CSCE584 HW 1

Codes are available in Github. If links below are not opening, please use this link -

https://github.com/Ritvik-G/USC_CSCE584/tree/main/HW1/codes

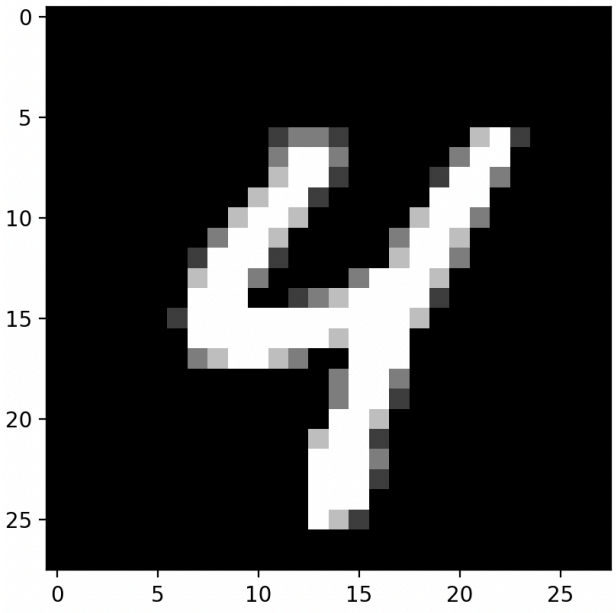
Question - 1

Read the article in the link below (a) and/or a paper from the reference (b). Use the codes in the article and re-run the code to generate the results. Submit the following (c-d).
(a) Article: Classification of handwritten digits
(b) Generate any figure that you think would convey if your neural network model is trained properly (submit captioned image with good resolution).
(c) Copy of the code with your understanding of how the code works (a pseudo-code with comments would do).
(d) With the same neural network architecture with the same number of hidden layers, if you increase the number of neurons in the hidden layers, does the network performance improve, stay the same, or degrade? Generate a figure (or figures) that conveys the reason for your conclusion.

Code Link -

(a-c) - Link

(D) - LINK



Original Label : 4

Predicted value: 4

Pseudocode of how the algo works:

1. Load and prep data
    a. Read mnist_train.csv
    b. x_train = pixels (columns 1..784) / 255 to get values in [0,1]

    c. One-hot encode labels -> size 10 vectors

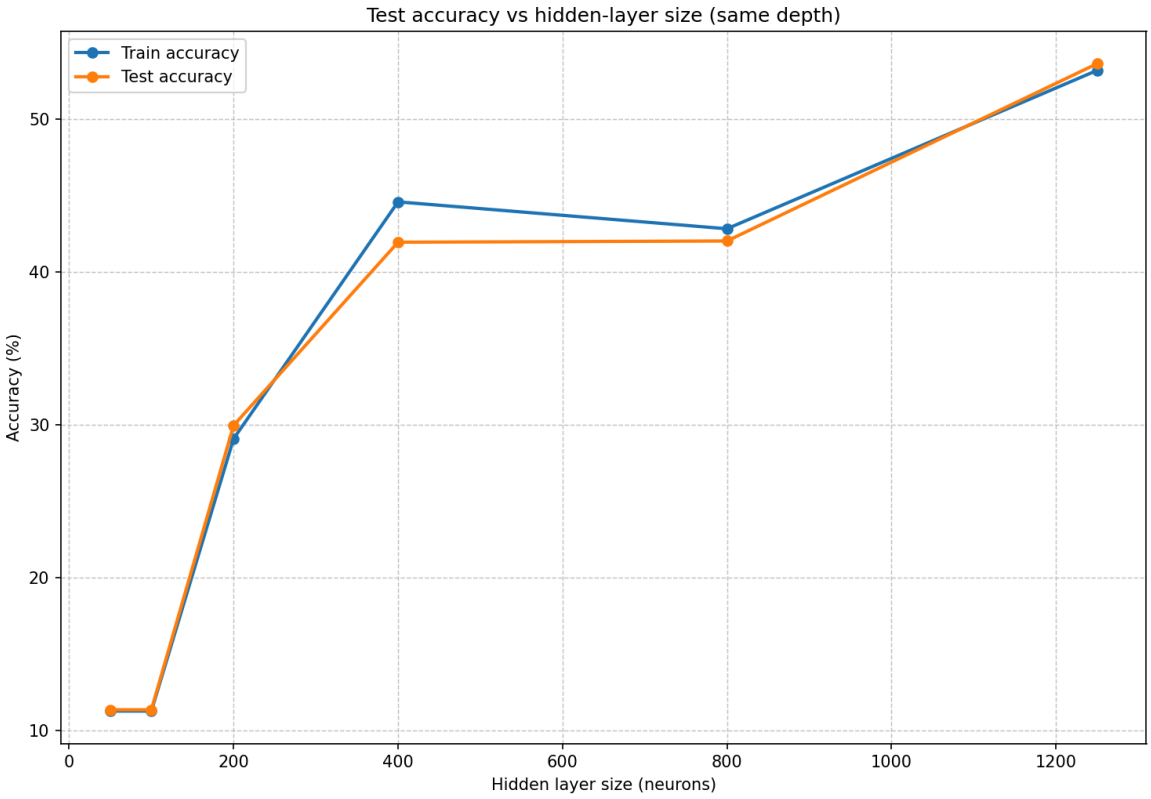
2. Define a simple DNN with one hidden layer
    a. DNN(layers): e.g., [784, 1250, 10]
    b. For each layer pair, make a weight matrix of shape (next, current + 1) (+1 is for the bias we'll append)
    c. sigmoid(x): 1 / (1 + exp(-0.01 * x)) # small scale to avoid saturation
    d. predict(x):
        i. For each layer: append a 1 to x for bias
        ii. z = W @ x
        iii. x = sigmoid(z)
        iv. return final 10 numbers (one per digit)

3. Train step (backprop, one sample)
    a. Do the forward pass and save activations (with bias where needed)
    b. Compute output error using squared error + sigmoid derivative
    c. Backpropagate that error through the layers
    d. Compute gradients for each weight matrix
    e. Update weights: W = W - 0.1 * gradient

4. Training loop
    a. Repeat for many random samples (SGD)
    b. Keep a rolling average of recent errors to monitor progress

5. Testing
    a. Load mnist_test.csv and normalize
    b. For each test image: predict -> argmax
    c. Compare to true label; compute % correct

