CSCI 6960/4140: Homework 4

Assigned Friday March 20, 2026. Due by 11:59pm ET Tuesday April 3, 2026.

Objective of this assignment. This assignment builds understanding of neural network training through a deliberate progression of three stages.

- Problem 1 makes the mechanics concrete: you implement and train a two-layer network entirely from scratch in NumPy, verify your gradients, compare three optimizers, and probe the geometry of the loss landscape around the solution you find.
- Problem 2 uses that concrete experience to ask structural questions: why does initialization matter? You prove that symmetry cannot be broken by gradient descent and then compare He and Glorot initialization empirically in a deep network.
- Problem 3 confronts the problems that persist even with good initialization: gradient signal can vanish or explode with depth. You observe this empirically for sigmoid and ReLU networks, track dead neurons over training, and then test whether the architectural remedies from Lecture 16—residual connections and batch normalization—actually fix these pathologies.

Instructions. Submit a well-structured Jupyter notebook with all code, outputs, plots, and requested discussion/derivations. Write documented, well-structured, readable code. Submit the notebook with all outputs already shown; the TA will not re-run it to produce them. Label all plots: axes (with units if applicable), legends, and titles; use differentiating line styles for different series; choose scales appropriately.

Academic Honesty and AI Usage. *Your submitted work must reflect your own understanding and effort.* Use of AI systems to generate or debug code, derivations, or analysis defeats the learning objectives of this assignment, undermines assessment integrity, is unfair to other students, and harms your learning.

Authorship Statement (Required). At the top of your notebook, you must include the following statement verbatim:

> "I completed this work without assistance from AI tools. All code, derivations, analysis, and discussion are my own."

Submissions that omit this statement *will be treated as late submissions*.

Verification of Authorship. The instructor and TA may conduct oral spot-checks or individual explanations to confirm authorship. These may be random, non-random, or triggered by discrepancies between in-class performance and submitted work. If significant gaps in understanding are observed, the assignment grade may be reduced and, in severe cases, treated as an academic honesty violation.

Notation. Throughout this assignment, layer indices appear as plain superscripts: W^s, a^s, o^s, b^s. The symbol $\ell(\cdot)$ is reserved exclusively for the per-sample loss function. Unparenthesized superscripts also denote matrix powers or transposes. For a vector v, $\|v\|_2$ denotes the Euclidean norm; for a matrix A, $\|A\|_F$ denotes the Frobenius norm and $\|A\|_2$ the spectral norm. The notation $\mathrm{diag}(v)$ denotes the diagonal matrix with v on its diagonal, and $\odot$ denotes elementwise multiplication.

Weight storage convention. In the mathematical formulation of Lecture 14, we write the affine map at layer s as $a^s = W^s o^{s-1} + b^s$, where $W^s \in \mathbb{R}^{n_s \times n_{s-1}}$ (rows = neurons in layer s). In code, however, PyTorch and NumPy conventionally store weight matrices in *transposed* form: the array `W` has shape $(\texttt{d_in},\ \texttt{d_out}) = (n_{s-1},\ n_s)$, so the forward pass is $Z = X @ W + b$ (matrix multiplication on the right). **All code in this assignment uses the** $(\texttt{d_in}, \texttt{d_out})$ **storage convention.** When the problem statement says "rows of W^s" (the mathematical matrix), this corresponds to *columns* of the code array `W`.

Code organization. Your notebook should be organized into clearly separated sections. Define reusable functions (forward pass, backward pass, optimizer steps) early and call them from experiment cells—do not duplicate logic. Training loops, data loading, and evaluation should live directly in experiment cells so you can inspect and modify them easily. A clean organization might look like:

Section	Contents
`Setup`	imports, data loading, constants
`Model`	`forward`, `backward`, `cross_entropy_loss`
`Optimizers`	`sgd_step`, `adam_step`, `adamw_step`
`Utilities`	batching, gradient check, plotting helpers
`Training from Scratch`	experiment cells for 1(a)–(c)
`Initialization`	experiment cells for 2(a)–(c)
`Gradient Flow`	experiment cells for 3(a)–(d)

Autograd policy. **Training from Scratch** (1(a)–(c)) must be implemented in pure NumPy—no `torch.autograd` or any other automatic differentiation. The 40-layer experiments in **Initialization (c)** and **Gradient Flow (a)–(d)** require PyTorch (with `torch.autograd`).

1 Training a Network from Scratch [35 pts]

Before analyzing what can go wrong with initialization or depth, you need to know what training feels like from the inside: what the forward pass computes, how gradients are assembled from the chain rule, and how different optimizers navigate the nonconvex loss landscape. This problem builds that operational understanding. The network you implement here will also serve as the experimental platform for the Initialization problem.

Architecture. A single hidden layer of width 256 with ReLU activations: $784 \to 256 \to 10$, trained with categorical cross-entropy loss.

Initialization. Use He initialization: $W^s_{ij} \sim \mathcal{N}(0,\ 2/n_{s-1})$, where n_{s-1} is the fan-in of layer s. Set all biases to zero.

Data. Load MNIST via `torchvision.datasets.MNIST` (data loading only; no autograd). Use the standard 60k/10k train/test split and set aside 10k of the training set as a validation set (50k train / 10k val / 10k test).

