SMAI Assignment 5

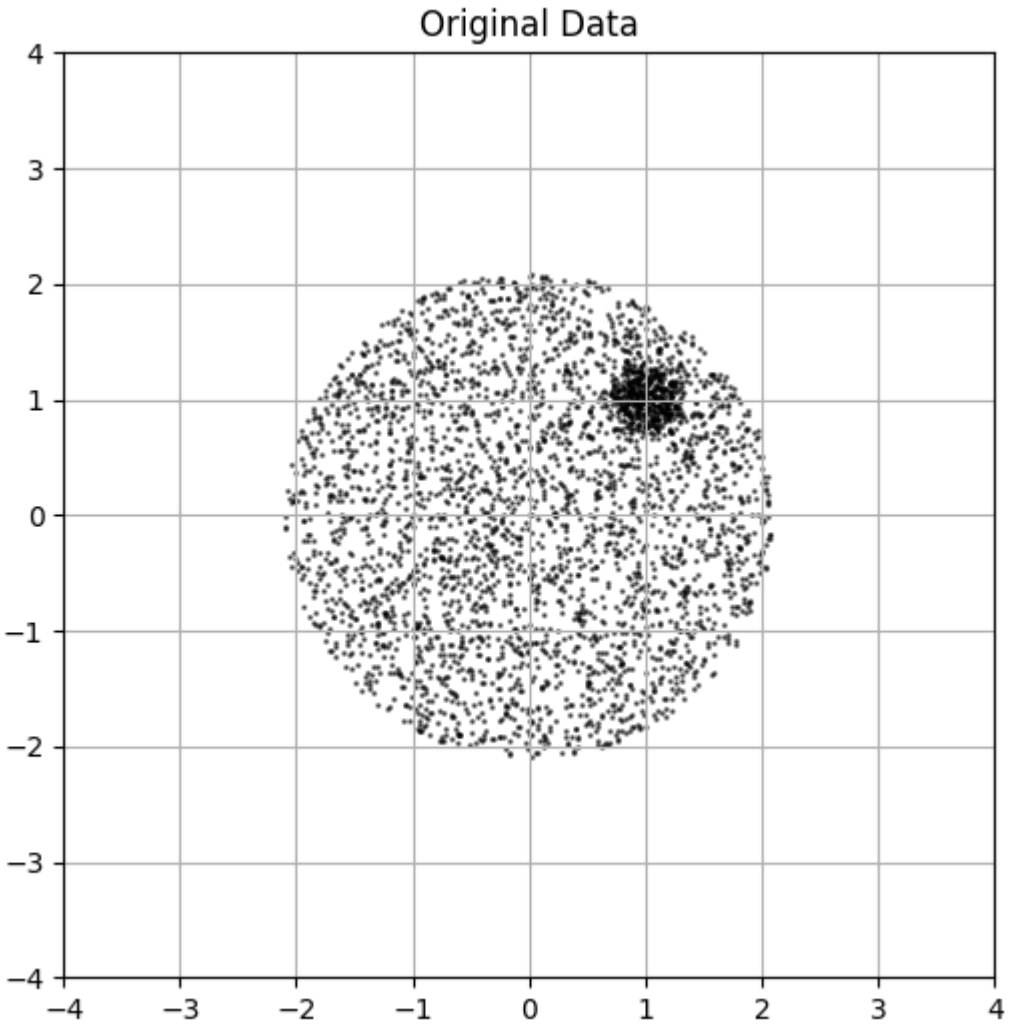
KDE

KDE Class

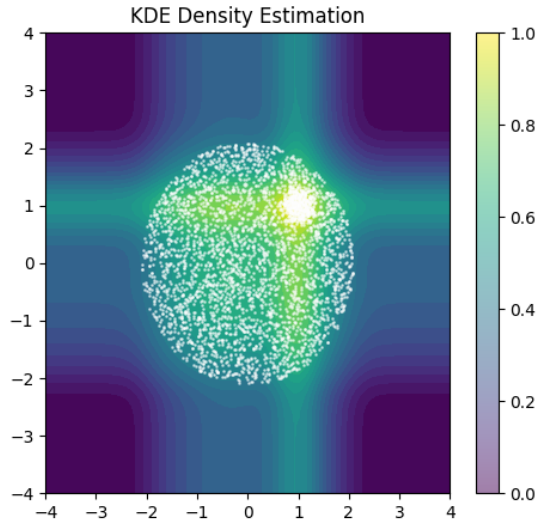
```
class KDE:
    def __init__(self, bandwidth=1.0, kernel='gaussian'):
        self.bandwidth = bandwidth
        self.kernel = kernel
        self.data = None

    def _box_kernel(self, u):
        """Box kernel: Returns 0.5 if |u| <= 1, else 0."""
        return np.where(np.abs(u) <= 1, 0.5, 0)

    def _gaussian_kernel(self, u):
        """Gaussian kernel."""
        return (1 / np.sqrt(2 * np.pi)) * np.exp(-0.5 * u ** 

    def _triangular_kernel(self, u):
        """Triangular kernel: Returns 1 - |u| if |u| <= 1, el
        return np.where(np.abs(u) <= 1, 1 - np.abs(u), 0)

    def _select_kernel(self, u):
        """Selects the kernel function based on the specified
        if self.kernel == 'box':
            return self._box_kernel(u)
        elif self.kernel == 'gaussian':
