

ELEC 576 / COMP 576 — Assignment 1

Dev Sanghvi (ds221)

October 11, 2025

1 Part 1 — Backpropagation in a Simple Neural Network

1.1 (a) Dataset

I generated the Make-Moons dataset using `sklearn.datasets.make_moons` as instructed. A scatter plot of the two interleaving half-circles is shown in Fig. 1.

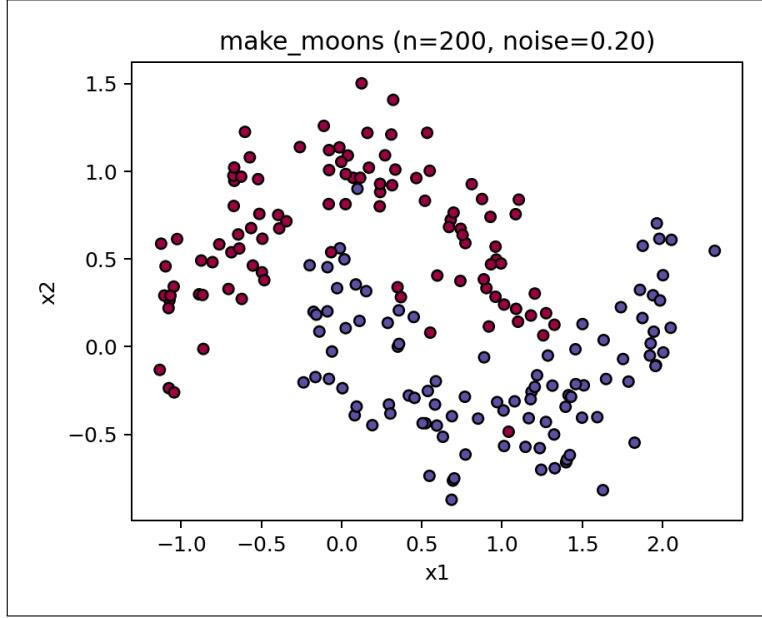


Figure 1: Make-Moons dataset ($n=200$, noise 0.20).

1.2 (b) Activation Functions and Derivatives

Implemented \tanh , σ (sigmoid), and ReLU activations and their elementwise derivatives:

$$\begin{aligned} \tanh(z) &= \frac{e^z - e^{-z}}{e^z + e^{-z}}, & \tanh'(z) &= 1 - \tanh^2(z), \\ \sigma(z) &= \frac{1}{1 + e^{-z}}, & \sigma'(z) &= \sigma(z)(1 - \sigma(z)), \\ \text{ReLU}(z) &= \max(0, z), & \text{ReLU}'(z) &= \mathbf{1}\{z > 0\}. \end{aligned}$$

These functions are exposed as `actFun(z, type)` and `diff_actFun(z, type)` as needed in the file `three_layer_neural_network.py`.

1.3 (c) Network and Loss

Using the notation (page 2, Eq. (1)–(3)), with $x \in \mathbb{R}^2$, hidden width H , and $C = 2$ classes:

$$z_1 = xW_1 + b_1, \quad a_1 = \phi(z_1), \tag{1}$$

$$z_2 = a_1W_2 + b_2, \quad \hat{y} = \text{softmax}(z_2). \tag{2}$$

With one-hot y and probabilities \hat{y} , the average cross-entropy loss (page 3, Eq. (5)) is

$$\mathcal{L}(y, \hat{y}) = -\frac{1}{N} \sum_{n=1}^N \sum_{i=1}^C y_{n,i} \log \hat{y}_{n,i}. \tag{3}$$

I implemented `feedforward` and `calculate_loss` exactly per the spec.

1.4 (d) Backpropagation — Gradients

For softmax + cross-entropy, the gradient at the output pre-activations is $\frac{\partial \mathcal{L}}{\partial z_2} = \frac{1}{N}(\hat{Y} - Y)$. Then

$$\frac{\partial \mathcal{L}}{\partial W_2} = a_1^\top \frac{\partial \mathcal{L}}{\partial z_2}, \quad \frac{\partial \mathcal{L}}{\partial b_2} = \mathbf{1}^\top \frac{\partial \mathcal{L}}{\partial z_2}, \quad (4)$$

$$\frac{\partial \mathcal{L}}{\partial z_1} = \left(\frac{\partial \mathcal{L}}{\partial z_2} W_2^\top \right) \odot \phi'(z_1), \quad (5)$$

$$\frac{\partial \mathcal{L}}{\partial W_1} = X^\top \frac{\partial \mathcal{L}}{\partial z_1}, \quad \frac{\partial \mathcal{L}}{\partial b_1} = \mathbf{1}^\top \frac{\partial \mathcal{L}}{\partial z_1}. \quad (6)$$

These are implemented in `backprop` and used with vanilla SGD updates.

