MM 811 Programming Assignment: Autoregressive Model

Mahbubur Rahman (mahbubu1@ualberta.ca)
Student ID: 1870909

In this assignment, four questions have been answered. All the source codes and running results are available online: `https://github.com/Durlov1603025/MM-811-Programming-Assignment.git`

1 Question 1

The goal of this task is to implement an image classification model using a Convolutional Neural Network (CNN) on the MNIST dataset. The model should achieve an accuracy greater than 90% while maintaining a compact architecture. The code was divided into three main parts: `model.py`, `train_eval.py`, and `main.py`, which are described below.

The file **`model.py`** defines the architecture used for MNIST digit classification. The network consists of three convolutional layers followed by two fully connected layers. Each convolutional layer is followed by Batch Normalization and ReLU activation to improve learning stability.

The core architecture was implemented as follows:

```python
import torch
import torch.nn as nn

class ConvNet(nn.Module):
    def __init__(self, in_channels=1, num_classes=10):
        """
        Architecture for MNIST classification.
        :param in_channels: number of input channels (1 for grayscale)
        :param num_classes: number of output classes (10 digits)
        """

        super().__init__()


        # Feature extraction layers
        self.features = nn.Sequential(
            nn.Conv2d(in_channels, 32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2),

            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2),

            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.AdaptiveAvgPool2d((4, 4))
        )

        # Classification layers
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Dropout(0.4),
```

```
            nn.Linear(128 * 4 * 4, 256),
            nn.ReLU(inplace=True),
            nn.Dropout(0.3),
            nn.Linear(256, num_classes)
        )

    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
        return x
```

The file `train_eval.py` includes helper functions for model training and validation. The `train_epoch()` function iterates through all training batches, computes loss using CrossEntropyLoss, and updates model weights with backpropagation. The evaluation function `evaluate()` is similar but runs in torch.no_grad() mode to disable gradient computation. This function measures accuracy on the test dataset.

The source code of `train_eval.py` is given below:

```
import torch
import torch.nn as nn

# ---------------------------------------------------------
# Training and evaluation utilities
# ---------------------------------------------------------


def train_epoch(model, device, dataloader, optimizer, criterion):
    """
    Train the model for one epoch on the training dataset.

    """
    model.train()
    total_loss, correct, total = 0.0, 0, 0

    for images, targets in dataloader:
        images, targets = images.to(device), targets.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()

        total_loss += loss.item() * images.size(0)
        _, pred = outputs.max(1)
        correct += pred.eq(targets).sum().item()
        total += images.size(0)

    avg_loss = total_loss / total
    acc = correct / total * 100.0
    return avg_loss, acc


def evaluate(model, device, dataloader, criterion):
    """
    Evaluate model on validation or test dataset.

    """
    model.eval()
    total_loss, correct, total = 0.0, 0, 0

    with torch.no_grad():
        for images, targets in dataloader:
            images, targets = images.to(device), targets.to(device)
            outputs = model(images)
            loss = criterion(outputs, targets)

            total_loss += loss.item() * images.size(0)
            _, pred = outputs.max(1)
            correct += pred.eq(targets).sum().item()
            total += images.size(0)

```

```python
    avg_loss = total_loss / total
    acc = correct / total * 100.0
    return avg_loss, acc
```

The model accuracy is computed using the equation:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Samples}} \times 100 \tag{1}$$

The `main.py` script acts as the entry point of the project. It initializes the dataset, model, loss function, and optimizer, and coordinates the training and evaluation process.

The **`main.py`** is implemented as following:

```python
import argparse
import torch
import torch.optim as optim
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms
from datetime import datetime

from model import ConvNet
from train_eval import train_epoch, evaluate


def main(args):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print("Using device:", device)

    # Data preprocessing
    transform = transforms.Compose([
        transforms.Resize((32, 32)),
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ])

    # Load MNIST dataset
    train_ds = datasets.MNIST(root="./data", train=True, download=True, transform=transform)
    test_ds = datasets.MNIST(root="./data", train=False, download=True, transform=transform)

    if args.subset is not None and args.subset > 0:
        train_ds = Subset(train_ds, list(range(args.subset)))
        print(f"Using subset of {len(train_ds)} images for training")

    train_loader = DataLoader(train_ds, batch_size=args.batch_size, shuffle=True)
    test_loader = DataLoader(test_ds, batch_size=args.batch_size, shuffle=False)

    model = ConvNet(in_channels=1, num_classes=10).to(device)
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=args.lr, momentum=0.9)
    scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=3, gamma=0.5)

    best_acc = 0.0
    for epoch in range(1, args.epochs + 1):
        t0 = datetime.now()
        train_loss, train_acc = train_epoch(model, device, train_loader, optimizer, criterion)
        test_loss, test_acc = evaluate(model, device, test_loader, criterion)
        scheduler.step()

        print(f"Epoch {epoch:02d}: Train acc {train_acc:.2f}% | Test acc {test_acc:.2f}%")
        print(f"Train loss {train_loss:.4f} | Test loss {test_loss:.4f}")

        if test_acc > best_acc:
            best_acc = test_acc
            torch.save(model.state_dict(), "best_model.pth")

        print(f"Epoch time: {(datetime.now() - t0).total_seconds():.1f}s\n")

    print(f"Training complete! Best test accuracy: {best_acc:.2f}%")


if __name__ == "__main__":
```

```python
    parser = argparse.ArgumentParser()
    parser.add_argument("--epochs", type=int, default=5)
    parser.add_argument("--batch-size", type=int, default=128)
    parser.add_argument("--lr", type=float, default=0.01)
    parser.add_argument("--subset", type=int, default=None)
    args = parser.parse_args()

    main(args)
```

A screenshot of program output is demonstrated in Fig. 1.

```
E:\MM 811\Ass1\Q1>python main.py
Using device: cuda
100%|█| 9.91M/9.91M [00:03<00:00,
100%|█| 28.9k/28.9k [00:00<00:00,
100%|█| 1.65M/1.65M [00:00<00:00,
100%|█| 4.54k/4.54k [00:00<?, ?B/
Epoch 01: Train acc 93.16% | Test acc 98.38%
Train loss 0.2316 | Test loss 0.0466
Epoch time: 9.5s

Epoch 02: Train acc 97.95% | Test acc 98.99%
Train loss 0.0688 | Test loss 0.0298
Epoch time: 10.7s

Epoch 03: Train acc 98.42% | Test acc 99.09%
Train loss 0.0521 | Test loss 0.0291
Epoch time: 10.4s

Epoch 04: Train acc 98.82% | Test acc 99.18%
Train loss 0.0388 | Test loss 0.0258
Epoch time: 10.7s

Epoch 05: Train acc 98.93% | Test acc 99.23%
Train loss 0.0343 | Test loss 0.0245
Epoch time: 15.3s

Training complete! Best test accuracy: 99.23%
```