Increasing hidden layers increase the performance of the model as better learning is enabled. Below are the results with tests done for different size of inputs:



Through the image below, we can clearly see the increase in accuracy with the increase in hidden layers

```
(csce584) ritvikg@MacBookPro-17 q1 % python q1b.py

=== Training model with hidden size = 50 ===
  step 2000: avg loss over last 2000 ≈ 1.0633
  step 4000: avg loss over last 2000 ≈ 0.9008
  step 6000: avg loss over last 2000 ≈ 0.9005
  step 8000: avg loss over last 2000 ≈ 0.9007
  step 10000: avg loss over last 2000 ≈ 0.9002
  Train accuracy (@5000 samples): 11.26%
  Test  accuracy: 11.35%

=== Training model with hidden size = 100 ===
  step 2000: avg loss over last 2000 ≈ 1.0155
  step 4000: avg loss over last 2000 ≈ 0.9014
  step 6000: avg loss over last 2000 ≈ 0.9006
  step 8000: avg loss over last 2000 ≈ 0.9010
  step 10000: avg loss over last 2000 ≈ 0.9008
  Train accuracy (@5000 samples): 11.26%
  Test  accuracy: 11.35%

=== Training model with hidden size = 200 ===
  step 2000: avg loss over last 2000 ≈ 0.9919
  step 4000: avg loss over last 2000 ≈ 0.9010
  step 6000: avg loss over last 2000 ≈ 0.9013
  step 8000: avg loss over last 2000 ≈ 0.8988
  step 10000: avg loss over last 2000 ≈ 0.8928
  Train accuracy (@5000 samples): 29.08%
  Test  accuracy: 29.95%

=== Training model with hidden size = 400 ===
  step 2000: avg loss over last 2000 ≈ 0.9827
  step 4000: avg loss over last 2000 ≈ 0.8992
  step 6000: avg loss over last 2000 ≈ 0.8944
  step 8000: avg loss over last 2000 ≈ 0.8848
  step 10000: avg loss over last 2000 ≈ 0.8502
  Train accuracy (@5000 samples): 44.58%
  Test  accuracy: 41.94%

=== Training model with hidden size = 800 ===
  step 2000: avg loss over last 2000 ≈ 1.0142
  step 4000: avg loss over last 2000 ≈ 0.8974
  step 6000: avg loss over last 2000 ≈ 0.8855
  step 8000: avg loss over last 2000 ≈ 0.8514
  step 10000: avg loss over last 2000 ≈ 0.7798
  Train accuracy (@5000 samples): 42.82%
  Test  accuracy: 42.02%

=== Training model with hidden size = 1250 ===
  step 2000: avg loss over last 2000 ≈ 1.1503
  step 4000: avg loss over last 2000 ≈ 0.8959
  step 6000: avg loss over last 2000 ≈ 0.8797
  step 8000: avg loss over last 2000 ≈ 0.8269
  step 10000: avg loss over last 2000 ≈ 0.7312
  Train accuracy (@5000 samples): 53.18%
  Test  accuracy: 53.62%

Saved figure: hidden_size_vs_accuracy.png
```

Question - 2

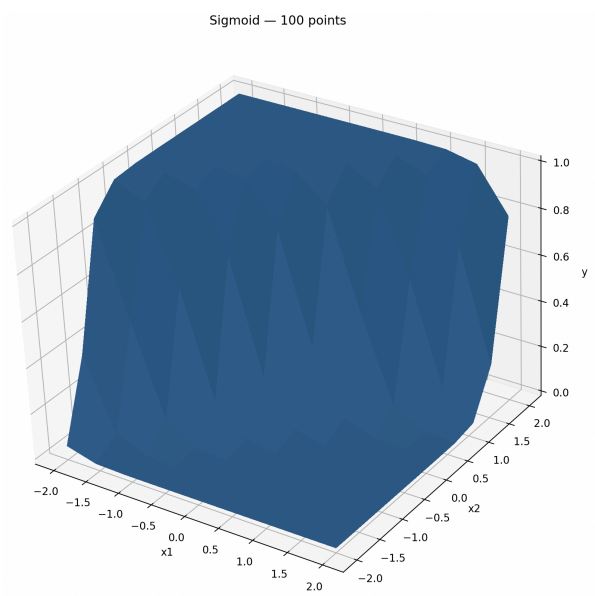
(a)

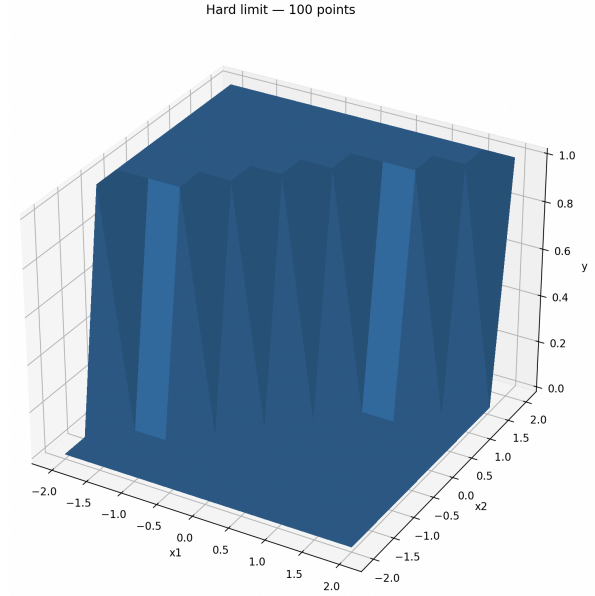
A perceptron with two inputs and one output is given by y= σ(−4.79x1 + 5.90x2− 0.93).

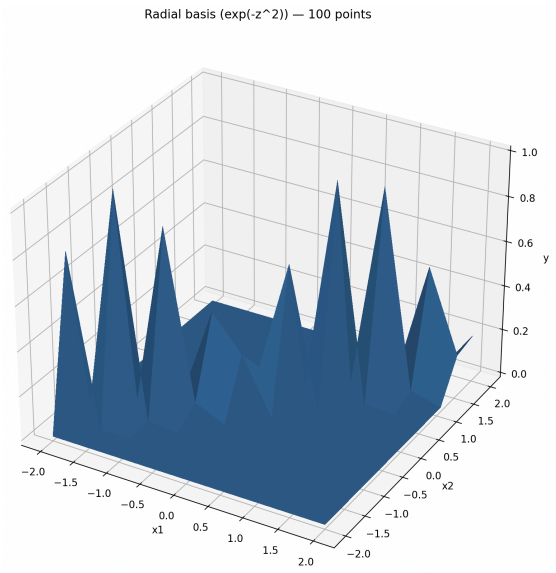
For the inputs defined in the domain [−2,2] × [−2,2], plot the output surface y as a function of x1,x2, when the activation function is (a) sigmoid (b) Hard limit and (c) Radial basis function. Plot the function with 100, 5000, 10000, sample points from the domain.