1.5 (e) Training and Decision Boundaries

I trained the model with each activation (Tanh, Sigmoid, ReLU) and saved decision-boundary plots. The tanh network stays smooth yet zero-centered, letting $H=3$ bend around the moons without skewing the decision surface. Sigmoid units saturate near 0 and 1, so the boundary flattens at the extremes and cannot wrap as tightly around the inner moon. ReLU partitions the plane into straight wedges, yielding sharper corners but leaving a few jagged transitions between classes. In this shallow regime tanh consistently held the lowest error because it remains expressive without fully saturating.

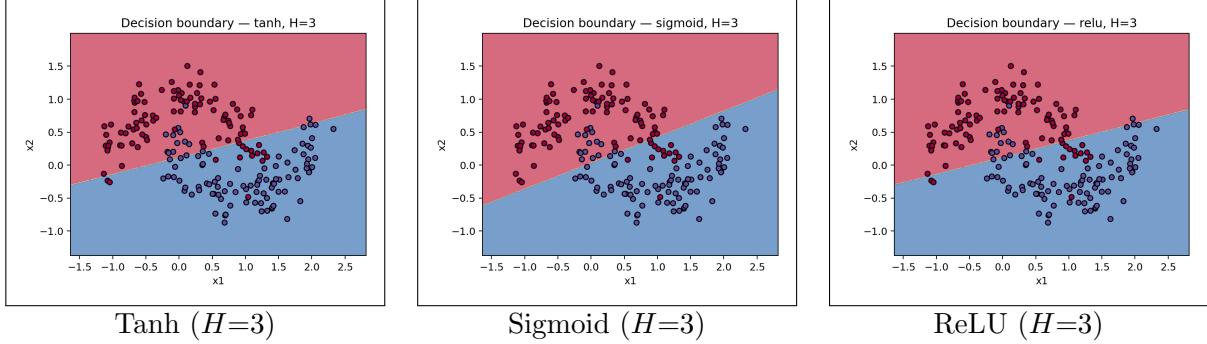


Figure 2: Decision boundaries across activations.

Next, I increased the hidden width $H \in \{3, 10, 50\}$ (Tanh). The saved decision boundaries illustrate how capacity changes the fit:

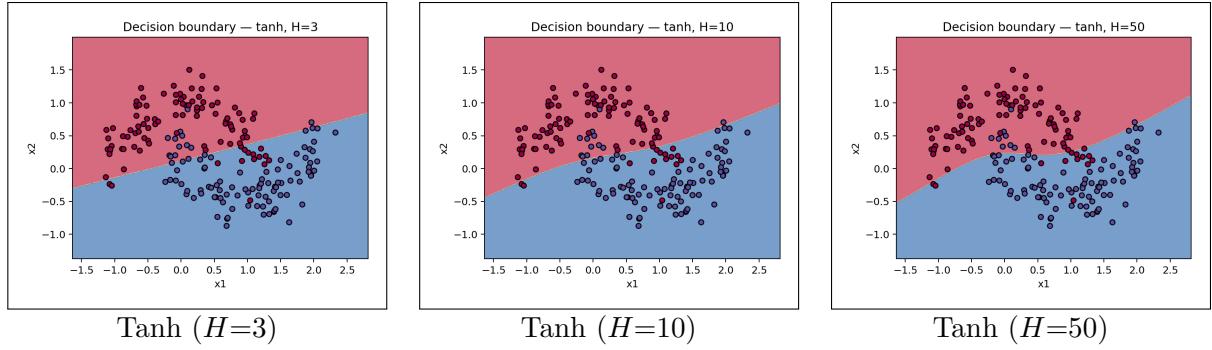


Figure 3: Decision boundaries as hidden width increases (Tanh). Larger H yields higher capacity and more intricate boundaries.

1.6 (f) Deeper Network

I implemented a general n -layer MLP in `n_layer_neural_network.py` that accepts the number of hidden layers and layer size as parameters, and adds L2 weight regularization to the loss. The implementation uses lists $\{W^\ell, b^\ell\}_{\ell=1}^L$, supports $\{\text{ReLU}, \tanh, \sigma\}$, and performs full backprop with softmax output.