            return self._gaussian_kernel(u)
        elif self.kernel == 'triangular':
            return self._triangular_kernel(u)
        else:
            raise ValueError("Unsupported kernel type. Choose
             or 'triangular'.")
```

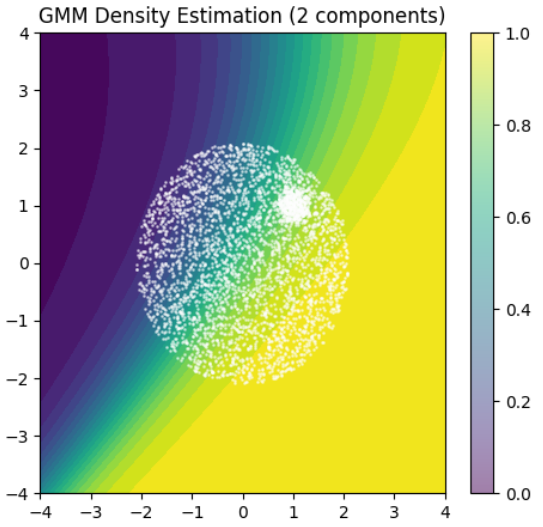
```
    def fit(self, data):
        """Fit the KDE model to data."""
        self.data = data

    def predict(self, x):
        """Calculate the density estimation at a single point
        distances = (x - self.data) / self.bandwidth
        kernel_values = self._select_kernel(distances)
        density = np.sum(kernel_values, axis=0) / (len(self.d
        return np.sum(density)  # Ensure a single density val

    def score_samples(self, points):
        """Calculate the density estimation for multiple poin
        densities = np.array([self.predict(point) for point i
        return np.log(densities)  # Return log density for co
```