Start here: data loading scaffold. The following cell loads MNIST, creates the train/val/test split with a fixed random seed, and initializes the parameter dictionary. Complete this cell *before* implementing any functions.

Why fix the seed? `np.random.default_rng(seed=0)` produces the same sequence of random numbers on every run, on any machine. Fixing it means your train/val split and weight initialization are identical each time the notebook is executed, making your results reproducible and your gradient check deterministic.

```
import numpy as np
import torch
from torchvision import datasets, transforms

# -- reproducibility ------------------------------------------------------
SEED = 0
rng  = np.random.default_rng(SEED)

# -- load MNIST (normalize to [0, 1]) -------------------------------------
transform = transforms.ToTensor()
mnist_train_full = datasets.MNIST('.', train=True,  download=True,
                                  transform=transform)
mnist_test_ds    = datasets.MNIST('.', train=False, download=True,
                                  transform=transform)

X_all  = mnist_train_full.data.numpy().reshape(-1, 784) / 255.0
y_all  = mnist_train_full.targets.numpy()
X_test = mnist_test_ds.data.numpy().reshape(-1, 784) / 255.0
y_test = mnist_test_ds.targets.numpy()

# -- reproducible 50k / 10k split -----------------------------------------
idx = rng.permutation(60_000)
X_train, y_train = X_all[idx[:50_000]], y_all[idx[:50_000]]
X_val,   y_val   = X_all[idx[50_000:]], y_all[idx[50_000:]]

N_TRAIN, D_IN, N_CLASSES, HIDDEN = 50_000, 784, 10, 256

# -- He initialization helper ---------------------------------------------
def he_init(d_in, d_out):
    return rng.standard_normal((d_in, d_out)) * np.sqrt(2.0 / d_in)

params = {
    'W1': he_init(D_IN, HIDDEN),
    'b1': np.zeros((1, HIDDEN)),
    'W2': he_init(HIDDEN, N_CLASSES),
    'b2': np.zeros((1, N_CLASSES)),
}
```

Throughout this problem, pass `params` as the parameter argument to all functions. The keys and

shapes are:

Key	Shape	Description
`'W1'`	$(784, 256)$	first-layer weights
`'b1'`	$(1, 256)$	first-layer biases
`'W2'`	$(256, 10)$	output-layer weights
`'b2'`	$(1, 10)$	output-layer biases

(a) **[14 pts] NumPy implementation.**

Goal: build a complete forward–backward pipeline from scratch to internalize how each quantity flows through the computation graph.

Below is the scaffold showing *how these functions will be called* in the experiment cells. Implement each function using NumPy array operations only (no loops over individual examples). **Do not use PyTorch autograd at any point in this problem.**

```
# -- gradient check (run this before any training) ------------------------
B_check = 32
X_chk   = X_train[:B_check]
Y_chk   = np.eye(N_CLASSES)[y_train[:B_check]]   # one-hot labels

cache  = forward(X_chk, params)
_      = cross_entropy_loss(cache['Phat'], Y_chk)
grads  = backward(X_chk, Y_chk, cache, params)

h = 1e-5;  max_rel_err = 0.0
for _ in range(20):
    key     = rng.choice(['W1', 'b1', 'W2', 'b2'])
    idx_f   = rng.integers(0, params[key].size)
    idx_nd  = np.unravel_index(idx_f, params[key].shape)
    orig    = params[key][idx_nd]

    params[key][idx_nd] = orig + h
    Lp = cross_entropy_loss(forward(X_chk, params)['Phat'], Y_chk)
    params[key][idx_nd] = orig - h
    Lm = cross_entropy_loss(forward(X_chk, params)['Phat'], Y_chk)
    params[key][idx_nd] = orig

    g_an   = grads['d' + key][idx_nd]
    g_fd   = (Lp - Lm) / (2 * h)
    denom  = max(abs(g_an), abs(g_fd), 1e-8)
    max_rel_err = max(max_rel_err, abs(g_an - g_fd) / denom)

print(f"Max relative gradient error: {max_rel_err:.2e}")
# Expected: 1e-8 to 1e-7.  Values above 1e-5 indicate a bug.
```

Now implement the three functions:

- `forward(X, params) -> cache` — given a batch $X \in \mathbb{R}^{B \times 784}$ and the parameter dictionary, returns a `cache` dictionary containing:

Key	Shape	Computation
`'Z1'`	$(B, 256)$	$a^1 = X \mathbin{@} W^1 + b^1$
`'A1'`	$(B, 256)$	$o^1 = \mathrm{ReLU}(a^1)$
`'Z2'`	$(B, 10)$	$a^2 = o^1 \mathbin{@} W^2 + b^2$
`'Phat'`	$(B, 10)$	$\hat{P} = \mathrm{softmax}(a^2)$, applied row-wise

Use a numerically stable softmax: shift each row of a^2 by its maximum before exponentiating.