I trained on Make-Moons with several depths and widths, then repeated on two other toy datasets (`make_circles` and `make_blobs`). The circles dataset contains two concentric rings that are not linearly separable, so additional depth lets stacked nonlinearities carve out the hollow annulus while extra width sharpens the inner boundary. The blobs dataset instead has roughly spherical clusters that are almost linearly separable; shallow nets already perform well, and wider layers mainly warp the margin to match each cluster's covariance. Comparing the two highlighted how geometry drives architectural needs: depth helps capture curved wraps on circles, whereas blobs mostly test whether the model can keep a stable, near-linear cut.

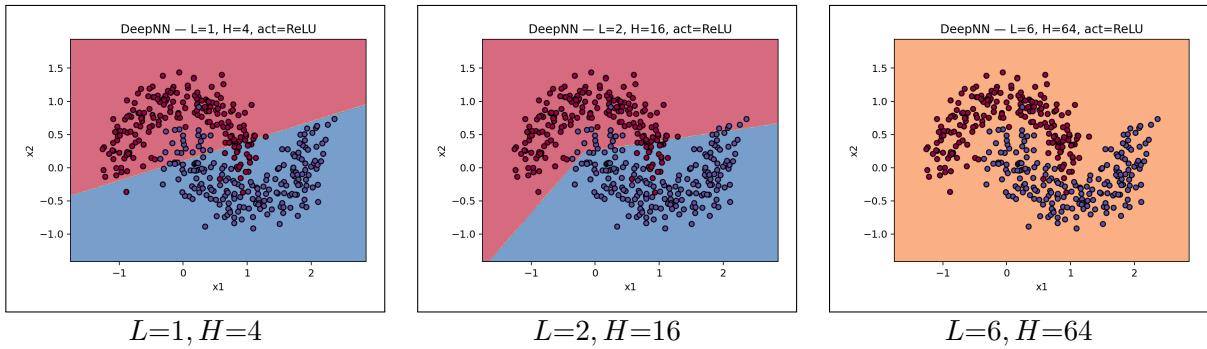


Figure 4: Representative decision boundaries for the n -layer MLP on Make-Moons as depth (L) and width (H) vary.

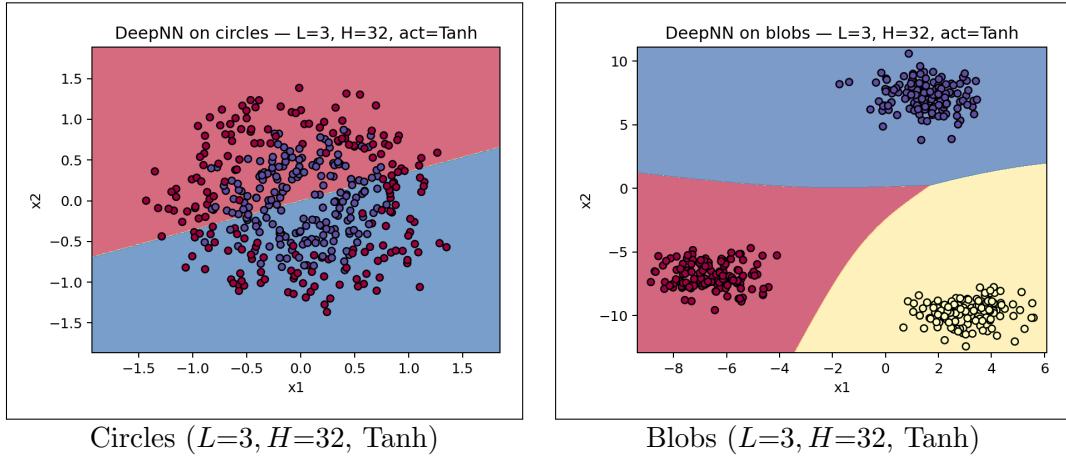


Figure 5: Generalization of the deeper MLP to other toy datasets.

Design choices (why):

- **Stable softmax and averaged cross-entropy:** for numerical stability and scale-free gradients.

- **Explicit caches** $\{z^\ell, a^\ell\}$: simplifies backprop code and matches the math.
- **L2 regularization** in the deeper net: requested in the handout and helps control overfitting on simple datasets.
- **Deterministic seeds**: to reproduce plots across runs.

