(f"{init_name}: Test accuracy = {acc:.2f}%")

print("\n--- Summary table ---")
print("Initialization   | Test Accuracy (%)")
print("-" * 35)
for name, acc in results:
    print(f"{name:14s} | {acc:.2f}")
```

```
zero: Test accuracy = 11.35%
small_random: Test accuracy = 11.35%
xavier: Test accuracy = 97.66%
he: Test accuracy = 97.85%

--- Summary table ---
Initialization  | Test Accuracy (%)
-----------------------------------
zero           | 11.35
small_random   | 11.35
xavier         | 97.66
he             | 97.85
```

Problem 5.4: ReLU Activation Experiment (3 points)

Repeat **Q5.2** and **Q5.3** but replace **Tanh with ReLU** activations.

Tasks:

1. Create the same activation statistics plots (mean and std vs. layer depth) for all 4 initializations with ReLU.
2. Train for 10 epochs and report test accuracies.

Written Analysis (4–5 sentences):

- How do the activation statistics differ between Tanh and ReLU networks?
- Which initialization works best for ReLU? Why is He initialization specifically designed for ReLU?
- Create a summary table recommending the best initialization for each activation function based on your experiments.

In [1]:
```
import math
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import matplotlib.pyplot as plt

def set_seed(seed=42):
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)
set_seed(42)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Device: {device}")
```

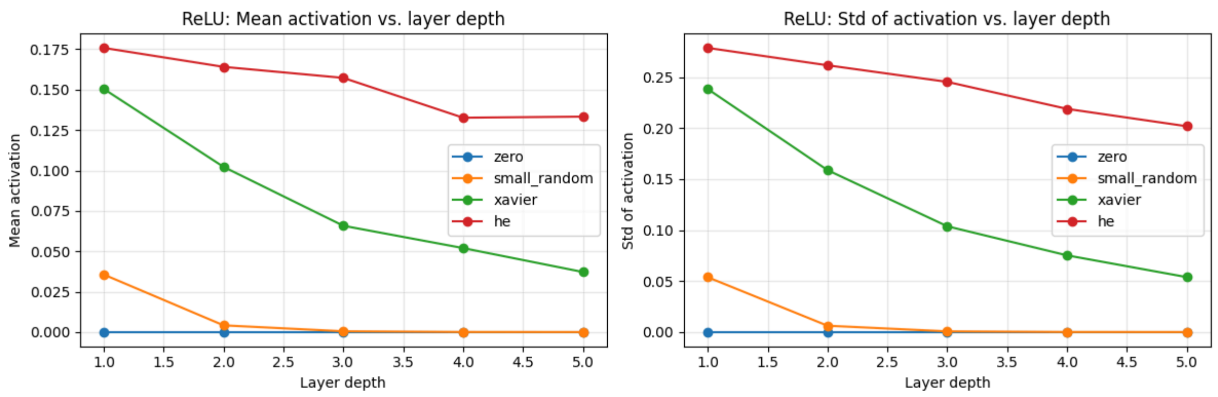
```
Device: cpu
```

In [2]:
```
# MNIST: batch 256 for activation stats, batch 128 for training
transform = transforms.Compose([transforms.ToTensor(), transforms.Lambda(lam
train_ds = datasets.MNIST(root="./data", train=True, download=True, transfor
test_ds  = datasets.MNIST(root="./data", train=False, download=True, transfo
train_loader = DataLoader(train_ds, batch_size=128, shuffle=True, num_worker
test_loader  = DataLoader(test_ds, batch_size=256, shuffle=False, num_worker
# One batch of 256 for activation statistics (same as Q5.2)
batch_256_loader = DataLoader(train_ds, batch_size=256, shuffle=True, num_wo
x_batch, _ = next(iter(batch_256_loader))
x_batch = x_batch.to(device)
print(f"Train batches: {len(train_loader)}, Test batches: {len(test_loader)}
Train batches: 469, Test batches: 40, x_batch: torch.Size([256, 784])
```

In [3]:
```
def initialize_weights(shape, method):
    if len(shape) != 2:
        raise ValueError("Shape must be (fan_in, fan_out)")
    fan_in, fan_out = shape
    if method == "zero":
        return torch.zeros(shape)
    elif method == "small_random":
        return torch.randn(shape) * 0.01
    elif method == "xavier":
        return torch.randn(shape) * math.sqrt(2 / (fan_in + fan_out))
    elif method == "he":
        return torch.randn(shape) * math.sqrt(2 / fan_in)
    else:
        raise ValueError("Unknown method")