- `backward(X, Y, cache, params) -> grads` — given the batch $X \in \mathbb{R}^{B \times 784}$, one-hot labels $Y \in \mathbb{R}^{B \times 10}$, the forward `cache`, and `params`, returns a gradient dictionary $\nabla_\theta f$ with keys and shapes matching the parameter dictionary:

Key	Shape
`'dW2'`	$(256, 10)$
`'db2'`	$(1, 10)$
`'dW1'`	$(784, 256)$
`'db1'`	$(1, 256)$

- `cross_entropy_loss(Phat, Y) -> float` — returns the mean categorical cross-entropy over the batch:

$$f = -\frac{1}{B} \sum_{i=1}^{B} \sum_{k=1}^{10} Y_{ik} \log(\hat{P}_{ik} + \varepsilon), \qquad \varepsilon = 10^{-8}.$$

Gradient check (required before proceeding). Run the scaffold cell above. Report the maximum relative error. You should expect values in the range 10^{-8} to 10^{-7}. Values above 10^{-5} indicate a bug. **Do not proceed to (b) until your gradient check passes.**

(b) **[11 pts] Optimizer comparison.**

Goal: develop intuition for how optimizer design choices affect convergence speed and solution quality on the same nonconvex problem.

Train the network for 20 epochs with batch size 128. Implement and compare three optimizers, each applied to an independently initialized copy of the network. Each optimizer should follow a unified interface:

```
def optimizer_step(params, grads, state, lr, **kwargs) -> (params, state)
```

where `state` is a dictionary holding any running statistics (empty for SGD; momentum estimates for Adam/AdamW). The three optimizers:

- **SGD:** $\theta_t = \theta_{t-1} - \eta\, g_t$, with $\eta = 0.1$.
- **Adam:** $\eta = 10^{-3}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 10^{-8}$.

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1)\, g_t, \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2)\, g_t^2, \\ \hat{m}_t &= m_t / (1 - \beta_1^t), \quad \hat{v}_t = v_t / (1 - \beta_2^t), \\ \theta_t &= \theta_{t-1} - \eta\, \hat{m}_t \,/\, (\sqrt{\hat{v}_t} + \varepsilon). \end{aligned}$$

All squaring, square roots, and divisions are elementwise.

- **AdamW:** identical to Adam, except weight decay is applied directly to the weights before the Adam step:

$$\theta_t = (1 - \eta\lambda)\,\theta_{t-1} - \eta\,\hat{m}_t \,/\, (\sqrt{\hat{v}_t} + \varepsilon),$$

with $\lambda = 10^{-4}$. Do *not* apply weight decay to bias parameters.

Note: AdamW is **not** equivalent to using L_2 regularization with standard Adam.

Implementation note: Adam and AdamW share most of their logic. Factor the shared computation (momentum update, bias correction, adaptive step) into a common function. The only difference is the weight-decay line: AdamW applies $(1 - \eta\lambda)$ shrinkage to weight parameters before the adaptive step; Adam does not.

Training. Write the training loop *directly in a notebook cell*—do not wrap it in a function. This keeps the loop visible and easy to inspect. For each optimizer, reinitialize the parameters (call `he_init` again with `rng`), then loop over epochs: shuffle the training data at the start of each epoch (using `rng.permutation`), iterate over minibatches of size 128, call `forward`, `cross_entropy_loss`, `backward`, and the optimizer step. After each epoch, evaluate on the validation set; after the final epoch, evaluate on the test set. Record per-epoch training loss and validation accuracy in lists for plotting.

Deliverables. Plot training loss and validation accuracy versus epoch for all three on the same axes (two plots). Report the final test accuracy for each. Fill in the table:

Optimizer	Key design change vs. Adam	Final test acc.	Epochs to 95% val. acc.
SGD	(baseline: single scalar step size)	?	?
Adam	element-wise adaptive LR via $\hat{v}_t$	?	?
AdamW	weight decay decoupled from gradient	?	?

If 95% validation accuracy is not reached within 20 epochs, report "> 20".

In 3–4 sentences, explain why adding $\lambda\theta$ to the gradient (standard Adam weight decay) is *not* equivalent to the direct multiplicative decay in AdamW, and when this difference matters most.

(c) **[10 pts] The loss landscape.**

Goal: probe whether the nonconvexity of the loss surface creates practical difficulties for the overparameterized network you just trained.

Having successfully trained the network, probe how sensitive the final solution is to randomness and perturbation. Neural network objectives are nonconvex, yet overparameterized networks often exhibit surprisingly flat basins around the solutions found by gradient descent. The following experiments provide evidence—and raise questions—about the local geometry of those solutions.

(i) **Before running this experiment,** state a hypothesis: do you expect the 5 runs to converge to solutions with similar or very different test accuracy? Why?

Train the same network 5 times from independent random seeds using the optimizer that achieved the highest test accuracy in (b). Report the mean and standard deviation of final test accuracy. Does the spread surprise you given that the objective is nonconvex? Explain what the result suggests about the loss landscape in terms of the overparameterization ratio (number of parameters vs. number of training examples).

(ii) Compute the *full-batch* gradient norm $\|\nabla_\theta f(\theta_T)\|_2$ at the end of one training run, where the loss is computed over the *entire* training set in a single forward and backward pass (not as an average of per-minibatch gradient norms). Here $\nabla_\theta f$ denotes the gradient with respect to all parameters concatenated into a single vector, so $\|\cdot\|_2$ is the Euclidean norm of that vector. (Later problems report per-layer Frobenius norms instead.)