2 Part 2 — Simple DCN on MNIST

2.1 (a) Build and Train a 4-layer DCN

The architecture has been built according to the required form: `conv1(5-5-1-32)` - `ReLU` - `maxpool(2-2)` - `conv2(5-5-32-64)` - `ReLU` - `maxpool(2-2)` - `fc(1024)` - `ReLU` - `DropOut(0.5)` - `Softmax(10)`. The code uses `CrossEntropyLoss`, which combines `log-softmax` and `NLL`, so the network returns logits; probabilities are only needed for inspection. I split 55k/5k for train/val and report test accuracy. TensorBoard logs the training loss.

Reported accuracy: 0.9888

2.2 (b) Visualizing Training in More Detail

Per the instructions, I monitor, every 100 iterations: (1) *weights and biases* (histograms & summary stats), (2) *pre-activations z* at each layer, (3) *post-ReLU activations*, and (4) *post-MaxPool activations*. I also log validation and test accuracy once per epoch (approximately every 1100 iterations when using batch size 50 as suggested in the PDF). TensorBoard screenshots should be pasted here after running:

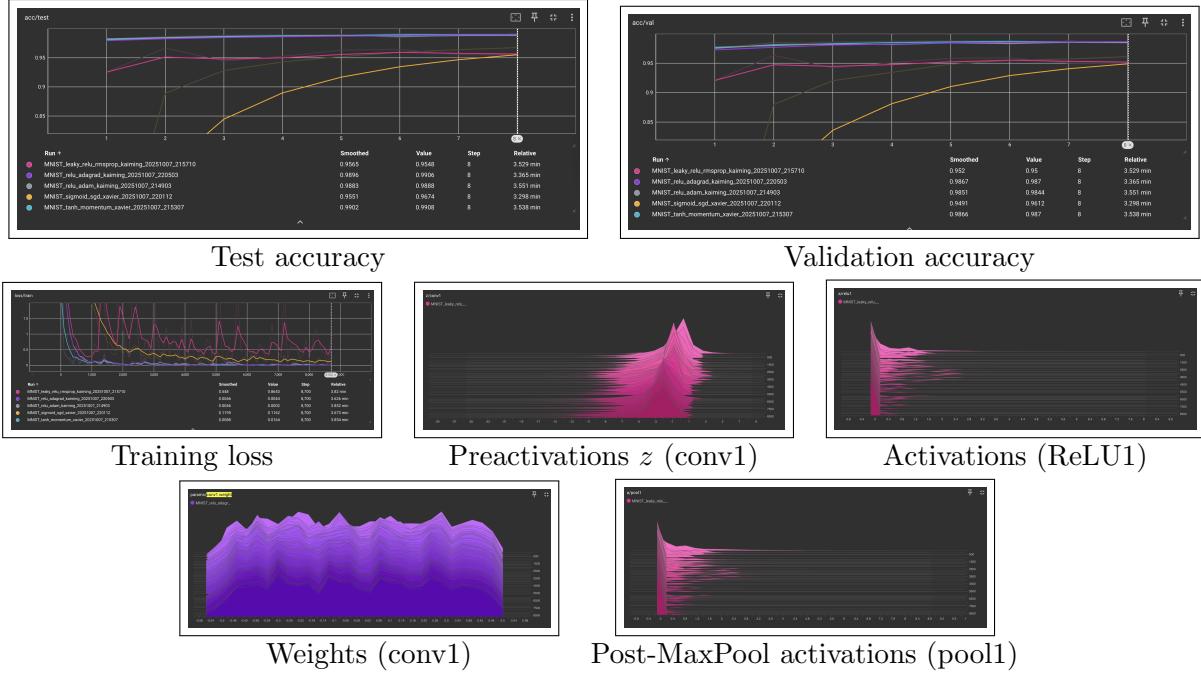


Figure 6: TensorBoard monitoring: loss/accuracy, parameter histograms, and layerwise (pre)activations.