Figure 1: Output of Question 1

2 Question 2

The binary autoencoder was implemented in the file `autoencoder_model.py`. The architecture consists of three main parts: an encoder, a binarizer, and a decoder. The encoder compresses the input image of size $1 \times 32 \times 32$ into a smaller latent feature map of size $3 \times 4 \times 4$ using a series of convolutional, batch normalization, and ReLU layers. Immediately after the encoder, the latent representation is passed through a binarization function that converts the continuous values into discrete $\{-1, +1\}$ activations using a Straight-Through Estimator (STE) [1]. The decoder reconstructs the original image from these binary latent values using transposed convolutional layers. A final `tanh` activation ensures that the reconstructed image values lie within the normalized range $[-1, 1]$. The source code (***autoencoder_model.py***) is shown as follows:

```python
import torch
import torch.nn as nn

# BinarizeFunction
class BinarizeFunction(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x):
        return x.sign()  # outputs -1 or +1

    @staticmethod
    def backward(ctx, grad_output):
        return grad_output  # straight-through estimator


def binarize(x: torch.Tensor) -> torch.Tensor:
    return BinarizeFunction.apply(x)


class AutoEncoder(nn.Module):
    def __init__(self, use_binary: bool = True):
        super().__init__()
        self.use_binary = use_binary

        # Encoder: 1x32x32 -> 3x4x4
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 32, 3, 1, 1),  # 1x32x32 -> 32x32x32
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.Conv2d(32, 64, 4, 2, 1),  # -> 64x16x16
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 128, 4, 2, 1),  # -> 128x8x8
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.Conv2d(128, 256, 4, 2, 1),  # -> 256x4x4
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 3, 3, 1, 1),  # -> 3x4x4 (latent logits)
        )

        # Decoder: 3x4x4 -> 1x32x32
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(3, 128, 4, 2, 1),  # -> 128x8x8
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(128, 64, 4, 2, 1),  # -> 64x16x16
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(64, 32, 4, 2, 1),  # -> 32x32x32
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.Conv2d(32, 1, 3, 1, 1),  # -> 1x32x32
            nn.Tanh(),  # output values in {-1, +1}
        )

    def forward(self, x):
        latent_logits = self.encoder(x)  # (N, 3, 4, 4), real-valued
```

```python
        if self.use_binary:
            latent = binarize(latent_logits)  # strictly +1 or -1
        else:
            latent = latent_logits  # warm-up (no binarization)
        recon = self.decoder(latent)
        # Return both for training utilities
        return recon, latent, latent_logits
```

The training and evaluation procedures are implemented in the file `train_autoencoder.py`. Two key functions are defined: `train_epoch()` and `evaluate()`. The `train_epoch()` function performs one full epoch of training using either L1 or MSE reconstruction loss.

The `evaluate()` function computes the average PSNR on the dataset using the functions `compute_psnr_torch()` and `compute_psnr_sigs()`. The source code of `train_autoencoder.py` is given below:

```python
import torch
import torch.nn.functional as F


def compute_psnr_torch(imgs, refs, eps=1e-8):
    mse = F.mse_loss(imgs, refs, reduction='none').mean(dim=[1, 2, 3])
    psnr = 20 * torch.log10(255.0 / torch.sqrt(mse + eps))
    return psnr


def compute_psnr_sigs(imgs, refs):
    imgs_norm = imgs + 1.0
    refs_norm = refs + 1.0
    imgs_norm *= 0.5
    refs_norm *= 0.5
    imgs_norm *= 255.0
    refs_norm *= 255.0
    psnr_vals = compute_psnr_torch(imgs_norm, refs_norm)
    return psnr_vals


def train_epoch(model, dataloader, device, optimizer, *,
                use_l1=True, reg_lambda=1e-3, push_margin=1.0):
    model.train()
    total_loss = 0.0
    total_psnr = 0.0

    for imgs, _ in dataloader:
        imgs = imgs.to(device)
        imgs = imgs * 2.0 - 1.0

        optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
        optimizer.zero_grad(set_to_none=True)
        recon, binary_latent, latent_logits = model(imgs)

        # Reconstruction loss
        if use_l1:
            rec_loss = F.l1_loss(recon, imgs)
        else:
            rec_loss = F.mse_loss(recon, imgs)

        commit = F.relu(push_margin - latent_logits.abs()).mean()
        loss = rec_loss + reg_lambda * commit

        loss.backward()
        optimizer.step()

        total_loss += rec_loss.item()
        total_psnr += compute_psnr_sigs(recon.detach(), imgs).mean().item()

    avg_loss = total_loss / len(dataloader)
    avg_psnr = total_psnr / len(dataloader)
    return avg_loss, avg_psnr


def evaluate(model, dataloader, device):
```

```python
    model.eval()
    total_psnr = 0.0
    with torch.no_grad():
        for imgs, _ in dataloader:
            imgs = imgs.to(device)
            imgs = imgs * 2.0 - 1.0
            recon, _, _ = model(imgs)
            total_psnr += compute_psnr_sigs(recon, imgs).mean().item()

    avg_psnr = total_psnr / len(dataloader)
    return avg_psnr
```

The file `main.py` manages the data loading, model initialization, and training loop. During the first few epochs (warm-up phase), binarization is disabled to allow the decoder to learn from continuous latent features. After the warm-up period, binarization is activated so that the latent space becomes strictly $\{-1, +1\}$. The model is trained and evaluated on the same subset of the MNIST dataset to test its memorization capability. The best-performing model is automatically saved based on PSNR improvement. The source code of `main.py` is given below:

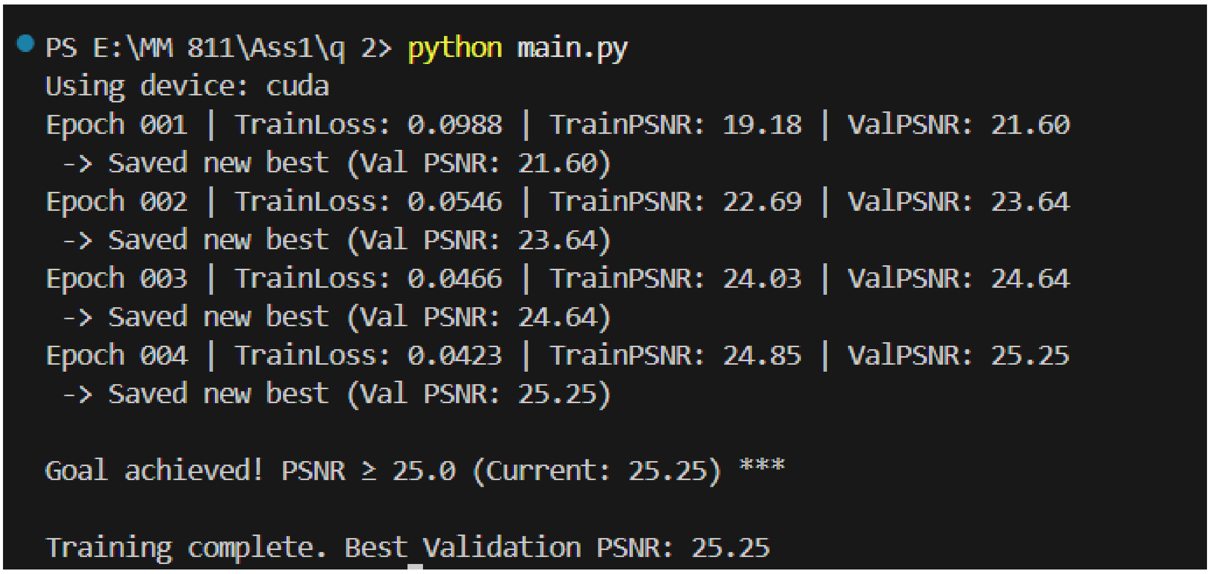
```python
import argparse
import torch
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

from autoencoder_model import AutoEncoder
from train_autoencoder import train_epoch, evaluate

def get_dataloaders(batch_size: int, num_workers: int = 2):
    tfm = transforms.Compose([
        transforms.Resize(32),
        transforms.ToTensor(),

    ])
    train_ds = datasets.MNIST(root='./data', train=True, download=True, transform=tfm)
    test_ds = train_ds

    train_loader = DataLoader(
        train_ds, batch_size=batch_size, shuffle=True,
        num_workers=num_workers, pin_memory=True
    )
    val_loader = DataLoader(
        test_ds, batch_size=batch_size, shuffle=False,
        num_workers=num_workers, pin_memory=True
    )
    return train_loader, val_loader


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--epochs', type=int, default=200)
    parser.add_argument('--warmup_epochs', type=int, default=10, help='Train without binarization for first N epochs')
    parser.add_argument('--batch_size', type=int, default=128)
    parser.add_argument('--lr', type=float, default=1e-3)
    parser.add_argument('--reg_lambda', type=float, default=1e-3, help='Weight for latent push-away regularizer')
    parser.add_argument('--use_l1', action='store_true', help='Use L1 loss (default off if flag absent)')
    parser.add_argument('--no_l1', action='store_true', help='Force MSE even if --use_l1 set elsewhere')
    args = parser.parse_args()

    # Resolve L1 flag
    use_l1 = True
    if args.no_l1:
        use_l1 = False
    elif args.use_l1:
        use_l1 = True

    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    print('Using device:', device)

```

```python
    train_loader, val_loader = get_dataloaders(args.batch_size)

    model = AutoEncoder(use_binary=False)  # start with warm-up by default
    model.to(device)

    optimizer = optim.Adam(model.parameters(), lr=args.lr)
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(
        optimizer, mode='max', factor=0.5, patience=10
)


    best_psnr = 0.0

    for epoch in range(1, args.epochs + 1):
        # Toggle binarization after warmup period
        if epoch == args.warmup_epochs + 1:
            model.use_binary = True


        train_loss, train_psnr = train_epoch(
            model, train_loader, device, optimizer,
            use_l1=use_l1, reg_lambda=args.reg_lambda
        )
        val_psnr = evaluate(model, val_loader, device)
        scheduler.step(val_psnr)

        print(f"Epoch {epoch:03d} | TrainLoss: {train_loss:.4f} | TrainPSNR: {train_psnr:.2f} | ValPSNR: {val_psnr:.2f}")

        # save best
        if val_psnr > best_psnr:
            best_psnr = val_psnr
            torch.save(model.state_dict(), 'autoencoder_best.pth')
            print(f" -> Saved new best (Val PSNR: {best_psnr:.2f})")

        # Early exit if goal reached
        if val_psnr > 25.0:
            print(f"\nGoal achieved! PSNR  25.0 (Current: {val_psnr:.2f}) ***")
            break

    print(f"\nTraining complete. Best Validation PSNR: {best_psnr:.2f}")



if __name__ == '__main__':
    main()
```

Table 1: AutoEncoder Network Architecture

Part	Layer Type	Kernel / Filter Size	Stride	Padding	Output Size
Encoder	Conv2d(1, 32)	3×3	1	1	$32 \times 32 \times 32$
	Conv2d(32, 64)	4×4	2	1	$64 \times 16 \times 16$
	Conv2d(64, 128)	4×4	2	1	$128 \times 8 \times 8$
	Conv2d(128, 256)	4×4	2	1	$256 \times 4 \times 4$
	Conv2d(256, 3)	3×3	1	1	$3 \times 4 \times 4$
Decoder	ConvTranspose2d(3, 256)	3×3	1	1	$256 \times 4 \times 4$
	ConvTranspose2d(256, 128)	4×4	2	1	$128 \times 8 \times 8$
	ConvTranspose2d(128, 64)	4×4	2	1	$64 \times 16 \times 16$
	ConvTranspose2d(64, 32)	4×4	2	1	$32 \times 32 \times 32$
	Conv2d(32, 1)	3×3	1	1	$1 \times 32 \times 32$

An overview of AutoEncoder Network Architecture can be found from Table 1. The output is shown in Fig. 2



```
PS E:\MM 811\Ass1\q 2> python main.py
Using device: cuda
Epoch 001 | TrainLoss: 0.0988 | TrainPSNR: 19.18 | ValPSNR: 21.60
 -> Saved new best (Val PSNR: 21.60)
Epoch 002 | TrainLoss: 0.0546 | TrainPSNR: 22.69 | ValPSNR: 23.64
 -> Saved new best (Val PSNR: 23.64)
Epoch 003 | TrainLoss: 0.0466 | TrainPSNR: 24.03 | ValPSNR: 24.64
 -> Saved new best (Val PSNR: 24.64)
Epoch 004 | TrainLoss: 0.0423 | TrainPSNR: 24.85 | ValPSNR: 25.25
 -> Saved new best (Val PSNR: 25.25)

Goal achieved! PSNR ≥ 25.0 (Current: 25.25) ***

Training complete. Best Validation PSNR: 25.25
```

Figure 2: Output of question 2

3 Question 3

3.1 Sequence to Token

The goal of this experiment is to train an autoregressive model that can memorize binary latent sequences extracted from the trained AutoEncoder (Question 2) and generate new MNIST-like digit images token by token. The success criterion is achieving more than 80% memorization rate, defined as the proportion of generated sequences that exactly match the training data.