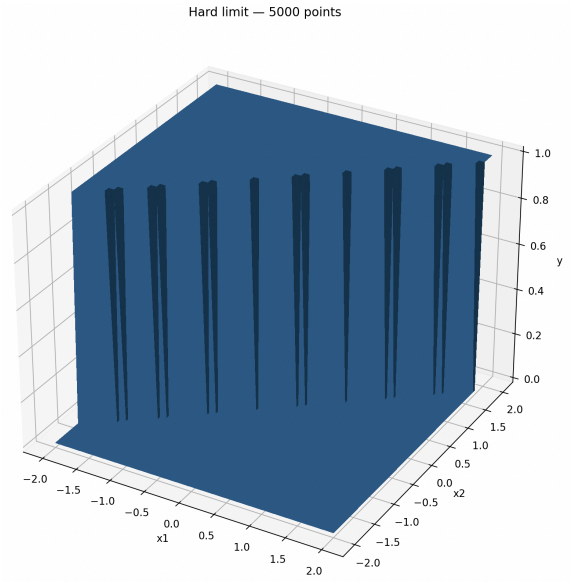
Code Link - [Link]

Results

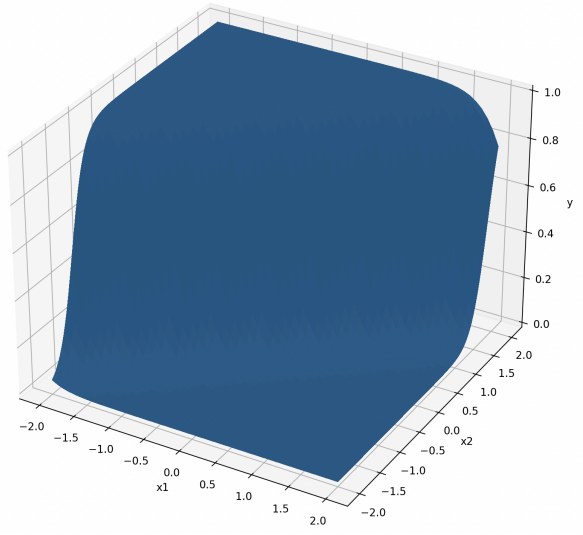




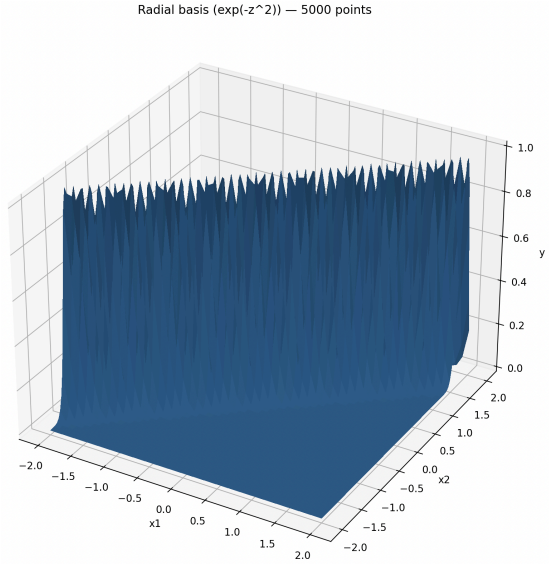




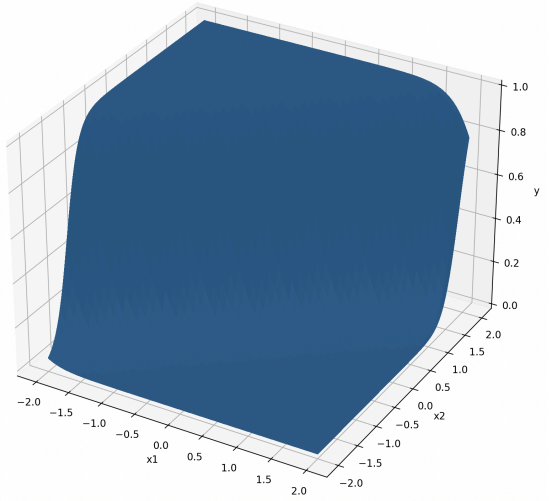
Sigmoid — 5000 points



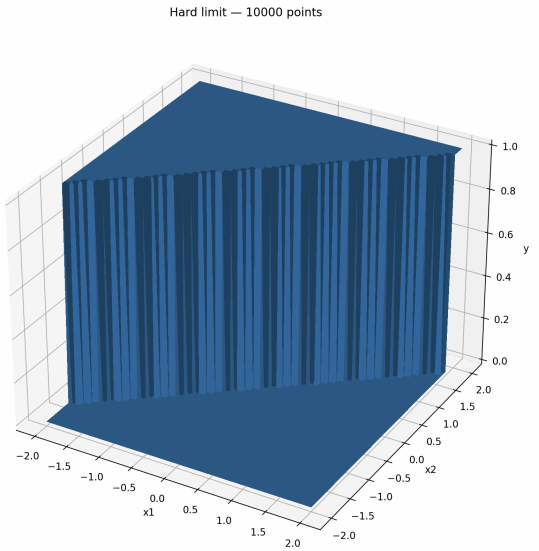
Radial basis (exp(-z^2)) — 5000 points



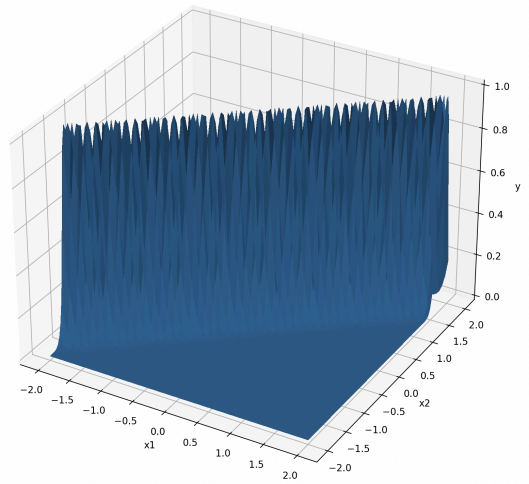
Sigmoid — 10000 points



Hard limit — 10000 points