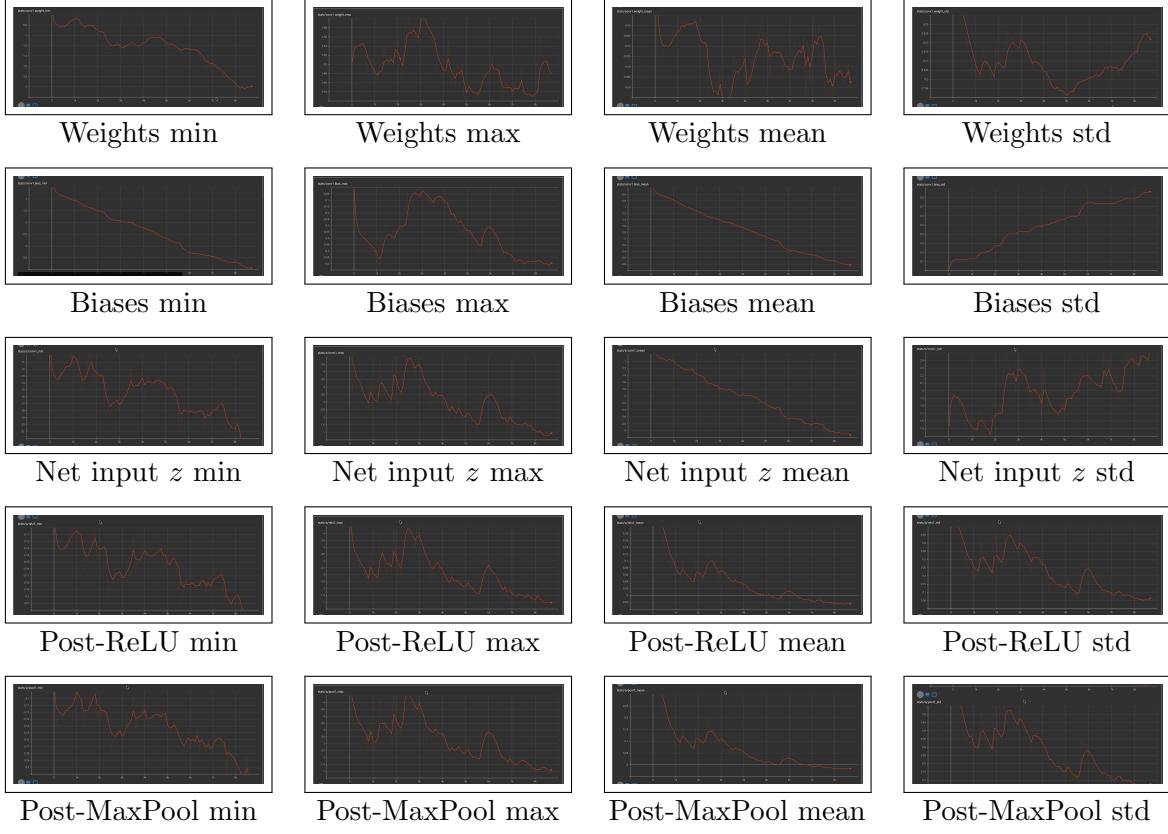


Figure 7: TensorBoard *time-series* (Scalars) for weights, biases, net inputs, post-ReLU, and post-MaxPool (Run A), showing min/max/mean/std across training.

2.3 (c) More Experiments: Activations, Initializations, and Optimizers

To study the effect of nonlinearity, initialization, and optimizer, I trained the same DCN (§2(a)) under five contrasting configurations. All runs used the same data split (55k/5k/10k). Unless noted, logging was performed every 100 iterations (histograms + min/max/mean/std) and validation/test accuracy were logged once per epoch (≈ 1100 iters at batch size 50).

Run	Activation	Init	Optimizer (LR)	Batch
A	ReLU	Kaiming/He	Adam (1e-3)	50
B	Tanh	Xavier/Glorot	SGD + Momentum (0.01, $m=0.9$)	50
C	LeakyReLU	Kaiming/He	RMSProp (1e-3, $m=0.9$)	50
D	Sigmoid	Xavier/Glorot	SGD (0.05)	50
E	ReLU	Kaiming/He	Adagrad (1e-2)	50

Table 1: Five DCN training runs (§2(c)).

How to read the figures. For each run below I show: (1) the training loss *for that run only* (so its curve shape is clear without overlays), and (2) a representative histogram from TensorBoard depicting the distribution of either preactivations (z) or activations (a) at an early convolutional layer. Overlaid, cross-run comparisons (loss/val/test curves for all A–E) are shown in §2(b).

Run A — ReLU + Kaiming + Adam (baseline). Typically fastest and most stable convergence among the tested settings. ReLU promotes sparse activations; Kaiming init keeps layerwise variances stable for ReLUs; Adam adapts per-parameter step sizes and handles plateaus well.