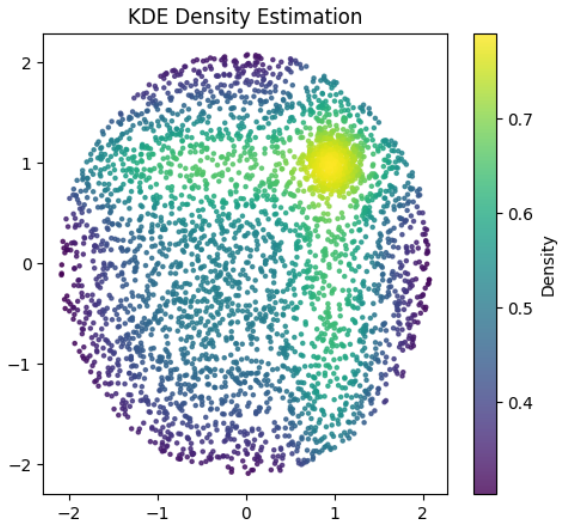
Generate synthetic data

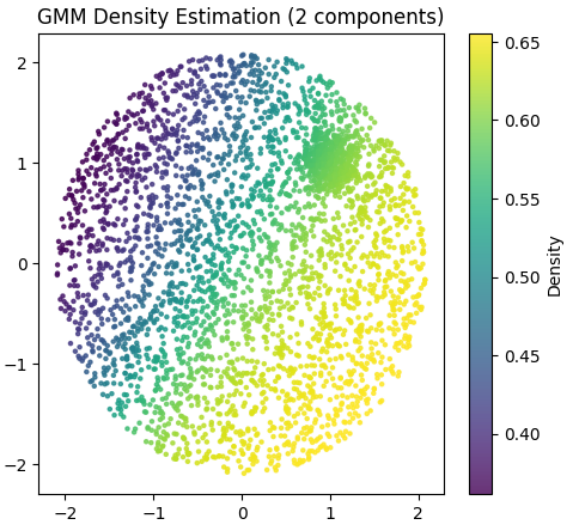

Original Data
4
3
2
1
0
−1
−2
−3
−4
−4
−3
−2
−1
0
1
2
3
4

KDE vs GMM


KDE Density Estimation
4
3
2
1
0
−1
−2
−3
−4
−4
−3
−2
−1
0
1
2
3
4
1.0
0.8
0.6
0.4
0.2
0.0

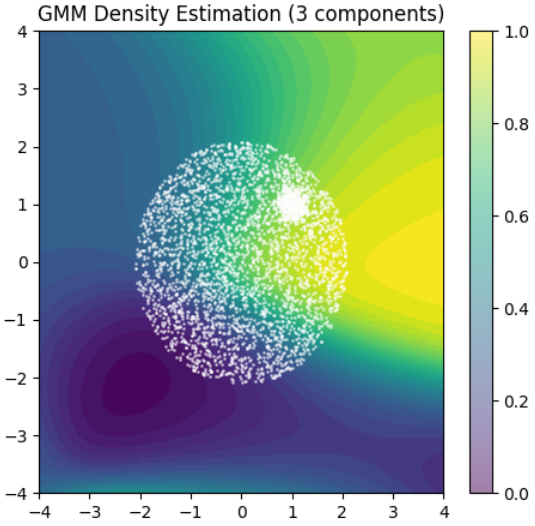

GMM Density Estimation (2 components)
4
3
2
1
0
−1
−2
−3
−4
−4
−3
−2
−1
0
1
2
3
4
1.0
0.8
0.6
0.4
0.2
0.0