This requires passing the full 50k training set as a single batch. If memory is an issue, you may accumulate gradients over minibatches, but report the norm of the *average* gradient.

Is it zero? What does this tell you about the first-order stationarity condition from Lecture 13, and what does the nonconvex convergence theorem guarantee about the iterate found by (S)GD?

(iii) Perturb the final weights: $\theta' = \theta_T + r \cdot z$, where $z \sim \mathcal{N}(0, I)$ and $r \in \{0.001, 0.01, 0.1\}$.

Note: Since z has approximately $p \approx 200{,}000$ entries, $\|z\| \approx \sqrt{p} \approx 450$, so the total perturbation norm is approximately $r\sqrt{p}$—much larger than r.

For each r, report the change in loss and test accuracy. Explain what the sensitivity (or insensitivity) of the solution to perturbation tells you about the local curvature of the minimum found, specifically in terms of the Hessian eigenvalue spectrum.

2 Initialization [28 pts]

Lecture 16 established the theory of weight initialization: the forward variance propagation argument, the backward gradient norm preservation via exchangeability, and the derivation of both Xavier and He initialization. This section does not re-derive that theory. Instead, it asks you to prove that random initialization is a *mathematical necessity* (not merely a good idea) and then to verify the lecture's predictions experimentally in a deep network.

(a) **[10 pts] The symmetry collapse theorem.**

Goal: prove that uniform initialization is an absorbing state for gradient-based training—symmetry, once present, can never be broken.

Random initialization is sometimes described as "breaking symmetry." This problem shows that symmetry collapse is not just an empirical concern—it is a mathematical certainty for any deterministic gradient-based optimizer when initialization is uniform.

Consider any MLP. Suppose all neurons in hidden layer s are initialized identically: every row of W^s equals the same vector $w^\top$, and every entry of b^s equals the same scalar c.

Assumption: The optimizer is deterministic and applies per-element updates (no injected noise such as dropout or stochastic perturbation of the weights themselves).

(i) Show that all pre-activations in layer s are equal ($a_i^s = a_j^s$ for all i, j), and hence all activations are equal.

(ii) For any differentiable loss f, show that all rows of $\nabla_{W^s} f$ are identical and all entries of $\nabla_{b^s} f$ are identical. *Hint:* use the backpropagation recurrence from Lecture 14 and the fact established in (i).

(iii) Conclude that after one gradient descent step with any step size, all rows of the updated W^s remain identical. Then argue by induction that no deterministic gradient-based optimizer with per-element updates (including Adam and AdamW, whose update direction for θ_i depends only on the history of $\partial f / \partial \theta_i$) can ever break this symmetry.

State explicitly: what is the *effective* width of a network trained from such an initialization, regardless of its nominal width n_s?

Sanity check: What happens if you initialize all weights to zero? Does this constitute a special case of the symmetry collapse above, and if so, is it worse?

(b) **[8 pts] Empirical symmetry failure.**

Goal: confirm the symmetry collapse theorem experimentally.

State your hypothesis before running the experiment: for each initialization below, predict whether training will reduce the loss, and whether test accuracy will approach that of He initialization.

Using your implementation from the first problem, train the MNIST network under three initializations. **Train a fresh, independently initialized copy for each experiment.**

(i) **He initialization** (you've already done this in the first problem, no need to reprise)

(ii) **All-zeros:** $W^s = 0$, $b^s = 0$.

(iii) **Constant-row:** For each layer s, draw a single random vector $w \sim \mathcal{N}(0, (2/n_{s-1})\, I)$ and set every *row* of the mathematical W^s equal to $w^\top$. Use a different draw for each layer.
In code: Since your weight array has shape $(\texttt{d_in}, \texttt{d_out})$, "every row of W^s" corresponds to every *column* of the code array. Concretely, draw $w \in \mathbb{R}^{n_{s-1}}$ and tile it across all n_s columns:

```
w = rng.standard_normal((d_in, 1)) * np.sqrt(2.0 / d_in)
W = np.tile(w, (1, d_out))         # shape (d_in, d_out)
```

Plot training loss vs. epoch for all three on the same axes. For (ii) and (iii), report the final test accuracy and explain the failures and successes in terms of the theorem in (a).

(c) **[10 pts] He vs. Glorot initialization in a deep ReLU network.**

Goal: verify experimentally that Glorot initialization causes signal attenuation in a deep ReLU network, as predicted by the variance propagation analysis from Lecture 16.

Recall from Lecture 16 that He initialization preserves forward activation variance in ReLU networks ($v_{s+1} = v_s$), while Glorot initialization halves it at every ReLU layer ($v_{s+1} = \frac{1}{2} v_s$). In a 40-layer network, this predicts that Glorot will attenuate the activation second moment by a factor of $2^{-39} \approx 10^{-12}$ relative to He.

Implement a **40-layer** ReLU network with uniform hidden width 128 for MNIST ($784 \to 128 \to \cdots \to 128 \to 10$, i.e., 39 hidden layers plus the output layer) **using PyTorch**. Train two copies: one with He and one with Glorot initialization, otherwise identical (SGD with $\eta = 0.1$, 10 epochs, batch size 128). You will reuse this 40-layer architecture in the next problem.