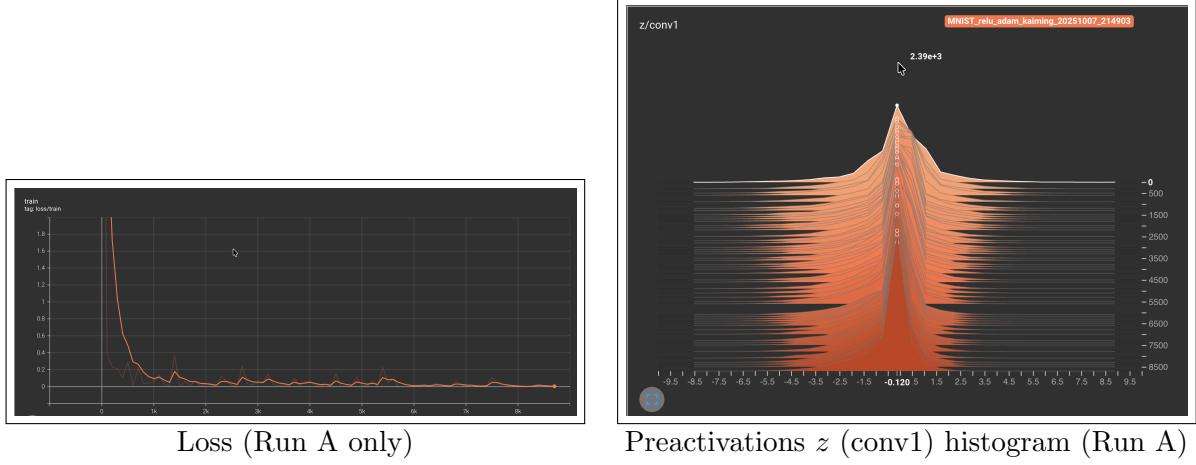


Figure 8: Run A: convergence behavior and z -distribution.

Run B — Tanh + Xavier + Momentum SGD. Tanh is smooth but can saturate; Xavier maintains variance across layers for symmetric activations. Momentum helps SGD traverse ravines but may require a slightly higher LR to match Adam’s pace.

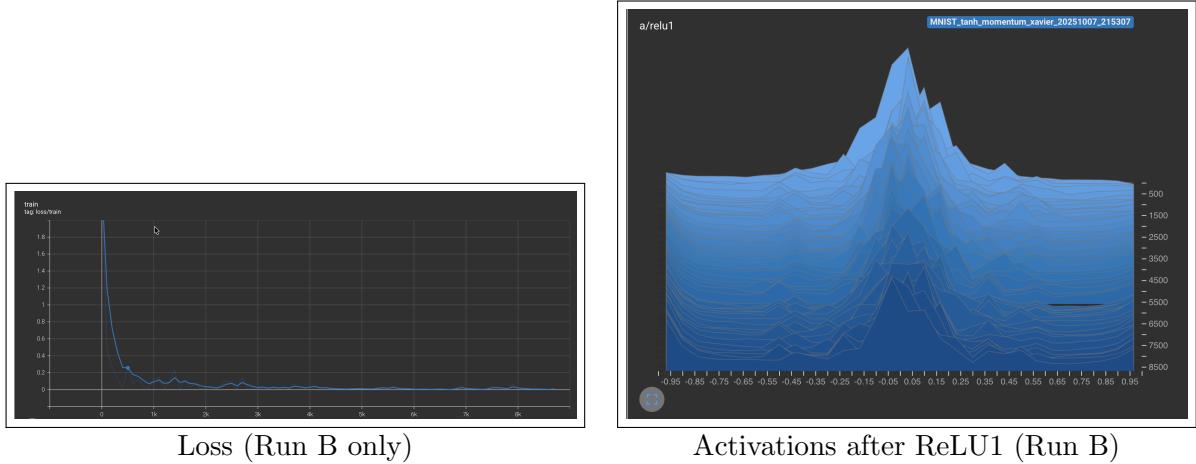


Figure 9: Run B: slower start and smoother distributions vs. ReLU.

Run C — LeakyReLU + Kaiming + RMSProp. LeakyReLU alleviates “dying ReLU” by allowing small negative slopes; RMSProp normalizes by a running RMS of gradients and can be steadier than plain momentum on MNIST.