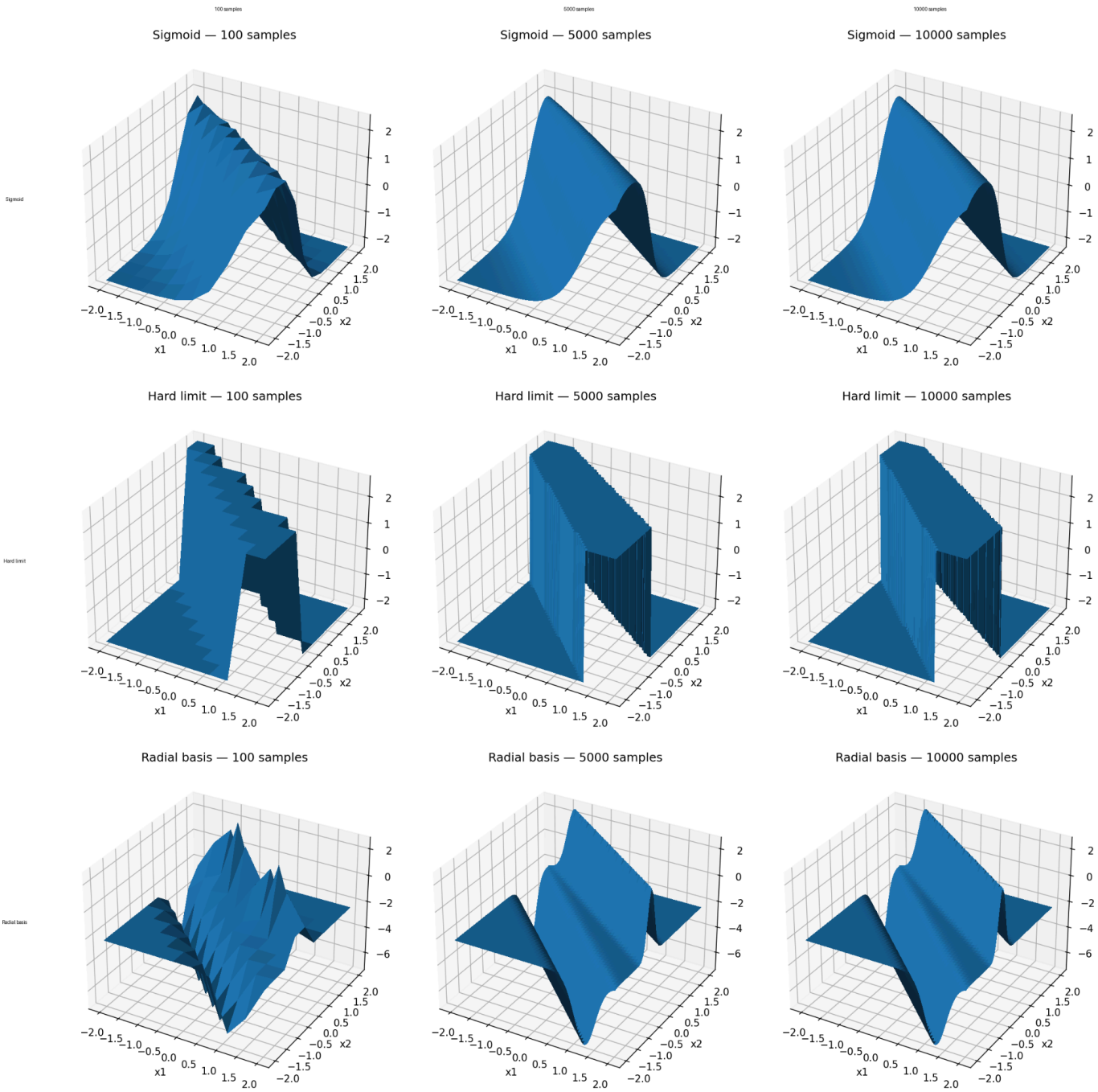
Radial basis (exp(-z^2)) — 10000 points



(b)

Code Link - [Link]

Results



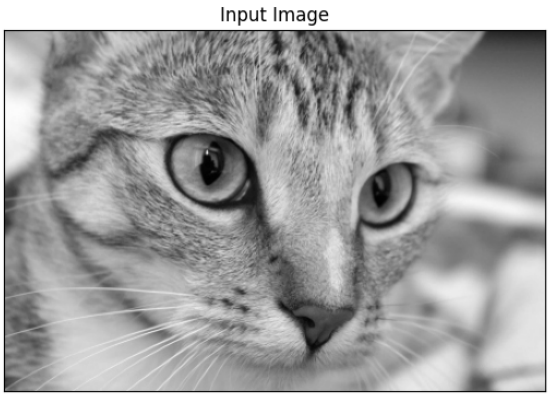
Question - 3

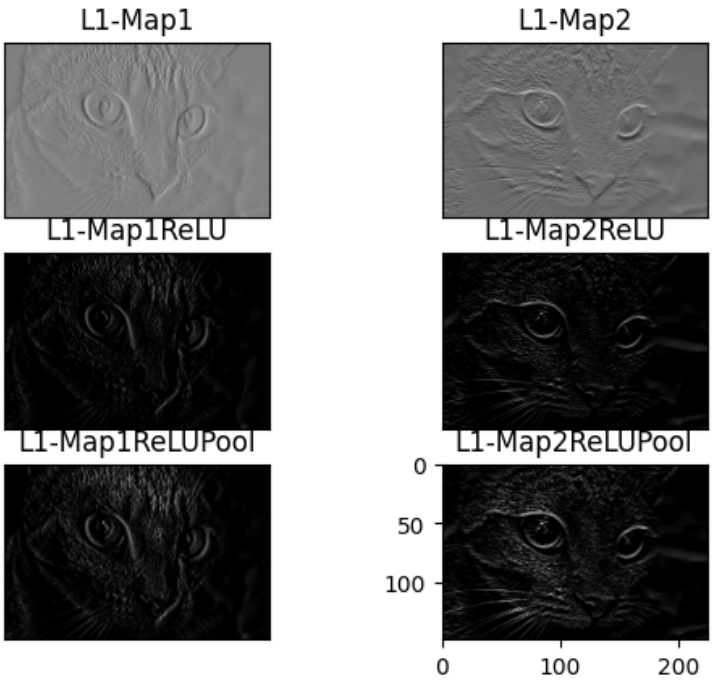
Read the article in the link given. Use the codes in the article to reproduce the results given in it. Submit the results, your code, and a detailed step-by-step description of the algorithm.