State your hypothesis explicitly before reporting results: based on the analysis from Lecture 16, predict the qualitative shape of the per-layer activation second moment plot for each initialization.

Record and plot:

- Per-layer activation second moment $\mathbb{E}[(a_j^{(s)})^2]$ at initialization (before any gradient step), computed over a minibatch of 256 training examples. Plot this for both initializations on the same axes with layer index on the horizontal axis.
- Per-layer gradient Frobenius norm $\|\nabla_{W^s} f\|_F$, averaged over all minibatches of epoch 1.
- Training loss vs. epoch (both on the same axes).
- Final test accuracy.

Explain your observations in terms of the Lecture 16 variance propagation results.

Setup scaffold (run once before both 2(c) and 3(a)–(d) experiments). The following cell sets up standard PyTorch data loading for the deep-network experiments. Fix the seed for reproducibility.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

TORCH_SEED = 42

transform = transforms.Compose([
    transforms.ToTensor(),          # converts to [0,1] float32
    transforms.Lambda(lambda x: x.view(-1))   # flatten 28x28 -> 784
])
```

```
train_ds = datasets.MNIST('.', train=True,  download=True, transform=transform)
test_ds  = datasets.MNIST('.', train=False, download=True, transform=transform)

# -- reproducible 50k / 10k val split (matches Section 1) --------------------
train_sub, val_sub = torch.utils.data.random_split(
    train_ds, [50_000, 10_000],
    generator=torch.Generator().manual_seed(TORCH_SEED))

train_loader = DataLoader(train_sub, batch_size=128, shuffle=True,
    generator=torch.Generator().manual_seed(TORCH_SEED))
val_loader   = DataLoader(val_sub,   batch_size=1000, shuffle=False)
test_loader  = DataLoader(test_ds,   batch_size=1000, shuffle=False)

WIDTHS_40 = [784] + [128] * 39 + [10]
```

The network class (you implement this). Define `DeepMLP` in its own cell. The class must support:

- A configurable list of layer widths `widths` (e.g. `WIDTHS_40`).
- An `activation` argument (`'relu'` or `'sigmoid'`).
- An `init` argument (`'he'` or `'glorot'`), applied via `nn.init.kaiming_normal_` or `nn.init.xavier_normal_`.
- A `residual` argument (`True` or `False`), default `False`. When `True`, add a skip connection every 2 layers (see problem 3(c)).
- A `use_batchnorm` argument (`True` or `False`), default `False`. When `True`, insert `nn.BatchNorm1d` before each activation (see problem 3(d)).
- A `forward(x, return_activations=False)` method. When `return_activations=True`, return `(logits, acts)` where `acts` is a list of post-activation tensors $[o^1, o^2, \ldots, o^{L-1}]$ (detached from the graph).

```
class DeepMLP(nn.Module):
    def __init__(self, widths, activation='relu', init='he',
                 residual=False, use_batchnorm=False):
        super().__init__()
        # TODO: build self.layers as an nn.ModuleList of nn.Linear layers,
        # optionally self.bns as nn.ModuleList of nn.BatchNorm1d layers,
        # apply the chosen initialization to each layer's weight,
        # zero-initialize all biases.
        # Store self.residual and self.use_batchnorm for forward().
        ...

    def forward(self, x, return_activations=False):
        # TODO: run the forward pass; if return_activations,
        # collect and return post-activation tensors.
        # If self.residual: add skip connections every 2 layers
        #   (with a learned projection if dimensions change).
        # If self.use_batchnorm: apply BN before the activation.
        ...
```

Test this class thoroughly before proceeding. `DeepMLP` is used in every experiment from here through the end of the assignment. Verify that it runs without error for each combination of `activation`, `init`, `residual`, and `use_batchnorm` that you will need; check that `return_activations=True` returns the correct number of tensors with the expected shapes.

Training cell (write this directly in the experiment cell, not in a function). For each initialization (`'he'` and `'glorot'`):

```
torch.manual_seed(TORCH_SEED)
model     = DeepMLP(WIDTHS_40, activation='relu', init='he')  # change init
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
criterion = nn.CrossEntropyLoss()

# -- activation second moments at initialization (before any gradient step) --
model.eval()
with torch.no_grad():
    X_sample, _ = next(iter(DataLoader(train_ds, batch_size=256)))
    _, acts = model(X_sample, return_activations=True)
    init_moments = [(a**2).mean().item() for a in acts]

train_loss_hist, test_acc_hist, grad_norm_hist = [], [], []

for epoch in range(10):
    # -- training pass ------------------------------------------------------
    model.train()
    total_loss, n_batches = 0.0, 0
    for xb, yb in train_loader:
        # TODO: forward, loss, zero_grad, backward, optimizer step
        ...
        total_loss += loss.item(); n_batches += 1
    train_loss_hist.append(total_loss / n_batches)