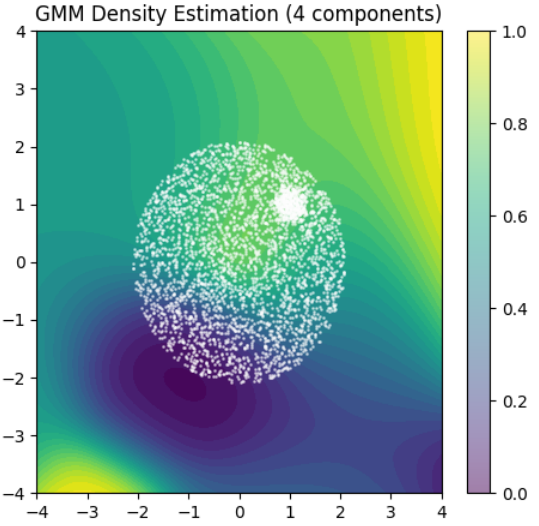


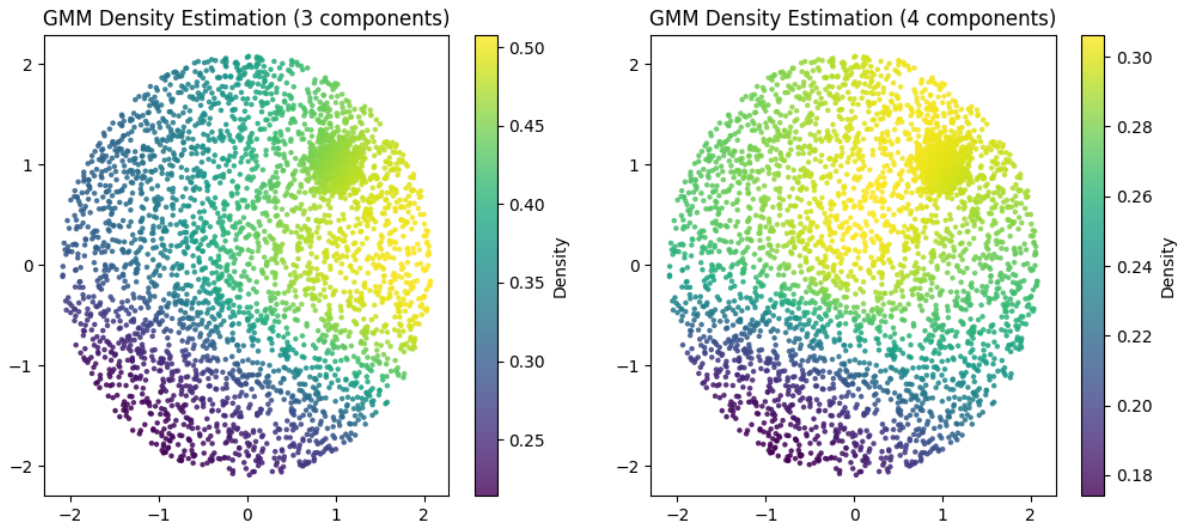


The KDE model estimates the density better.

GMM estimation improves as the number of components are increased

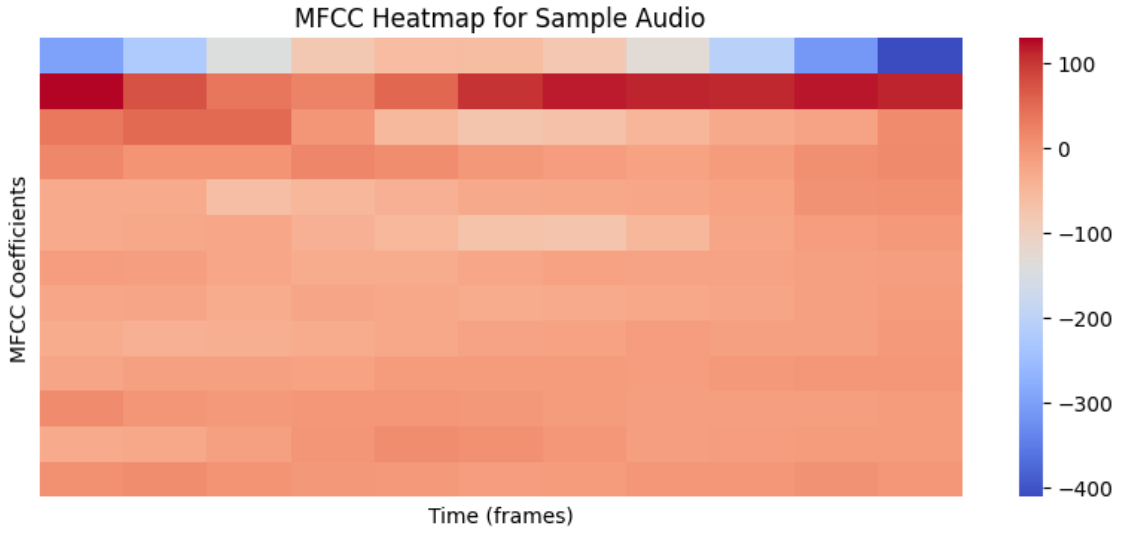






HMM

Dataset



Model Architecture

```
# Initialize HMM models and train them
def train_hmm_models(train_data):
    models = {}
    for digit, features in train_data.items():
        model = hmm.GaussianHMM(n_components=4, covariance_ty
        X = np.vstack(features)  # Stack all MFCC features fo
```

```
        lengths = [len(f) for f in features]  # Lengths of ea
        model.fit(X, lengths)  # Train the HMM model
        models[digit] = model
    return models
```

```
# Predict the digit based on highest HMM model score
def predict_digit(mfcc_features, models):
    max_score = float("-inf")
    best_digit = None
    for digit, model in models.items():
        score = model.score(mfcc_features)
        if score > max_score:
            max_score = score
            best_digit = digit
    return best_digit
```