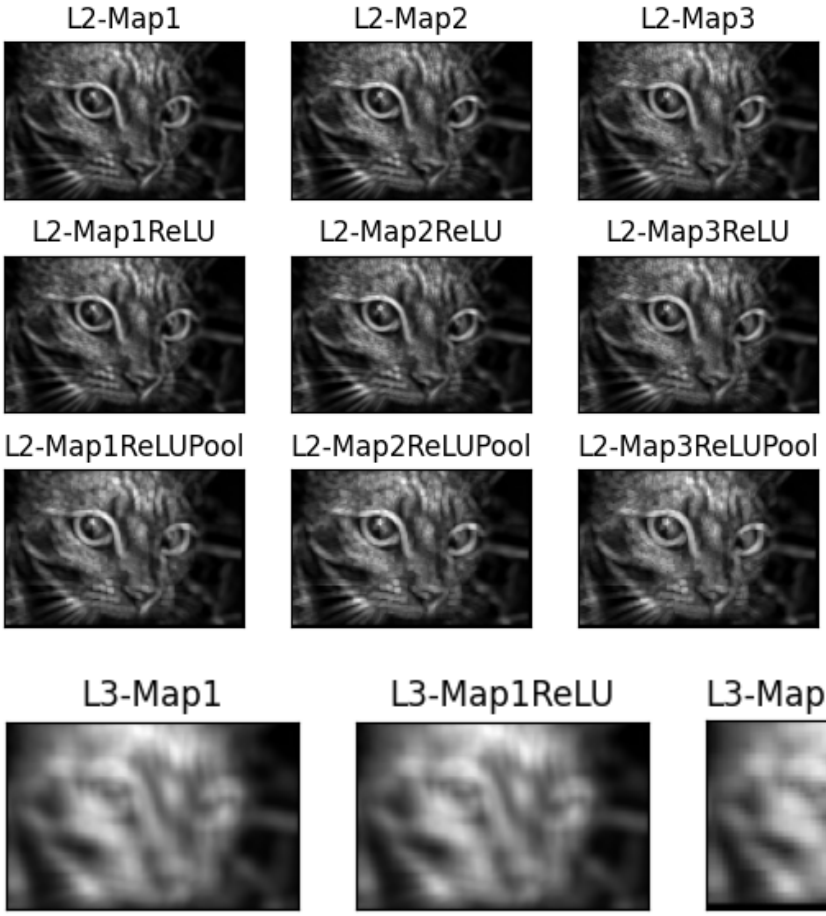
Code Link - [Link]

Results

```
(csce584) ritvikg@MacBookPro-17 HW1 % python3 q3.py

**Working with conv layer 1**
Filter  1
Filter  2

**ReLU**

**Pooling**
**End of conv layer 1**


**Working with conv layer 2**
Filter  1
Filter  2
Filter  3

**ReLU**

**Pooling**
**End of conv layer 2**


**Working with conv layer 3**
Filter  1

**ReLU**

**Pooling**
**End of conv layer 3**
```







Step By Step Description

This code runs a CNN workflow for mapping on a cat image.

1) Goal
Show how a tiny convolutional neural network works using only NumPy
(no PyTorch/TensorFlow). We run a few conv→ReLU→pool “blocks” on one image
to see how feature maps evolve. There is NO training here—just fixed filters
and forward passes.

2) Load one image
- Pull a sample image (e.g., from scikit-image) or a local file.
- Keep it as a NumPy array; if it’s color, the code may convert/select a single channel.
- This array is the network’s input.

3) Define simple filters for each layer
- For layer 1, layer 2, and layer 3, the code sets up small filter banks
  (e.g., edge detectors / sharpeners / identity-like filters).
- Each “filter bank” is just a stack of tiny matrices (kernels).

4) Convolution block 1
- Convolution: slide each kernel over the image and compute weighted sums → feature maps.
  (Implemented by `numpycnn.conv`.)

- ReLU: set negative values to 0 to keep only “activated” responses.
  (Implemented by `numpycnn.relu`.)
- Pooling: downsample with a 2×2 window to shrink width/height and keep strong signals.
  (Implemented by `numpycnn.pooling(..., 2, 2)`.)
- Print statements show progress and (often) the shapes of the feature maps.

5) Convolution block 2
- Treat the pooled maps from block 1 as the “new image.”
- Repeat: conv → ReLU → 2×2 pooling with the second filter bank.

6) Convolution block 3
- Same pattern again with the third filter bank: conv → ReLU → 2×2 pooling.
- By now, spatial size is smaller; channels may be larger (more filters).

7) Visualize results
- Plot and save the original input image.
- (Typically) also plot and/or save grids of the feature maps after each block
  so you can see edges/textures becoming more pronounced and abstract.

8) Intuition
- Convolution detects small patterns (edges/corners) by matching kernels to patches.
- ReLU keeps positive evidence and zeroes negatives (nonlinear step).
- Pooling reduces resolution, keeps the strongest responses, and gives some invariance.
- Stacking 3 blocks means later maps represent “patterns of patterns.”

How to read the code
- Core ops live in `numpycnn.conv`, `numpycnn.relu`, and `numpycnn.pooling`.
- Each block follows the exact same three steps on the current tensor.
- The notebook logs progress and writes images so you can inspect each stage.