    # -- per-layer gradient Frobenius norms ---------------------------------
    # Note: model.eval() uses running BN statistics (not batch statistics).
    # This is intentional: it makes gradient norms comparable across epochs
    # and avoids batch-size artifacts, but the values will differ slightly
    # from gradients computed during training.
    model.eval()
    grad_sums = [0.0] * len(model.layers); n_grad = 0
    for xb, yb in train_loader:
        logits = model(xb)
        loss   = criterion(logits, yb)
        model.zero_grad(); loss.backward()
        for i, layer in enumerate(model.layers):
            grad_sums[i] += layer.weight.grad.norm(p='fro').item()
        n_grad += 1
    grad_norm_hist.append([g / n_grad for g in grad_sums])

    # -- test accuracy ------------------------------------------------------
    # TODO: compute and append to test_acc_hist
```

...

Reading out the quantities you need:

- **Activation second moments at init:** `init_moments[s]` gives $\mathbb{E}[(o_j^s)^2]$ at layer s before any training. Plot against layer index.
- **Per-layer gradient norms:** `grad_norm_hist[epoch][s]` gives $\|\nabla_{W^s} f\|_F$ averaged over the training set at the end of that epoch. For the epoch 1 gradient norm plot, use `grad_norm_hist[0]`.

3 Gradient Flow in Deep Networks [37 pts]

Goal: observe how gradient signal degrades with depth, track a concrete pathology (dead neurons), and test whether the architectural remedies from Lecture 16—residual connections and batch normalization—fix the problem in practice.

Lecture 16 established that the gradient at layer s satisfies the spectral norm bound

$$\|\nabla_{o^s} f\|_2 \;\le\; \prod_{i=s+1}^{L} \underbrace{\|W^i\|_2 \cdot \|\sigma'\|_\infty}_{=:\, m_i} \cdot \|\nabla_{o^L} f\|_2,$$

where m_i is the per-layer multiplier. For sigmoid, $\|\sigma'\|_\infty \le 1/4$ drives m_i well below 1; for ReLU, $\|\sigma'\|_\infty \le 1$ and the bound is tighter. This section tests these predictions empirically and goes beyond them.

Throughout, use the 40-layer, width-128 MNIST architecture and `DeepMLP` class from problem 2(c).

(a) **[10 pts] Sigmoid vs. ReLU: gradient attenuation in a deep network.**

This experiment tests whether the spectral bound from Lecture 16 is reflected in actual training dynamics, and whether ReLU mitigates gradient attenuation relative to sigmoid.

State your hypothesis: based on the Lecture 16 bound, predict the qualitative shape of the gradient-norm-vs.-layer plot for each activation function. For sigmoid with $\|W^s\|_2 \approx 1$, the bound gives $\|\nabla_{o^s} f\|_2 \lesssim (1/4)^{L-s} \|\nabla_{o^L} f\|_2$. At $L = 40$, this is an attenuation of $(1/4)^{39} \approx 10^{-24}$ from the output to the first layer. For ReLU, the worst-case bound is $\approx 1^{39} = 1$.

Using the 40-layer architecture, train two copies with the same He initialization—one with sigmoid activations, one with ReLU—for 10 epochs using SGD with learning rate $\eta = 0.1$ and batch size 128.

Use He initialization (`kaiming_normal_`) for **both** networks, even though He initialization was derived for ReLU. Observing the sigmoid network's behavior under He initialization is part of the exercise.

After each epoch, record $\|\nabla_{W^s} f\|_F$ for each layer $s = 1, \ldots, 40$, averaged over one full pass through the training set (use the training cell structure from problem 2(c): accumulate `layer.weight.grad.norm(p='fro')` over minibatches and divide by the number of batches). Produce one plot per activation function showing gradient norm vs. layer index at epochs 1, 5, and 10. Explain the qualitative difference between the two plots in terms of the Lecture 16 spectral bound.

(b) **[7 pts] Dead neuron tracking.**

Goal: observe that dead ReLU neurons accumulate over training and quantify the effect on the network's effective capacity.

A ReLU neuron i in layer s is **dead** if $o_i^s(x) = 0$ for *every* training example x. A dead neuron receives zero gradient on its own weight row ($\nabla_{W_i^s} f = 0$ whenever $o_i^s = 0$ for all inputs in a minibatch), so it cannot recover through its own gradient signal.

(i) **Instrumentation.** Add the following measurement to your 40-layer ReLU training loop (He init, same hyperparameters as part (a)). After each epoch:

i. Pass the *entire training set* through the network (in minibatches, with `torch.no_grad()`) using `return_activations=True`. For each hidden layer s, maintain a boolean mask `ever_active[s]` of shape `(layer_width,)`, initialized to `False`. For each minibatch, set `ever_active[s] |= (acts[s] > 0).any(dim=0)`.

ii. A neuron is dead if it was never active across the entire training set: `dead[s] = ~ever_active[s]`.

iii. Record the *dead fraction*: (number of dead neurons at layer s) / (layer width).