Metrics

On trained data

```
Test Accuracy: 89.00%
Validation Accuracy: 91.33%
Train Accuracy: 92.71%
```

On own audio

```
File: _2.wav, Predicted Digit: 2
File: _8.wav, Predicted Digit: 8
File: _4.wav, Predicted Digit: 4
File: _3.wav, Predicted Digit: 9
File: _0.wav, Predicted Digit: 1
File: _1.wav, Predicted Digit: 0
File: _6.wav, Predicted Digit: 6
File: _5.wav, Predicted Digit: 5
```

```
File: _7.wav, Predicted Digit: 7
File: _9.wav, Predicted Digit: 9
```

Accuracy = 70 percent

RNN

Counting Bits

Dataset

```
Training examples:
                                        sequence  count_of_ones
75721     [0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1]             7
80184  [0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0]             5
19864                                [1, 0, 0, 1]             2
76699           [1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0]             5
92991     [1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1]             5


Validation examples:
                                     sequence  count_of_ones
52882                    [0, 0, 0, 1, 1, 0, 1]             3
9835                  [0, 1, 1, 1, 0, 0, 0, 0]             3
42963  [0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0]             5
5380                           [0, 0, 1, 1, 1]             3
69603        [1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0]             4


           Test examples:
                             sequence  count_of_ones
           5           [0, 0, 1, 1, 1]             3
           15  [0, 1, 0, 1, 0, 1, 0, 0]             3
           20                      [0]             0
           37             [0, 1, 1, 1]             3
           42  [0, 1, 0, 0, 1, 0, 0, 0]             2
```

Architecture

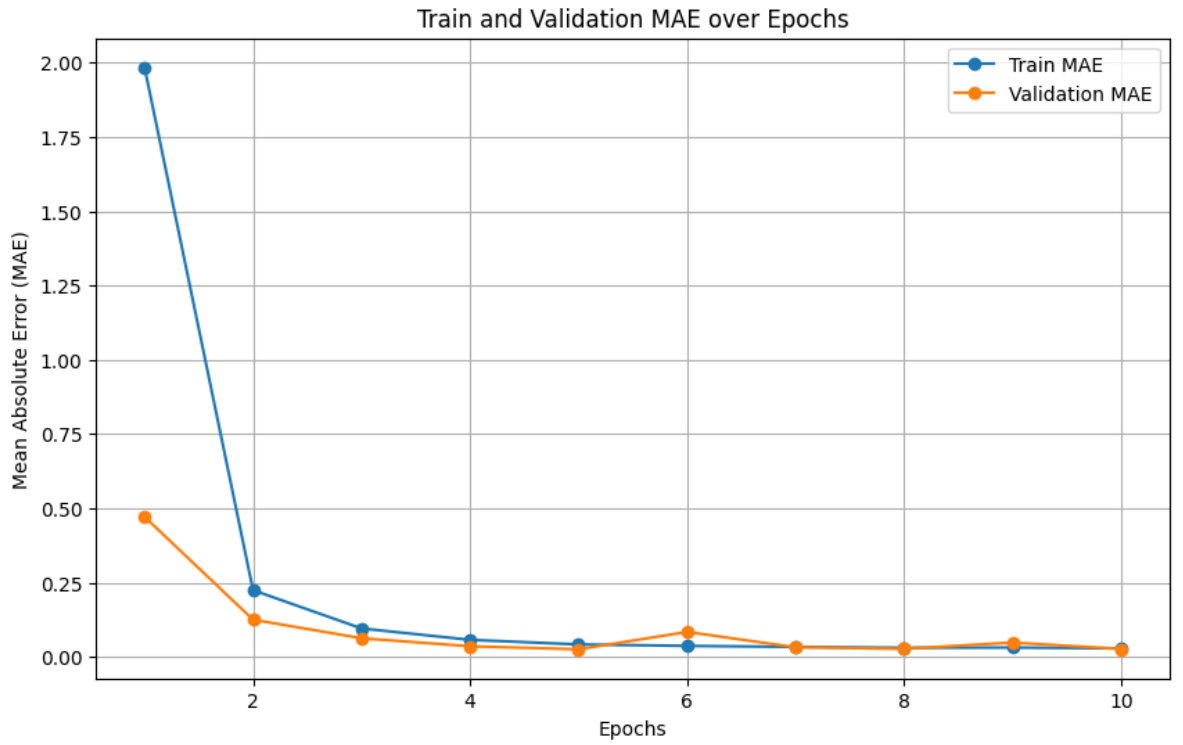
```
# Define the RNN model
class RNNBinaryCounter(nn.Module):
    def __init__(self, input_size=1, hidden_size=16, output_s
        super(RNNBinaryCounter, self).__init__()
        self.hidden_size = hidden_size
```

```
        self.rnn = nn.RNN(input_size, hidden_size, batch_firs
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        # Initialize hidden state with zeros
        h0 = torch.zeros(1, x.size(0), self.hidden_size).to(x

        # Forward propagate through RNN
        out, _ = self.rnn(x.unsqueeze(-1), h0)

        # Use the last output from the RNN for the final coun
        out = self.fc(out[:, -1, :])
        return out
```