Question - 4

Code Link - Link

Results : Train - 97.14%, Val - 80.8%, Test - 82.10%

```
(csce584) ritvikg@MacBookPro-17 HW1 % python q4.py
Device: mps
Config: TrainConfig(model='gcn', dataset='Cora', hidden=64, epochs=200, lr=0.01, weight_decay=0.0005, dropout=0.5, heads=8, out_heads=1, patience=50, runs=1, seed=42, verbose=True)
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.cora.x
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.cora.tx
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.cora.allx
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.cora.y
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.cora.ty
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.cora.ally
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.cora.graph
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.cora.test.index
Processing...
Done!
Epoch 001 | Loss 1.9459 | Train 40.71% | Val 27.40% | Test 30.10%
Epoch 010 | Loss 1.7660 | Train 96.43% | Val 72.20% | Test 74.40%
Epoch 020 | Loss 1.3878 | Train 95.71% | Val 77.00% | Test 78.40%
Epoch 030 | Loss 0.9648 | Train 97.14% | Val 79.00% | Test 80.90%
Epoch 040 | Loss 0.6701 | Train 97.14% | Val 79.60% | Test 81.50%
Epoch 050 | Loss 0.4631 | Train 97.86% | Val 80.60% | Test 82.20%
Epoch 060 | Loss 0.3996 | Train 99.29% | Val 80.20% | Test 81.20%
Epoch 070 | Loss 0.3191 | Train 100.00% | Val 79.80% | Test 81.70%
Epoch 080 | Loss 0.3140 | Train 100.00% | Val 80.20% | Test 80.90%
Early stopping at epoch 86 (no val improvement for 50 epochs).
[Run 1] Final: Train 97.14% | Val 80.80% | Test 82.10%
```

Step By Step Description

The code runs a GNN network that does the following -

1) Load and tidy the graph
We pull a small citation network (Cora, CiteSeer, or PubMed) from PyTorch Geometric. It gives us one package with node features (what each paper looks like), edges (who connects to whom), labels (the topic), and masks that say which nodes are for training, validation, and testing. We normalize the features so comparisons are fair.

2) Pick a model “style”
Choose GCN, GAT, or GraphSAGE. They all follow the same idea: each node listens to its neighbors, summarizes what it hears, and mixes that with its own info using learnable weights.

3) Do a forward pass
For each layer, every node gathers its neighbors’ features using the edge list. It combines them (GCN = weighted average, GAT = attention-weighted average, GraphSAGE = mean), then applies a linear transform, an activation (ReLU/ELU), and sometimes dropout. The final layer produces class scores for every node—higher score means “more likely this class.”

4) Compute the training signal
We only compute loss on the training nodes. Cross-entropy compares the model’s scores to the true labels there and tells us how wrong we are.

5) Learn from it
Adam updates the weights to reduce that loss. Weight decay and dropout help keep the model from overfitting. One cycle is: forward → loss → backward (gradients) → optimizer step.

6) Repeat and watch validation
We run many epochs of those updates. After each one, we check accuracy on the validation nodes. If validation stops improving for a while, we stop early and keep the best version we saw.

7) Report test results
With the best weights loaded, we run once more and measure accuracy on the test nodes—the fraction where the top-scoring class matches the true label.

8) Mental picture (one node)
“I’m a node. I look at my neighbors, average or weight their info, blend it with my own features, and decide my label. With two layers, I also indirectly hear my neighbors’ neighbors.”

Quick picks:
- GCN = fast, reliable baseline
- GAT = learns which neighbors matter most
- GraphSAGE = simple, scales well

Question - 5

Read the article in the link below. Use the codes in the article to reproduce the results given in it. Submit the results, your code, and a detailed step-by-step description of the algorithm.

Code Link - Link

Results

```
(csce584) ritvikg@MacBookPro-17 HW1 % python q5.py
562 batches of size 16
187 batches of size 16
/Users/ritvikg/Desktop/Classes/Neural Networks/HW1/csce584/lib/
  warnings.warn(
Training and validating model
---------------------------- Epoch 1 ----------------------------
Training loss: 0.7361
Validation loss: 0.4264

---------------------------- Epoch 2 ----------------------------
Training loss: 0.4229
Validation loss: 0.3726

---------------------------- Epoch 3 ----------------------------
Training loss: 0.3740
Validation loss: 0.3207

---------------------------- Epoch 4 ----------------------------
Training loss: 0.3344
Validation loss: 0.2889

---------------------------- Epoch 5 ----------------------------
Training loss: 0.3110
Validation loss: 0.2495

---------------------------- Epoch 6 ----------------------------
Training loss: 0.2898
Validation loss: 0.2241

---------------------------- Epoch 7 ----------------------------
Training loss: 0.2739
Validation loss: 0.2065

---------------------------- Epoch 8 ----------------------------
Training loss: 0.2611
Validation loss: 0.1961

---------------------------- Epoch 9 ----------------------------
Training loss: 0.2515
Validation loss: 0.1859

---------------------------- Epoch 10 ----------------------------
Training loss: 0.2412
Validation loss: 0.1692
```

Step By Step Description

This code runs neural network's transformer model.

1) Goal
We teach a tiny Transformer to copy short sequences of tokens. Tokens are 0, 1, plus special markers:
<SOS> = 2 (start), <EOS> = 3 (end).

2) Make synthetic data
We generate many sequences of length 8 wrapped with <SOS> … <EOS>.
Three patterns are used: all 1s, all 0s, or an alternating 0/1 pattern.
For each sample we set input X and target y to the same sequence (copy task), then shuffle.

3) Batch the data
We group the pairs (X, y) into mini-batches (default 16 examples per batch).
(Optional padding exists in code comments, but the run uses fixed-length sequences so no padding is applied.)

4) Positional encoding
We build the standard sine/cosine positional encoding so the model knows token order.
(It adds a “position signal” to each embedded token vector.)

5) The model
It’s an encoder-decoder Transformer with:
- one shared token embedding (vocab size 4 → model dim 8)
- 3 encoder layers, 3 decoder layers, 2 attention heads, dropout ~0.1
A small output projection turns decoder states into scores over the 4 tokens.

6) Masks (why the model can’t peek at the future)
We create a causal “target mask” (lower-triangular matrix) so at time step t the decoder only sees tokens ≤ t.
(There’s also support for padding masks if needed.)

7) Forward pass (per batch)
- Embed X (source) and y_input (target without its last token).
- Add positional encodings.
- Run the Transformer with the causal mask.
- Project to token scores (logits) for each time step.

8) Training signal
We shift the target once:
```
y_input = y[:, :-1]       # what we feed in (starts with <SOS>)
y_expected = y[:, 1:]     # what we must predict next (ends with <EOS>)
```
Loss = CrossEntropy(logits, y_expected) over all time steps.

9) Optimize
Use SGD (lr 0.01). For each batch:
forward → compute loss → backprop → optimizer.step().
We track the average loss over the epoch.

10) Validation
Same as training but without gradient updates. We report average validation loss.

11) Training loop
Train for ~10 epochs. After each epoch, print train loss and validation loss so we can see learning progress.