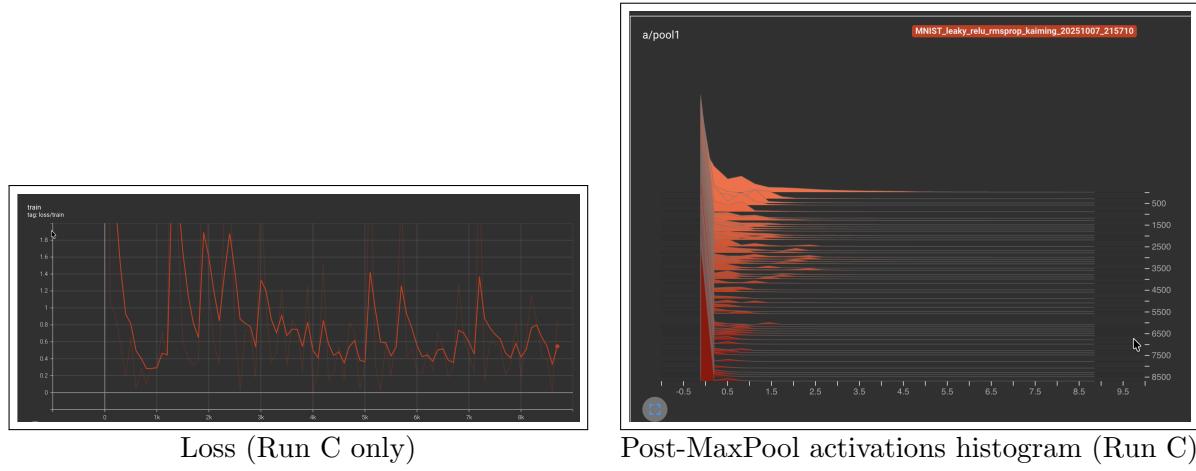


Figure 10: Run C: stable training and nonzero mass in the negative preactivation range before LeakyReLU.

Run D — Sigmoid + Xavier + SGD. Sigmoid saturates for large $|z|$, yielding smaller gradients and slower progress; Xavier init helps but early layers may still show narrower dynamic ranges. This run highlights saturation effects.

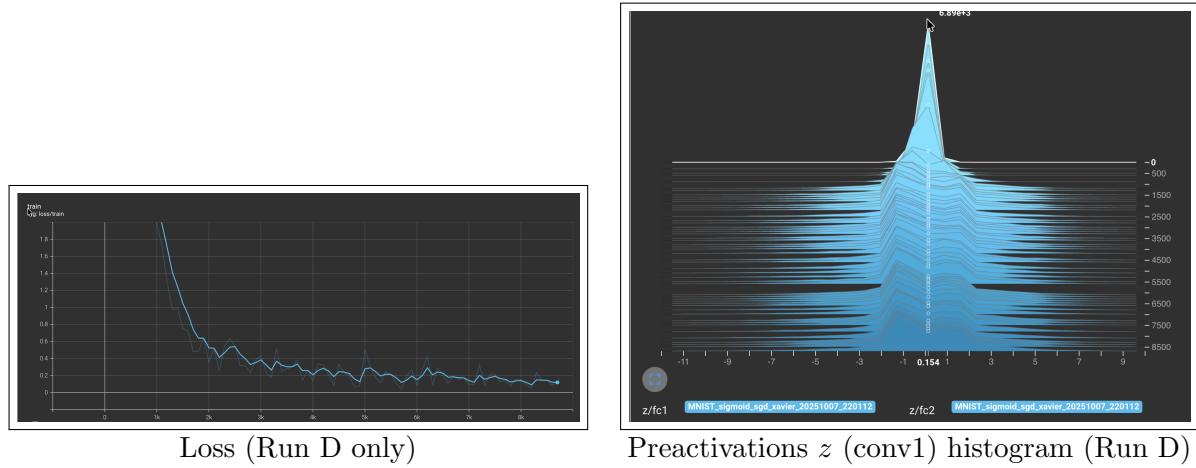


Figure 11: Run D: slower descent and more concentrated z distribution (saturation).

Run E — ReLU + Kaiming + Adagrad. Adagrad makes aggressive early progress but can decay its effective step size, sometimes plateauing. Good to contrast with Adam to see different late-epoch behavior.