Dataset Preparation

- Uses the **binary latent codes** produced by the trained autoencoder (Question 2).
- Each latent tensor ($3 \times 4 \times 4$) elements is flattened into a 48-token sequence.
- Continuous values in $[-1, 1]$ are mapped to binary values (either +1 or -1) using the transformation.
- Each training sample is split into prefix–target pairs

```
class LatentSequenceDataset(Dataset):
def __init__(self, autoencoder, images, device):
    self.device = device
    autoencoder.eval()

    # Extract binary latent codes
    with torch.no_grad():
        imgs_normalized = images.to(device) * 2 - 1
        _, latents, _ = autoencoder(imgs_normalized)

    self.latents = latents.view(latents.size(0), -1)  # (N, 48), values {-1, +1}
    self.tokens = ((self.latents + 1) / 2).long()

    # Create autoregressive training pairs
    seq_len = self.tokens.size(1)  # 48
    self.inputs, self.labels, self.lengths = [], [], []

    for seq in self.tokens:
        for i in range(1, seq_len):
```

```python
                self.inputs.append(seq[:i])
                self.labels.append(seq[i])
                self.lengths.append(i)

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

    def __getitem__(self, idx):
        x = self.inputs[idx].float()
        y = self.labels[idx].float()
        length = self.lengths[idx]
        return x, y, length


def collate_fn(batch):
    sequences, labels, lengths = zip(*batch)
    # Pad sequences to max length in this batch
    sequences_padded = pad_sequence(sequences, batch_first=True, padding_value=0)
    labels = torch.stack(labels)
    lengths = torch.tensor(lengths)

    return sequences_padded, labels, lengths
```

Model Architecture

The model, named **Seq2TokenLSTM**, is designed for next-token prediction. The model can be defined as:

- **Embedding Layer:** Maps binary tokens into 128-dimensional continuous embeddings.
- **LSTM [2] Layers:** Two stacked layers with hidden size 128, capturing temporal dependencies between tokens.
- **Fully Connected Layers:** Linear(128→64) → ReLU → Linear(64→1) → Sigmoid structure producing the probability for the next token.

Table 2: Seq2Token Network Architecture

Layer Type	Input Size	Output Size	Activation	Notes
Embedding(2, 128)	binary token	128-dim vector	–	Binary token $\in \{-1, +1\}$ embedded into 128-dim vector
LSTM Layer 1	128	128	–	Hidden size = 128, batch_first=True
LSTM Layer 2	128	128	–	num_layers = 2, no dropout
Linear(128, 64)	128	64	ReLU	First fully connected layer
Linear(64, 1)	64	1	Sigmoid	Predicts next binary token probability

An overview of the Sequence to Token Network Architecture is shown in the Table 2. The model can be defined as follows:

```python
class Seq2TokenLSTM(nn.Module):
def __init__(self, input_size=1, hidden_size=128, num_layers=2, dropout=0.0):
    super().__init__()

    self.hidden_size = hidden_size
    self.num_layers = num_layers

    # Embedding layer
    self.embedding = nn.Embedding(num_embeddings=2, embedding_dim=hidden_size)

    # LSTM layers
    self.lstm = nn.LSTM(
        input_size=hidden_size,
        hidden_size=hidden_size,
        num_layers=num_layers,
        batch_first=True,
        dropout=0  # Removed dropout to encourage memorization
    )
```

```python

        # Output layers
        self.fc = nn.Sequential(
            nn.Linear(hidden_size, hidden_size // 2),
            nn.ReLU(),
            nn.Linear(hidden_size // 2, 1),
            nn.Sigmoid()
        )

    def forward(self, x, lengths=None):
        batch_size = x.size(0)

        # Embed tokens
        embedded = self.embedding(x.long())
        if lengths is not None:
            packed = pack_padded_sequence(
                embedded, lengths.cpu(),
                batch_first=True,
                enforce_sorted=False
            )
            lstm_out, (hidden, cell) = self.lstm(packed)
            # Use last hidden state
            output = hidden[-1]
        else:
            # Process full sequence
            lstm_out, (hidden, cell) = self.lstm(embedded)
            output = hidden[-1]

        # Predict next token
        pred = self.fc(output).squeeze(-1)

        return pred
```

Training Procedure:

- Training uses Binary Cross-Entropy (BCE) loss [3]:

$$L = -[y \log(p) + (1-y)\log(1-p)] \tag{2}$$

  where y is the true token and p is the predicted probability.

- Adam optimizer with learning rate 1×10^{-1} is used for stable convergence.

The training process is done by following source code:

```python
def train_seq2token(model, loader, device, epochs=50, lr=1e-3):
optimizer = torch.optim.Adam(model.parameters(), lr=lr)
loss_fn = nn.BCELoss()

history = {'loss': [], 'acc': []}

for epoch in range(1, epochs + 1):
    model.train()
    total_loss = 0.0
    total_correct = 0
    total_samples = 0

    pbar = tqdm(loader, desc=f"Epoch {epoch}/{epochs}", leave=False)
    for x, y, lengths in pbar:
        x, y = x.to(device), y.to(device)
        lengths = lengths.to(device)

        # Forward pass
        pred = model(x, lengths)
        loss = loss_fn(pred, y)

        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
```

```
            optimizer.step()

            # Track metrics
            total_loss += loss.item()
            predicted_class = (pred > 0.5).float()
            total_correct += (predicted_class == y).sum().item()
            total_samples += y.size(0)

            pbar.set_postfix({
                'loss': f'{loss.item():.4f}',
                'acc': f'{total_correct/total_samples:.3f}'
            })

        avg_loss = total_loss / len(loader)
        avg_acc = total_correct / total_samples
        history['loss'].append(avg_loss)
        history['acc'].append(avg_acc)

        print(f"Epoch {epoch:03d} | Loss: {avg_loss:.4f} | Acc: {avg_acc:.4f}")

    return history
```

Token Generation

The trained model generates binary sequences autoregressively. Starting from an empty input, it predicts each token in order:

$$t_i \sim \text{Bernoulli}(P(t_i = 1)) \tag{3}$$

Each new token depends on all previous tokens, ensuring autoregressive dependency. After generating 48 tokens, each sequence is converted back to $\{-1, +1\}$, reshaped into $(3, 4, 4)$, and decoded through the AutoEncoder's decoder to obtain MNIST-like images.