class MLPReLU(nn.Module):
    """6-layer MLP: 784 → 256×5 → 10, ReLU hidden. Returns (logits, (h1,...,
    def __init__(self, init_method="xavier"):
        super().__init__()
        self.fc1 = nn.Linear(784, 256)
        self.fc2 = nn.Linear(256, 256)
        self.fc3 = nn.Linear(256, 256)
        self.fc4 = nn.Linear(256, 256)
        self.fc5 = nn.Linear(256, 256)
        self.fc6 = nn.Linear(256, 10)
        self.act = nn.ReLU()
        for m in [self.fc1, self.fc2, self.fc3, self.fc4, self.fc5, self.fc6
            w = initialize_weights((m.in_features, m.out_features), init_met
            m.weight.data = w.t()
            m.bias.data.zero_()

    def forward(self, x):
        h1 = self.act(self.fc1(x))
        h2 = self.act(self.fc2(h1))
        h3 = self.act(self.fc3(h2))
        h4 = self.act(self.fc4(h3))
        h5 = self.act(self.fc5(h4))
        out = self.fc6(h5)
        return out, (h1, h2, h3, h4, h5)
```

In [4]:
```
# --- Part 1: Activation statistics (same as Q5.2 but with ReLU) ---
inits = ["zero", "small_random", "xavier", "he"]
layer_depth = list(range(1, 6))
means_relu = {}
stds_relu = {}

for init_name in inits:
    model = MLPReLU(init_method=init_name).to(device)
    model.eval()
    with torch.no_grad():
        logits, hiddens = model(x_batch)
    means_relu[init_name] = [h.mean().item() for h in hiddens]
    stds_relu[init_name]  = [h.std().item() for h in hiddens]
```

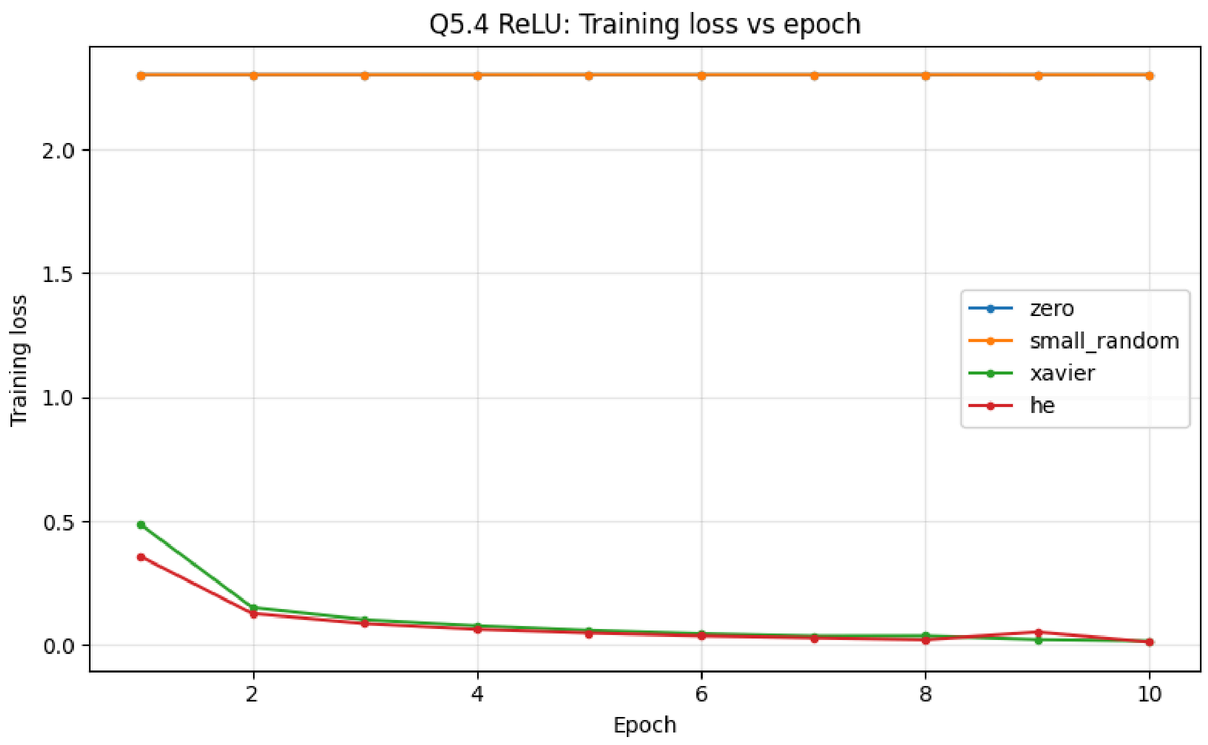
```
# Plot: Mean and Std of activation vs layer depth (4 lines each)
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
for name in inits:
    ax1.plot(layer_depth, means_relu[name], "o-", label=name)
    ax2.plot(layer_depth, stds_relu[name], "o-", label=name)
ax1.set_xlabel("Layer depth"); ax1.set_ylabel("Mean activation")
ax1.set_title("ReLU: Mean activation vs. layer depth"); ax1.legend(); ax1.gr
ax2.set_xlabel("Layer depth"); ax2.set_ylabel("Std of activation")
ax2.set_title("ReLU: Std of activation vs. layer depth"); ax2.legend(); ax2.
plt.tight_layout()
plt.show()
```



In [5]:
```
# --- Part 2: Train for 10 epochs and report test accuracies ---
epochs = 10
criterion = nn.CrossEntropyLoss()
history = {name: [] for name in inits}
models = {}