12) What the model learns (intuition)
Given <SOS> and the tokens so far, the decoder learns to predict the next token in the sequence, including when to finish with <EOS>. Since X == y here, it's effectively learning to copy the input stream token by token.

Question - 6

Read the article in the link below. Use the codes in the article to reproduce the results given in it for vanilla GAN and DCGAN. Submit the results, your code, and a detailed step-by-step description of the algorithm.

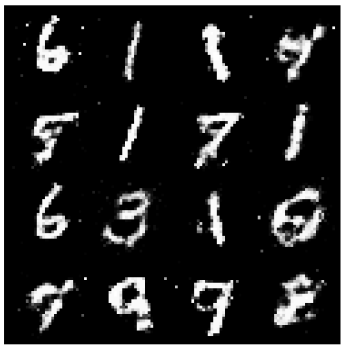
Code Link - Link

Results

GAN Outputs over iterations -

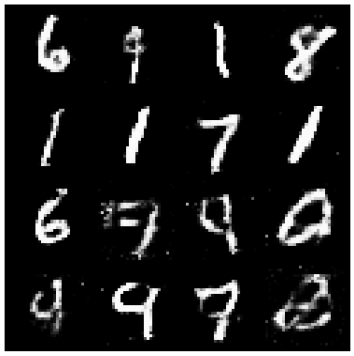
Epoch 0

Epoch 50

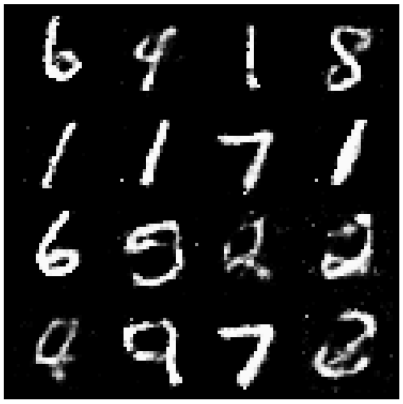


Epoch 100

Epoch 150

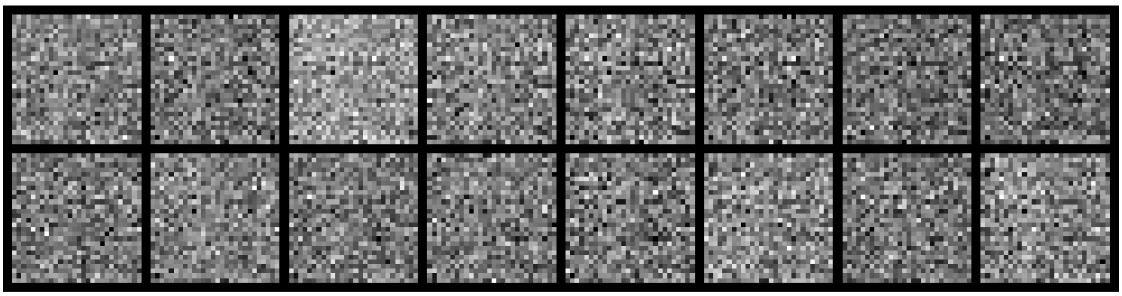


Epoch 200



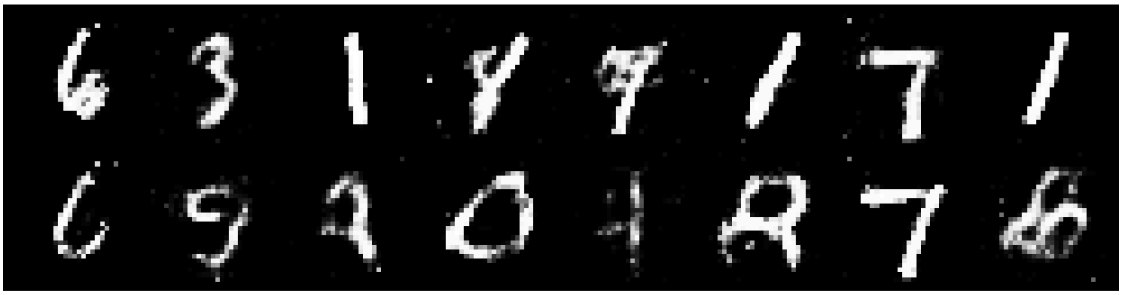
VGAN Results

Epoch 0



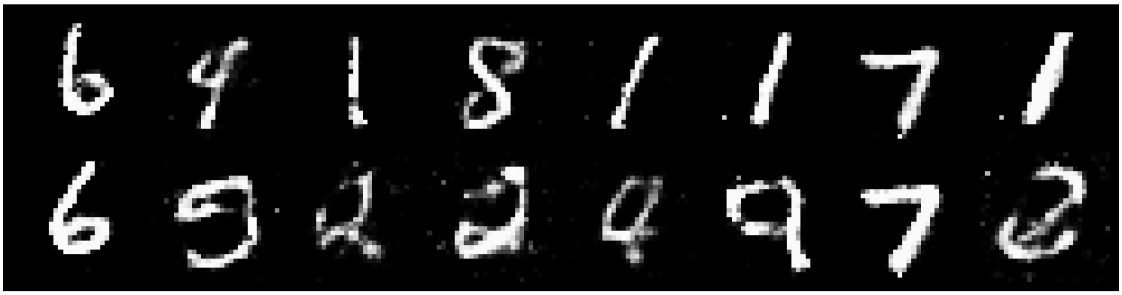
Epoch 50

Epoch 100



Epoch 150

Epoch 200



Step By Step Description

This code runs GAN and DCGAN from the article linked.

1) Load & normalize MNIST
- Pull the training set with torchvision.
- Convert images to tensors and normalize with mean=0.5, std=0.5 → pixel values in [-1, 1].
- Create a DataLoader with batch_size=100 and shuffling.

2) Shape helpers
- images_to_vectors: flatten 28×28 images → 784-D vectors.
- vectors_to_images: reshape 784-D vectors back to 1×28×28 for viewing.

3) Discriminator (D): “real or fake?”
- MLP that maps 784 → 1024 → 512 → 256 → 1.
- Layers use LeakyReLU(0.2) + Dropout(0.3).

- Final output uses Sigmoid giving a probability “this looks real”.

4) Generator (G): “make a digit from noise”
- MLP that maps 100-D Gaussian noise → 256 → 512 → 1024 → 784.
- Hidden layers use LeakyReLU(0.2).
- Final output uses Tanh so pixels land in [-1, 1] (matching the normalization).