The following function `generate_sequences` autoregressively builds sequences:

```
def sample_sequences(model, device, num_samples=64, seq_len=48, temperature=1.0):
    model.eval()
    generated_tokens = torch.zeros(num_samples, seq_len, device=device)

    with torch.no_grad():
        for i in range(seq_len):
            if i == 0:
                # First token: start with empty sequence
                dummy = torch.zeros(num_samples, 1, device=device)
                probs = torch.ones(num_samples, device=device) * 0.5
            else:
                # Use generated tokens so far
                current_seq = generated_tokens[:, :i]
                lengths = torch.full((num_samples,), i, device=device)
                probs = model(current_seq, lengths)

            # Apply temperature for diversity
            if temperature != 1.0:
                probs = torch.sigmoid(torch.logit(probs) / temperature)

            token = torch.bernoulli(probs)
            generated_tokens[:, i] = token

    return generated_tokens
```

The generated binary sequences are later decoded by the autoencoder's decoder to reconstruct MNIST-like images as follows:

```
def decode_latents(autoencoder, token_samples, device):
    autoencoder.eval()

    # Convert tokens
    latent_samples = token_samples * 2 - 1

    # Reshape to (N, 3, 4, 4) for decoder
```

```python
    latent_samples = latent_samples.view(-1, 3, 4, 4)

    with torch.no_grad():
        imgs = autoencoder.decoder(latent_samples.to(device))

    return imgs
```

Memorization Evaluation

The generated sequences are compared with training sequences to measure memorization. Exact matches are identified using:

```python
def check_memorization(generated_tokens, train_tokens):
    diffs = (generated_tokens.unsqueeze(1) - train_tokens.unsqueeze(0)).abs().sum(dim=-1)

    # A perfect match has distance 0
    matches = (diffs == 0).any(dim=1).float()

    return matches.mean().item()
```

In this experiment, 64 sequences were generated, 57 of which matched training data, giving an **89.06% memorization rate**

The output of this code is shown in Fig. 3

```
Epoch 078 | Loss: 0.1071 | Acc: 0.9445
Epoch 079 | Loss: 0.1057 | Acc: 0.9435
Epoch 080 | Loss: 0.1068 | Acc: 0.9405
Epoch 081 | Loss: 0.1062 | Acc: 0.9418
Epoch 082 | Loss: 0.1059 | Acc: 0.9407
Epoch 083 | Loss: 0.1062 | Acc: 0.9435
Epoch 084 | Loss: 0.1067 | Acc: 0.9446
Epoch 085 | Loss: 0.1074 | Acc: 0.9438
Epoch 086 | Loss: 0.1048 | Acc: 0.9435
Epoch 087 | Loss: 0.1059 | Acc: 0.9433
Epoch 088 | Loss: 0.1041 | Acc: 0.9450
Epoch 089 | Loss: 0.1055 | Acc: 0.9430
Epoch 090 | Loss: 0.1059 | Acc: 0.9423
Epoch 091 | Loss: 0.1134 | Acc: 0.9377
Epoch 092 | Loss: 0.1655 | Acc: 0.9234
Epoch 093 | Loss: 0.1809 | Acc: 0.9166
Epoch 094 | Loss: 0.1757 | Acc: 0.9209
Epoch 095 | Loss: 0.1427 | Acc: 0.9270
Epoch 096 | Loss: 0.1417 | Acc: 0.9325
Epoch 097 | Loss: 0.1263 | Acc: 0.9337
Epoch 098 | Loss: 0.1126 | Acc: 0.9400
Epoch 099 | Loss: 0.1071 | Acc: 0.9415
Epoch 100 | Loss: 0.1036 | Acc: 0.9433

OK - Saved model -> seq2token_lstm.pth

Memorization Ratio: 89.06%
(57/64 generated sequences match training data)


==================================================
EXPERIMENT SUMMARY Sequence to Token
==================================================
Training samples: 128
Training epochs: 100
Final loss: 0.1036
Final accuracy: 0.9433
Generated samples: 64
Memorization rate: 89.06%
==================================================
```

Figure 3: Output of Sequence to Token

3.2 Sequence to Sequence

The objective of this part is designing a direct **sequence-to-sequence (Seq2Seq)** model.

Model Architecture

The model uses a stack of fully-connected residual layers. This model is implemented as:

```
class Seq2Seq(nn.Module):
def __init__(self, seq_len=48, hidden_dim=256, num_blocks=12):
    super().__init__()

    self.seq_len = seq_len

    # Input projection
    self.input_proj = nn.Sequential(
        nn.Linear(seq_len, hidden_dim),
        nn.LayerNorm(hidden_dim),
        nn.GELU()
    )

    # Residual blocks
    self.blocks = nn.ModuleList([
        ResidualFC(hidden_dim, hidden_scale=4)
        for _ in range(num_blocks)
    ])

    # Output projection
    self.output_proj = nn.Sequential(
        nn.Linear(hidden_dim, hidden_dim // 2),
        nn.GELU(),
        nn.Linear(hidden_dim // 2, seq_len),
        nn.Sigmoid()
    )

def forward(self, x):
    # Project to hidden dim
    h = self.input_proj(x)

    # Process through residual blocks
    for block in self.blocks:
        h = block(h)

    # Project to output
    out = self.output_proj(h)

    return out
```

It consists of three major components:

- **Input Projection:** Projects each 48-dimensional binary input into a 256-dimensional hidden representation using a linear layer, layer normalization, and GELU activation.
- **Residual Blocks:** Twelve (or more) `ResidualFC` blocks, each containing two linear layers with a skip connection.
- **Output Projection:** Maps the hidden representation back to a 48-dimensional output that produces probabilities for each token.

The ResidualFC blocks are defined as follows:

```
class ResidualFC(nn.Module):
    def __init__(self, dim, hidden_scale=4):
        super().__init__()
        self.block = nn.Sequential(
            nn.Linear(dim, dim * hidden_scale),
            nn.LayerNorm(dim * hidden_scale),
            nn.GELU(),
            nn.Linear(dim * hidden_scale, dim),
        )

    def forward(self, x):
        return x + self.block(x)
```

An overview of Sequence to Sequence Network Architecture is shown in Table 3.