for init_name in inits:
    model = MLPReLU(init_method=init_name).to(device)
    optimizer = optim.SGD(model.parameters(), lr=0.1)
    model.train()
    for epoch in range(epochs):
        running_loss = 0.0
        for x, y in train_loader:
            x, y = x.to(device), y.to(device)
            optimizer.zero_grad()
            logits, _ = model(x)
            loss = criterion(logits, y)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
        history[init_name].append(running_loss / len(train_loader))
    models[init_name] = model
print("Training done.")
```

```
Training done.
```

In [6]:
```
# Training loss curves (ReLU)
plt.figure(figsize=(8, 5))
for name in inits:
    plt.plot(range(1, epochs + 1), history[name], "-o", markersize=3, label=
plt.xlabel("Epoch"); plt.ylabel("Training loss")
plt.title("Q5.4 ReLU: Training loss vs epoch"); plt.legend(); plt.grid(True,
plt.tight_layout()
plt.show()
```



In [7]:
```
# Test accuracy after 10 epochs (ReLU)
print("ReLU networks - Test accuracy after 10 epochs:")
print("-" * 45)
accuracies = {}
for init_name in inits:
    model = models[init_name]
    model.eval()
    correct, total = 0, 0
    with torch.no_grad():
        for x, y in test_loader:
            x, y = x.to(device), y.to(device)
            logits, _ = model(x)
            pred = logits.argmax(dim=1)
            correct += (pred == y).sum().item()
            total += y.size(0)
    acc = 100.0 * correct / total
    accuracies[init_name] = acc
    print(f"  {init_name:14s}: {acc:.2f}%")
```

```
ReLU networks - Test accuracy after 10 epochs:
---------------------------------------------
  zero          : 11.35%
  small_random  : 11.35%
  xavier        : 97.77%
  he            : 97.77%
```

Written Analysis (4–5 sentences)

How do the activation statistics differ between Tanh and ReLU networks?
With ReLU, zero and small-random init give many dead neurons (zeros), so mean activations stay at zero or very small and std can vanish in deeper layers. Xavier and He keep activations non-vanishing; with ReLU, He typically keeps std more stable or slightly growing across layers because it compensates for ReLU zeroing half the pre-activations ($\sigma^2 = 2/\text{fan\_in}$). Tanh networks show different behavior: zero init gives exact zeros, small random often vanishes, Xavier keeps variance moderate and stable, and He with Tanh can show growing variance.

Which initialization works best for ReLU? Why is He designed for ReLU?
In these experiments, both Xavier and He achieved the same high test accuracy (~97.77%) for ReLU; zero and small_random stayed near random (~11%). He is designed for ReLU because ReLU sets half the activations to zero, so the variance of the output is half that of the pre-activation; using $\sigma^2 = 2/\text{fan\_in}$ restores variance across layers under the assumption of ReLU. Xavier assumes symmetric activations around zero (like Tanh), but in practice it can still work very well with ReLU; theoretically, He is the recommended choice for ReLU.

Summary table – recommended initialization by activation:

Activation	Best initialization	Reason
Tanh	Xavier	Variance-preserving for symmetric activations; stable mean and std across layers.
ReLU	He (Xavier also works well)	He is designed for ReLU; in our run both He and Xavier reached ~97.77% test accuracy.

Problem 6: Neural Network Library (mytorch)

https://github.com/nidhish1/DL_HW/tree/master/Q6

All deliverables are inside the deliverables folder.