(ii) **Plots and analysis.** Produce two plots:

- **Dead fraction vs. layer index** at epochs 1, 5, and 10 (three curves on the same axes). Are later layers more affected than earlier layers?
- **Total dead neuron count vs. epoch** (summed over all hidden layers). Does the count tend to increase over training? Is it strictly monotone, or do you observe any epochs where it decreases?

In 3–4 sentences, explain: (i) why dead neurons *tend* to stay dead (what is the mechanism that makes recovery unlikely?); (ii) what the accumulation of dead neurons implies about the *effective width* of the network as training progresses; and (iii) if the count ever decreases, explain what could cause a dead neuron to revive.

(c) **[12 pts] Residual connections.**

Goal: test whether skip connections rescue gradient flow in a deep network, as predicted by the additive path-sum analysis from Lecture 16.

Lecture 16 showed that a residual layer $o^{s+1} = o^s + F(o^s)$ decomposes the gradient into an identity path (which passes through without attenuation) plus a main-branch path. Over k residual layers, the gradient is a sum of 2^k terms, and the short paths—which bypass most Jacobian factors—ensure that gradient signal reaches every layer regardless of activation saturation.

(i) **Implementation.** If you have not already done so, implement the `residual=True` mode in your `DeepMLP` class. When enabled:

- Add an identity skip connection every 2 layers: if layers are indexed $1, 2, \ldots, L$, then after layers $2, 4, 6, \ldots$ the output is $o^{s+1} = \sigma(W^{s+1}o^s + b^{s+1}) + o^{s-1}$. Concretely, layer 2 receives a skip from the input to layer 1, layer 4 from the output of layer 2, layer 6 from the output of layer 4, and so on.
- When the skip source and destination have different dimensions (e.g., the first hidden layer where $784 \to 128$), use a learned linear projection on the skip branch: $o^{s+1} = \sigma(W^{s+1}o^s + b^{s+1}) + P\,o^{s-1}$, where P is an `nn.Linear` without bias. Initialize P with He initialization.

(ii) **Experiment.** Train the 40-layer architecture in three configurations, all with He initialization, SGD with $\eta = 0.1$, 10 epochs, batch size 128:

Config	Activation	Residual	Label in plots
A	ReLU	No	`relu`
B	sigmoid	No	`sigmoid`
C	sigmoid	Yes	`sigmoid+skip`

State your hypothesis: based on Lecture 16, predict which configurations will train successfully and what the gradient-norm-vs.-layer plots will look like for each.

Record and plot:

- Training loss vs. epoch (all three on the same axes).
- Per-layer gradient Frobenius norm $\|\nabla_{W^s} f\|_F$ at epoch 1 and epoch 10 (all three on the same axes, two plots).
- Final test accuracy for each configuration.

In 3–4 sentences, explain whether the skip connections rescue the sigmoid network's gradient flow. Compare the gradient-norm profile of configuration C to the prediction from Lecture 16's additive path-sum argument. Is there still attenuation, or is the profile approximately flat?

(d) **[8 pts] Batch normalization.**

Goal: test whether batch normalization stabilizes gradient flow in a deep sigmoid network, and compare its effect to that of residual connections.

Lecture 16 presented batch normalization as a mechanism that dynamically re-centers and re-scales pre-activations, keeping them in the well-behaved region of σ and preventing the per-layer multiplier m_s from drifting during training.

(i) **Implementation.** If you have not already done so, implement the `use_batchnorm=True` mode in your `DeepMLP` class. When enabled, insert `nn.BatchNorm1d(width)` before each activation function (i.e., the computation at each hidden layer becomes Linear $\to$ BN $\to \sigma$). Remember that BN absorbs the bias, so you may set `bias=False` in the corresponding `nn.Linear` layers when BN is active.

(ii) **Experiment.** Train two copies of the 40-layer **sigmoid** network with He initialization, SGD with $\eta = 0.1$, 10 epochs, batch size 128:

Config	Batch Norm	Label in plots
D	No	`sigmoid`
E	Yes	`sigmoid+BN`

Note: Configuration D is the same as configuration B from part (c) (40-layer sigmoid, He init, no skip connections, no batch normalization). You do not need to retrain it; reuse the results.

Record and plot:

- Training loss vs. epoch (both on the same axes; optionally include configuration C from part (c) for a three-way comparison).
- Per-layer gradient Frobenius norm at epoch 1 and epoch 10.
- Final test accuracy.

(iii) **Discussion.** In 3–5 sentences, address:

- Does BN rescue the sigmoid network from vanishing gradients?
- How does BN's effect compare to that of residual connections (configuration C)? Are their gradient-norm profiles similar or qualitatively different?
- Lecture 16 noted that BN's stabilization of m_s is an empirical observation (marked with $\odot$ in the summary table), not a proven guarantee. Do your results support or complicate this claim?