5) Losses & optimizers
- Binary cross-entropy (BCE) everywhere.
  - ones_target(N) = label 1 for real (or “fool D” for G).
  - zeros_target(N) = label 0 for fake.
- Adam(lr=0.0002) for both D and G.

6) Train step — Discriminator (per batch)
- Flatten a batch of real images → real_data.
- Make fake_data = G(noise).detach() (detach so G isn’t updated now).
- Forward real: D(real_data) → BCE vs ones; backprop.
- Forward fake: D(fake_data) → BCE vs zeros; backprop.
- Optimizer step updates D.
- Return total D loss and D’s real/fake predictions.

7) Train step — Generator (per batch)
- Make fresh fake_data = G(noise).
- Score with D: D(fake_data).
- BCE vs ones (the generator wants D to say “real”).
- Backprop through G and D; optimizer step updates G.
- Return G loss.

8) Training loop
- For 200 epochs:
  - For each batch: train D, then train G.
  - Log d_error and g_error every batch.
  - Every 100 batches: generate 16 images from a fixed test_noise, log/save them, and print a status line (uses the provided Logger).

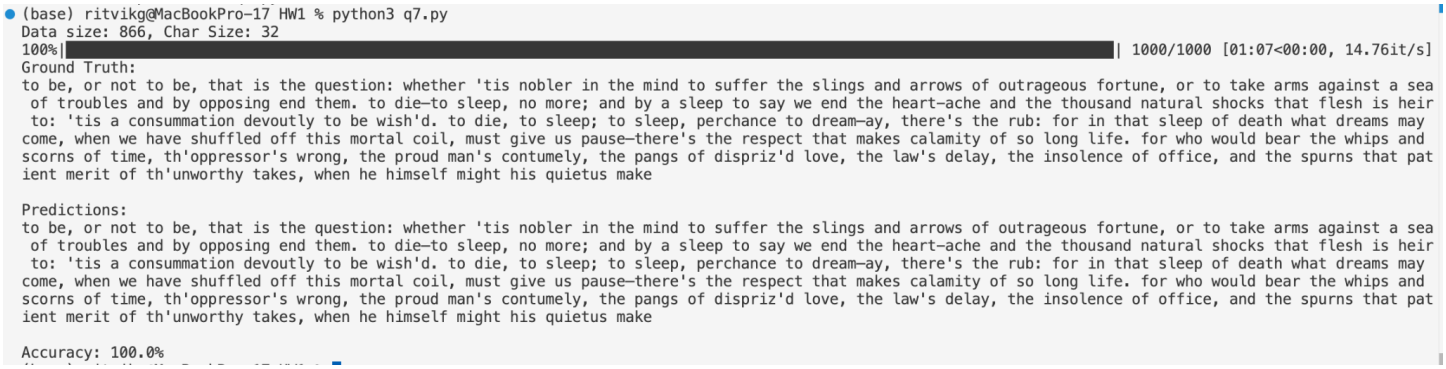
9) What it learns (intuition)
- D gets better at telling real MNIST from G’s fakes.
- G gets better at making fakes that fool D.
- Over time, the sample images start looking like handwritten digits.

Question - 7

Read the article in the link below. Use the codes in the article to reproduce the results given in it for the LSTM networks. Submit the results, your code, and a detailed step-by-step description of the algorithm

Code Link - Link

Results -



Step-By-Step Description -

This code runs an LSTM from the article

1) Goal
Teach a tiny LSTM to predict the next character in a chunk of Shakespeare. If it sees “to be,” it tries to guess the very next character.

2) Make the dataset
- Take the Hamlet quote as one long lowercase string.
- Build a vocabulary of all unique characters.
- Map each character ↔ index.
- Create pairs:
  X = all chars except the last
  y = all chars except the first
  (so at every position we try to predict “the next char”.)

3) One-hot inputs
- Convert each character in X into a one-hot column vector of size vocab_size.
- These one-hot vectors are what the network reads at each time step.

4) Initialize weights (Xavier)
- LSTM has separate weight/bias sets for 4 gates:
  forget (Wf, bf), input (Wi, bi), candidate (Wc, bc), output (Wo, bo).
- There’s also an output projection (Wy, by) that turns the hidden state into raw class scores (logits).
- Each gate uses the concatenated vector z_t = [h_{t-1}; x_t], so input_size = hidden_size + vocab_size.

5) Forward pass (time step t)
- Compute gates:
  f_t = σ(Wf·z_t + bf) # what to keep from old cell
  i_t = σ(Wi·z_t + bi) # how much new info to write
  g_t = tanh(Wc·z_t + bc) # candidate new content
  o_t = σ(Wo·z_t + bo) # how much of cell to expose

- Update memory and hidden:
  $c_t = f_t \odot c_{t-1} + i_t \odot g_t$
  $h_t = o_t \odot \tanh(c_t)$
- Predict next char:
  $y_t = W_y \cdot h_t + b_y$
  $p_t = \text{softmax}(y_t)$

6) Loss signal per step
- Target is the true next char (from y), represented as one-hot.
- The gradient wrt logits is: (one_hot(target) − p_t).
  (This is the standard softmax + cross-entropy gradient.)

7) Backprop through time (BPTT)
- Walk backward through the sequence:
  - Accumulate gradients for Wy, by from the output errors.
  - Backprop the error into h_t, then into o_t, c_t, f_t, i_t, g_t using the gate derivatives:
    $\sigma'(x) = \sigma(x)(1-\sigma(x))$, $\tanh'(x) = 1-\tanh(x)^2$.
  - From gate errors, accumulate dWf, dWi, dWc, dWo and their biases using the same $z_t = [h_{t-1}; x_t]$.
  - Pass “next” gradients (for h_{t-1} and c_{t-1}) to the previous time step.

8) Update weights
- After summing grads over all time steps in the sequence:
  $W \leftarrow W + lr * dW$ (sign matches the error definition above)
  $b \leftarrow b + lr * db$

9) Training loop
- Repeat for many epochs:
  - Forward through the whole text once (teacher forcing: always feed the true current char).
  - Build the per-step errors from (one_hot − softmax).
  - BPTT to get gradients, then update all parameters.
  - Show a progress bar while training.

10) Testing / sampling
- Run a forward pass over the same sequence.
- At each step, take argmax of p_t to get the predicted character.
- Stitch predictions into a string and compute accuracy = % of positions where predicted char == true next char.