Table 3: Seq2Seq Network Architecture

Layer Type	**Input Size**	**Output Size**	**Activation**	**Notes**
Linear(48, 256)	48	256	GELU	Input projection + LayerNorm
ResidualFC Block (×12)	256	256	GELU	Each block: Linear(256, 1024) → LN → GELU → Linear(1024, 256)
Linear(256, 128)	256	128	GELU	Output projection (part 1)
Linear(128, 48)	128	48	Sigmoid	Output projection (part 2); reconstructs full sequence

Training Procedure

Training uses the same logic as Part 1 with some key differences: Instead of next-token prediction, the model learns to denoise corrupted binary sequences. The denoising autoencoder framework [4] is used during training. Tokens are randomly flipped with probability noise_prob (default 0.05). This makes the model robust while still memorizing. Cosine Annealing LR scheduler has been used for smooth learning-rate decay.

```python
def train_seq2token(model, loader, device, epochs=50, lr=1e-3):
optimizer = torch.optim.Adam(model.parameters(), lr=lr)
loss_fn = nn.BCELoss()

history = {'loss': [], 'acc': []}

for epoch in range(1, epochs + 1):
    model.train()
    total_loss = 0.0
    total_correct = 0
    total_samples = 0

    pbar = tqdm(loader, desc=f"Epoch {epoch}/{epochs}", leave=False)
    for x, y, lengths in pbar:
        x, y = x.to(device), y.to(device)
        lengths = lengths.to(device)

        # Forward pass
        pred = model(x, lengths)
        loss = loss_fn(pred, y)

        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()

        # Track metrics
        total_loss += loss.item()
        predicted_class = (pred > 0.5).float()
        total_correct += (predicted_class == y).sum().item()
        total_samples += y.size(0)

        pbar.set_postfix({
            'loss': f'{loss.item():.4f}',
            'acc': f'{total_correct/total_samples:.3f}'
        })

    avg_loss = total_loss / len(loader)
    avg_acc = total_correct / total_samples
    history['loss'].append(avg_loss)
    history['acc'].append(avg_acc)

    print(f"Epoch {epoch:03d} | Loss: {avg_loss:.4f} | Acc: {avg_acc:.4f}")

return history
```

Sequence Generation

Unlike the other method, this model generates all tokens at once. Starting from random binary sequences, the network iteratively refines its predictions for several passes until convergence.

```
def generate_sequences_direct(model, device, num_samples=64, seq_len=48):
model.eval()

with torch.no_grad():
    sequences = torch.rand(num_samples, seq_len, device=device)
    sequences = (sequences > 0.5).float()

    # Let model refine (multiple passes for convergence)
    for _ in range(10):
        sequences = model(sequences)
        sequences = (sequences > 0.5).float()

return sequences
```

Each generated sequence is then mapped back to $-1, +1$ and decoded by the AutoEncoder's decoder to reconstruct MNIST-like images.

```
def decode_sequences(autoencoder, token_sequences, device):
autoencoder.eval()
latent_sequences = token_sequences * 2 - 1
latent_sequences = latent_sequences.view(-1, 3, 4, 4)

with torch.no_grad():
    imgs = autoencoder.decoder(latent_sequences.to(device))

return imgs
```

Evaluation and Memorization

The memorization evaluation is almost identical to Part 1. In this experiment, 64 sequences were generated, 60 of which matched training data, giving an **93.75% memorization rate**

The output of Sequence to Sequence is shown in Fig. 4

Question 4

The goal of this experiment is to reproduce the memorization task using a **non-image dataset**. Instead of MNIST images, we employ a **character-level text dataset** and train an autoregressive model to memorize and regenerate the same sentences under extremely limited data conditions using character-level modeling [5].

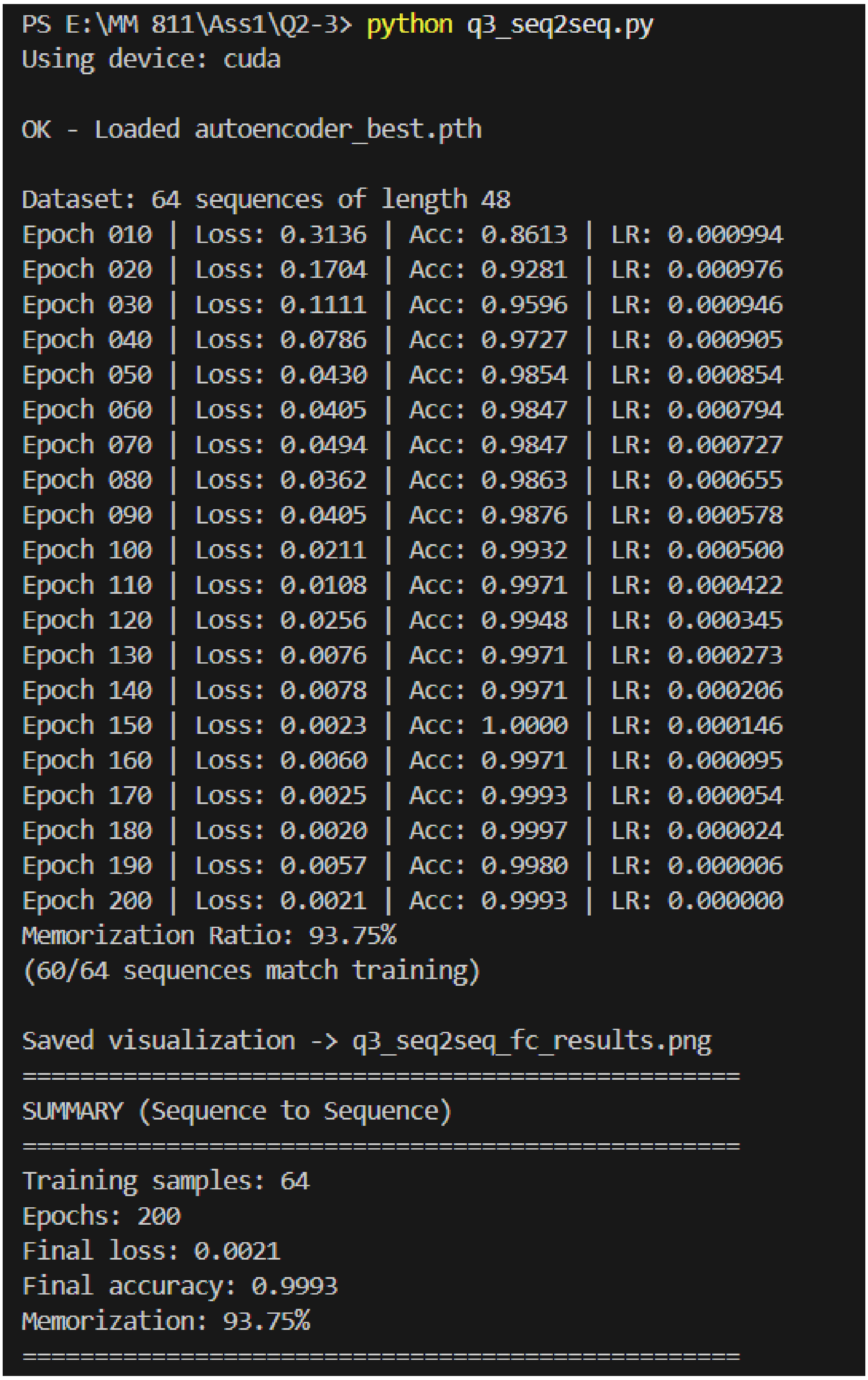
Dataset Preparation

A very small text dataset is used, consisting of two or more sentences. Each sentence is converted into a sequence of characters.