Graduate Extension (CSCI 6960 Only): Implicit Bias under PSD Constraints [15 pts]

This problem studies implicit bias in a simplified linear setting.

The implicit bias phenomenon you observed in problem 1(c)—where different random seeds converge to solutions with similar test accuracy—has deep theoretical roots. In overparameterized models many parameter choices produce exactly the same predictor. Gradient-based optimization nevertheless tends to select specific solutions. This phenomenon is known as *implicit bias*.

In this problem you will see that the choice of initialization, the structure of the optimization landscape, *and* the constraint set together determine which interpolating solution gradient descent converges to.

Let $\mathcal{A} : \mathbb{S}^n_+ \to \mathbb{R}^N$ be a linear map restricted to the cone of $n \times n$ positive semidefinite matrices $\mathbb{S}^n_+ = \{W \in \mathbb{R}^{n\times n} : W \succeq 0\}$ (with the Frobenius inner product on the domain), and consider the least-squares objective

$$L(W) = \frac{1}{2}\|\mathcal{A}(W) - y\|_2^2, \qquad W \in \mathbb{S}^n_+.$$

Assume the constrained interpolation problem $\mathcal{A}(W) = y,\ W \succeq 0$ has at least one solution.

Concrete example. If $\mathcal{A}(W) = \Phi\,\mathrm{vec}(W)$ for some measurement matrix $\Phi \in \mathbb{R}^{N\times n^2}$, then $\mathcal{A}$ is a linear map from matrices to vectors. Since $\mathcal{A}$ is defined on $\mathbb{S}^n$ (with the Frobenius inner product), its adjoint $\mathcal{A}^* : \mathbb{R}^N \to \mathbb{S}^n$ automatically maps into symmetric matrices. In general, $\mathcal{A}^*(\lambda)$ need not be PSD.

We study projected gradient descent on L initialized at $W_0 = 0$ and show it converges to the minimum Frobenius norm interpolating PSD solution. Denote the nonzero singular values of $\mathcal{A}$ (as a linear map from $\mathbb{S}^n_+$ with the Frobenius norm) by $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0$.

(a) **Existence and uniqueness.** Let $\mathcal{S} = \{W \in \mathbb{S}^n_+ : \mathcal{A}(W) = y\}$. Show that $\mathcal{S}$ is a nonempty closed convex set. Then argue that the minimum Frobenius norm solution $W^* = \arg\min_{W\in\mathcal{S}} \|W\|_F$ exists and is unique. (Identify properties of $\|\cdot\|_F$ and of $\mathcal{S}$ that together guarantee this.)

(b) **Projection onto $\mathbb{S}^n_+$.** The PSD projection $P_{\mathbb{S}^n_+}(M)$ of a symmetric matrix M is the closest PSD matrix in Frobenius norm. Show that

$$P_{\mathbb{S}^n_+}(M) = \sum_{i:\,\lambda_i>0} \lambda_i\, q_i q_i^\top,$$

where $M = \sum_i \lambda_i\, q_i q_i^\top$ is the eigendecomposition of M. That is, $P_{\mathbb{S}^n_+}$ zeroes out the negative eigenvalues and leaves the rest unchanged.

Hint: Write $\|M - W\|_F^2$ in terms of eigenvalues using the fact that the Frobenius norm is unitarily invariant, and minimize over $W \succeq 0$ coordinate-by-coordinate in the eigenbasis of M.

(c) **Optimality condition.** We seek to show convergence of projected gradient descent to $W^* = \arg\min_{W\succeq 0,\ \mathcal{A}(W)=y} \frac{1}{2}\|W\|_F^2$. Apply the first-order optimality condition for minimization over a closed convex set to obtain the optimality condition for W^*, then argue that this forces $W^* \in \mathrm{range}(\mathcal{A}^*)$, i.e., $W^* = \mathcal{A}^*(\lambda^*)$ for some $\lambda^* \in \mathbb{R}^N$.

(d) **Gradient of L and smoothness.** Compute $\nabla L(W)$ (as a Frobenius-inner-product gradient). Show L is σ_1^2-smooth on $\mathbb{S}^n_+$:

$$\|\nabla L(W_1) - \nabla L(W_2)\|_F \leq \sigma_1^2\, \|W_1 - W_2\|_F.$$

(e) **Projected gradient descent and convergence.** With step size $\eta = 1/\sigma_1^2$, PGD performs

$$W_{t+1} = P_{\mathbb{S}_+^n}\big(W_t - \eta\, \nabla L(W_t)\big) = P_{\mathbb{S}_+^n}\big(W_t - \eta\, \mathcal{A}^*(\mathcal{A}(W_t) - y)\big).$$

Using the nonexpansiveness of $P_{\mathbb{S}_+^n}$ (i.e., $\|P_{\mathbb{S}_+^n}(A) - P_{\mathbb{S}_+^n}(B)\|_F \le \|A - B\|_F$ for all symmetric A, B) and the fact that $W^* \succeq 0$ so $P_{\mathbb{S}_+^n}(W^*) = W^*$, show that

$$\|W_{t+1} - W^*\|_F \;\le\; \|(I - \eta\, \mathcal{A}^*\mathcal{A})(W_t - W^*)\|_F.$$

Then conclude that

$$\|W_t - W^*\|_F \;\le\; \left(1 - \frac{\sigma_r^2}{\sigma_1^2}\right)^t \|W^*\|_F.$$

How many iterations suffice to reduce the error to $\varepsilon\|W^*\|_F$? Express your answer in terms of the condition number $\kappa = \sigma_1^2/\sigma_r^2$ and ε.

Note: The contraction above shows W_t converges to W^*, but why is the limit the *minimum Frobenius norm* interpolant? Observe that the uniqueness from part (a) and the range characterization from part (c) together identify exactly one candidate, and the contraction from $W_0 = 0$ converges to it.