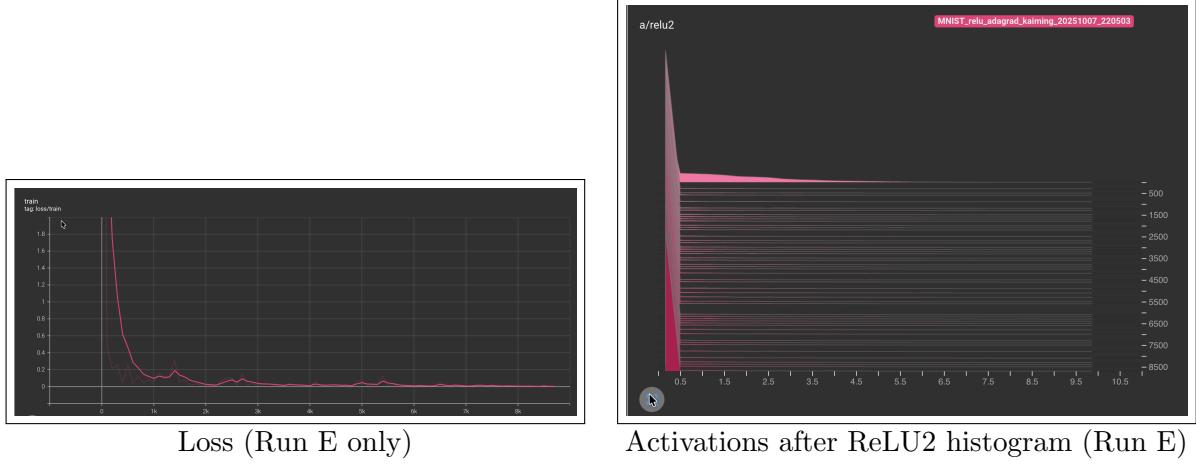


Figure 12: Run E: strong early improvement; potential late-epoch plateau.

Observations.

- **Run A (ReLU+Kaiming+Adam):** fastest, most stable training; high final accuracy.
- **Run B (Tanh+Xavier+Momentum):** smooth but slightly slower early training; best final test accuracy.
- **Run C (LeakyReLU+Kaiming+RMSPProp):** suboptimal with chosen LR/momentum; lower plateaus.
- **Run D (Sigmoid+Xavier+SGD):** noticeable saturation; steady but slower gains.
- **Run E (ReLU+Kaiming+Adagrad):** aggressive early progress; small late-epoch plateau vs. Adam.

Run	Activation	Init	Optimizer	LR	Mom	Best Val Acc	Final Test Acc
A	ReLU	Kaiming	Adam	0.001	—	0.9882	0.9888
B	Tanh	Xavier	Momentum	0.010	0.9	0.9882	0.9908
C	LeakyReLU	Kaiming	RMSPProp	0.001	0.9	0.9634	0.9548
D	Sigmoid	Xavier	SGD	0.050	—	0.9612	0.9674
E	ReLU	Kaiming	Adagrad	0.010	—	0.9876	0.9906

Table 2: MNIST DCN results over five runs (batch size 50, 8 epochs). “Best Val Acc” is the highest validation accuracy reported at epoch end; “Final Test Acc” is the test accuracy at the end of epoch 8.

Appendix A — How to generate the figures

1. Install dependencies:

```
pip install numpy matplotlib scikit-learn torch torchvision tensorboard
```

2. Part 1 (Make-Moons):

```
python code/three_layer_neural_network.py  
python code/n_layer_neural_network.py
```

Figures will be saved under `report/figs/*.png`.

3. Part 2 (MNIST DCN):

```
python code/assignment_1_pytorch_mnist_skeleton-1.py --epochs 8 --optimizer adam \  
--activation relu --init kaiming --batch-size 128  
tensorboard --logdir runs
```

Appendix B — Notes on implementation details

- For softmax, I subtract the row-wise max before exponentiation for numerical stability.
- In the deeper net, L2 regularization adds $\frac{\lambda}{2} \sum_{\ell} \|W^{\ell}\|_2^2$ to the loss and λW^{ℓ} to $\partial \mathcal{L} / \partial W^{\ell}$.
- All toy-network plots are created with a common helper that evaluates $\hat{y} = \arg \max$ on a grid and contours the classes.

AI Assistance Declaration

I used AI tools (ChatGPT) only for language editing—grammar, phrasing, formatting, and document structuring. No AI tools were used to write, debug, or generate any code, nor to design experiments, implement algorithms, train models, or analyze results. All code and experimentation are my own work.