- Each character is mapped to an index (character-level encoding).
- Autoregressive training pairs are created as prefix–next-token pairs.

Model Architecture

The network is a character-level LSTM with a small number of parameters, trained to predict the next token given a sequence. An overview of the architecture with the source code is given in the Table 4:



```
PS E:\MM 811\Ass1\Q2-3> python q3_seq2seq.py
Using device: cuda

OK - Loaded autoencoder_best.pth

Dataset: 64 sequences of length 48
Epoch 010 | Loss: 0.3136 | Acc: 0.8613 | LR: 0.000994
Epoch 020 | Loss: 0.1704 | Acc: 0.9281 | LR: 0.000976
Epoch 030 | Loss: 0.1111 | Acc: 0.9596 | LR: 0.000946
Epoch 040 | Loss: 0.0786 | Acc: 0.9727 | LR: 0.000905
Epoch 050 | Loss: 0.0430 | Acc: 0.9854 | LR: 0.000854
Epoch 060 | Loss: 0.0405 | Acc: 0.9847 | LR: 0.000794
Epoch 070 | Loss: 0.0494 | Acc: 0.9847 | LR: 0.000727
Epoch 080 | Loss: 0.0362 | Acc: 0.9863 | LR: 0.000655
Epoch 090 | Loss: 0.0405 | Acc: 0.9876 | LR: 0.000578
Epoch 100 | Loss: 0.0211 | Acc: 0.9932 | LR: 0.000500
Epoch 110 | Loss: 0.0108 | Acc: 0.9971 | LR: 0.000422
Epoch 120 | Loss: 0.0256 | Acc: 0.9948 | LR: 0.000345
Epoch 130 | Loss: 0.0076 | Acc: 0.9971 | LR: 0.000273
Epoch 140 | Loss: 0.0078 | Acc: 0.9971 | LR: 0.000206
Epoch 150 | Loss: 0.0023 | Acc: 1.0000 | LR: 0.000146
Epoch 160 | Loss: 0.0060 | Acc: 0.9971 | LR: 0.000095
Epoch 170 | Loss: 0.0025 | Acc: 0.9993 | LR: 0.000054
Epoch 180 | Loss: 0.0020 | Acc: 0.9997 | LR: 0.000024
Epoch 190 | Loss: 0.0057 | Acc: 0.9980 | LR: 0.000006
Epoch 200 | Loss: 0.0021 | Acc: 0.9993 | LR: 0.000000
Memorization Ratio: 93.75%
(60/64 sequences match training)

Saved visualization -> q3_seq2seq_fc_results.png
==================================================
SUMMARY (Sequence to Sequence)
==================================================
Training samples: 64
Epochs: 200
Final loss: 0.0021
Final accuracy: 0.9993
Memorization: 93.75%
==================================================
```

Figure 4: Output of Sequence to Sequence

Table 4: Architecture for Text Memorization

Layer Type	**Input Size**	**Output Size**	**Activation**	**Notes**
Embedding	vocab_size	hidden_size	–	Converts each character token into a dense vector.
LSTM (×3 layers)	hidden_size	hidden_size	–	Learns sequential dependencies across characters.
Linear	hidden_size	vocab_size	ReLU / Softmax	Produces probability distribution over next character.

```python
class TextAutoRegressiveLSTM(nn.Module):
def __init__(self, vocab_size, hidden_size=256, num_layers=3):
    super().__init__()

    self.vocab_size = vocab_size
    self.hidden_size = hidden_size
    self.num_layers = num_layers

    # Embedding layer
    self.embedding = nn.Embedding(
        num_embeddings=vocab_size,
        embedding_dim=hidden_size
    )

    # LSTM layers
    self.lstm = nn.LSTM(
        input_size=hidden_size,
        hidden_size=hidden_size,
        num_layers=num_layers,
        batch_first=True,
        dropout=0
    )

    # Output layers - produce distribution over vocab
    self.fc = nn.Sequential(
        nn.Linear(hidden_size, hidden_size),
        nn.ReLU(),
        nn.Linear(hidden_size, vocab_size)  # Output: logits for each character
    )

def forward(self, x, lengths=None):
    # Embed
    embedded = self.embedding(x.long())

    # LSTM
    if lengths is not None:
        from torch.nn.utils.rnn import pack_padded_sequence
        packed = pack_padded_sequence(
            embedded, lengths.cpu(),
            batch_first=True,
            enforce_sorted=False
        )
        lstm_out, (hidden, cell) = self.lstm(packed)
        output = hidden[-1]
    else:
        lstm_out, (hidden, cell) = self.lstm(embedded)
        output = hidden[-1]

    # Predict distribution over vocab
    logits = self.fc(output)
    probs = F.softmax(logits, dim=1)

    return logits, probs
```

Training process

The model is trained using the `CrossEntropyLoss` to predict the next token in each sequence.

- Loss: CrossEntropy between predicted and true next tokens.
- Optimizer: Adam, LR = 1×10^{-3}.
- Scheduler: CosineAnnealingLR for gradual decay.
- Metrics: tracks token-level accuracy and overall loss.

```python
def train_text_model(model, loader, device, epochs=200, lr=1e-3):
    """Train with CrossEntropyLoss (for multi-class distribution)"""
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=epochs)
    loss_fn = nn.CrossEntropyLoss()

    history = {'loss': [], 'acc': []}

    for epoch in range(1, epochs + 1):
        model.train()
        total_loss = 0.0
        total_correct = 0
        total_samples = 0

        pbar = tqdm(loader, desc=f"Epoch {epoch}/{epochs}", leave=False)
        for x, y, lengths in pbar:
            x, y = x.to(device), y.to(device)
            lengths = lengths.to(device)

            # Forward
            logits, probs = model(x, lengths)
            loss = loss_fn(logits, y)

            # Backward
            optimizer.zero_grad()
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
            optimizer.step()

            # Metrics
            total_loss += loss.item()
            predicted = logits.argmax(dim=1)
            total_correct += (predicted == y).sum().item()
            total_samples += y.size(0)

            pbar.set_postfix({
                'loss': f'{loss.item():.3f}',
                'acc': f'{total_correct/total_samples:.3f}'
            })

        scheduler.step()

        avg_loss = total_loss / len(loader)
        avg_acc = total_correct / total_samples
        history['loss'].append(avg_loss)
        history['acc'].append(avg_acc)

        if epoch % 20 == 0 or epoch == epochs:
            print(f"Epoch {epoch:03d} | Loss: {avg_loss:.4f} | Acc: {avg_acc:.4f}")

    return history
```

Generation (Autoregression)

- Starts from initial token.
- Repeatedly feeds generated tokens back to the model to predict the next one.
- Uses **temperature sampling** or **greedy decoding** to control randomness.
- Stops generation when a period (”.”) is predicted.

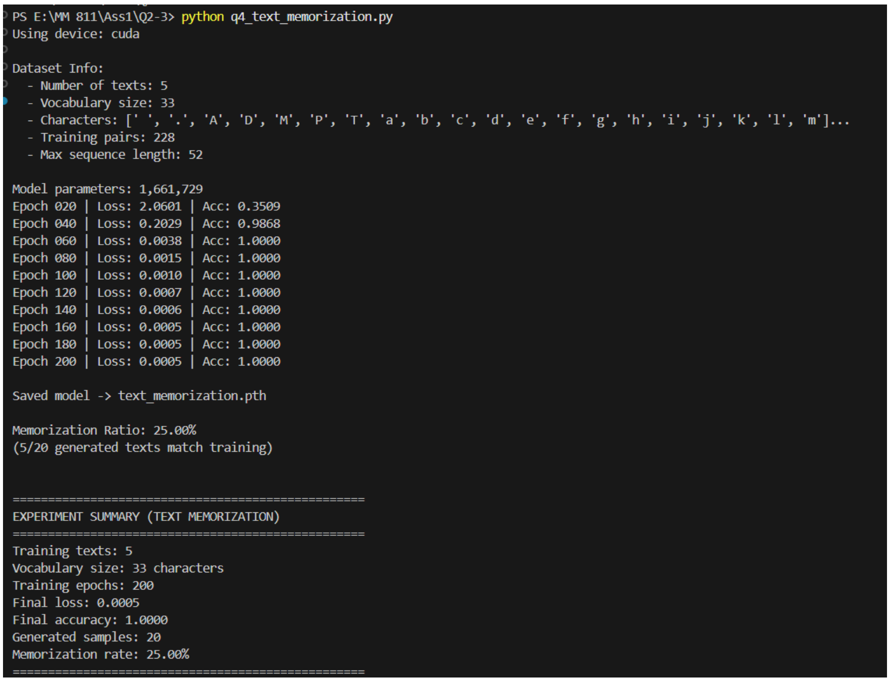
```python
def generate_text_sequences(model, dataset, device, num_samples=10,
                            max_len=50, temperature=0.5):
    model.eval()

    generated_texts = []

    with torch.no_grad():
```

```python
        for sample_idx in range(num_samples):
            generated_tokens = []

            # Use greedy decoding (argmax) for deterministic generation
            start_char_idx = sample_idx % dataset.vocab_size
            generated_tokens.append(start_char_idx)

            for i in range(1, max_len):
                current_seq_input = torch.tensor([generated_tokens], device=device).float()
                length = torch.tensor([len(generated_tokens)], device=device)

                logits, probs = model(current_seq_input, length)

                # Use temperature sampling (low temp = more deterministic)
                if temperature < 0.1:
                    # Greedy: always pick most likely
                    token = logits.argmax(dim=1).item()
                else:
                    # Sample from distribution with temperature
                    logits_scaled = logits / temperature
                    probs_scaled = F.softmax(logits_scaled, dim=1)
                    token = torch.multinomial(probs_scaled[0], num_samples=1).item()

                generated_tokens.append(token)

                # Stop at period
                if dataset.idx2char.get(token, '') == '.':
                    break

            # Decode to text
            text = dataset.decode_sequence(generated_tokens)
            generated_texts.append(text)

    return generated_texts, None
```

Memorization Evaluation

The generated sentences are compared to the original training sentences. Here 5/20 generated texts match training, giving **25% memorization rate**.

The output is shown in Fig. 5



```
PS E:\MM 811\Ass1\Q2-3> python q4_text_memorization.py
Using device: cuda

Dataset Info:
  - Number of texts: 5
  - Vocabulary size: 33
  - Characters: [' ', '.', 'A', 'D', 'M', 'P', 'T', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm']...
  - Training pairs: 228
  - Max sequence length: 52

Model parameters: 1,661,729
Epoch 020 | Loss: 2.0601 | Acc: 0.3509
Epoch 040 | Loss: 0.2029 | Acc: 0.9868
Epoch 060 | Loss: 0.0038 | Acc: 1.0000
Epoch 080 | Loss: 0.0015 | Acc: 1.0000
Epoch 100 | Loss: 0.0010 | Acc: 1.0000
Epoch 120 | Loss: 0.0007 | Acc: 1.0000
Epoch 140 | Loss: 0.0006 | Acc: 1.0000
Epoch 160 | Loss: 0.0005 | Acc: 1.0000
Epoch 180 | Loss: 0.0005 | Acc: 1.0000
Epoch 200 | Loss: 0.0005 | Acc: 1.0000

Saved model -> text_memorization.pth

Memorization Ratio: 25.00%
(5/20 generated texts match training)


==================================================
EXPERIMENT SUMMARY (TEXT MEMORIZATION)
==================================================
Training texts: 5
Vocabulary size: 33 characters
Training epochs: 200
Final loss: 0.0005
Final accuracy: 1.0000
Generated samples: 20
Memorization rate: 25.00%
==================================================
```

Figure 5: Output of question 4

References

[1] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[2] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[3] Maxim Sorokin. Binary Cross-Entropy: Mathematical Insights and Python Implementation — vergotten. `https://medium.com/@vergotten/binary-cross-entropy-mathematical-insights-and-python-implementation-31e5a4df78f3`. [Accessed 21-10-2025].

[4] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103, 2008.

[5] Andrej Karpathy, Justin Johnson, and Li Fei-Fei. Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078*, 2015.