Training

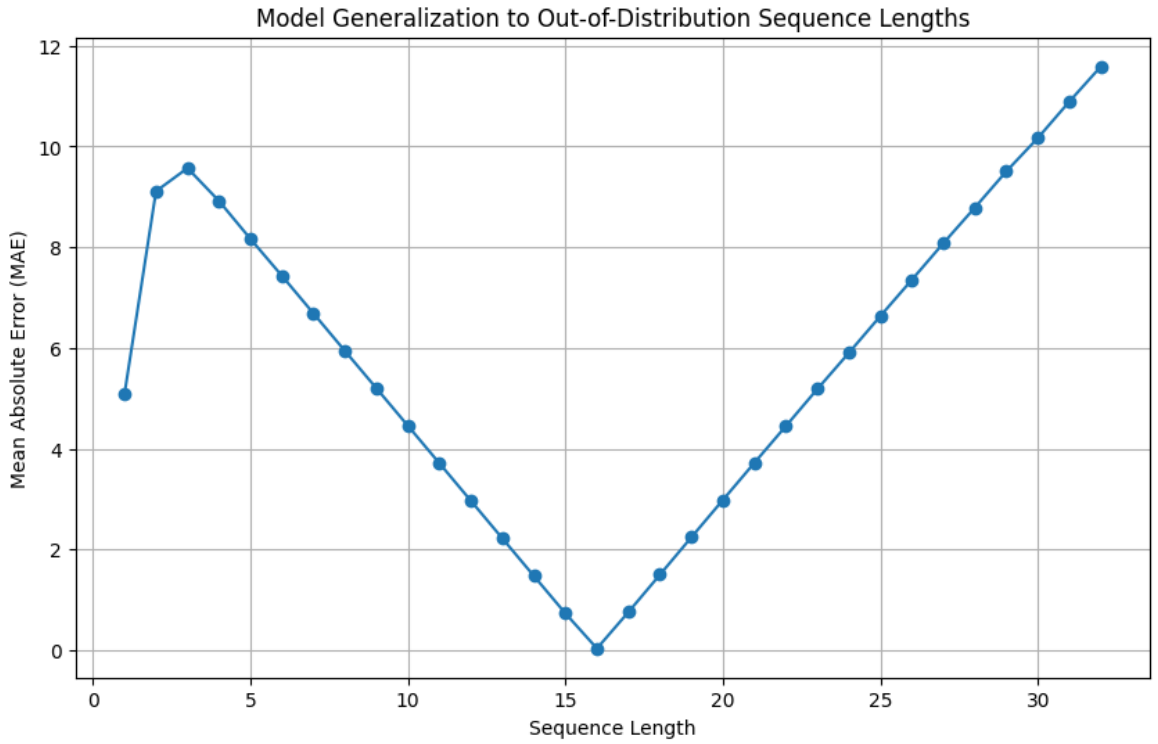
```
# Initialize lists to store MAE values for each epoch
train_mae_history = []
val_mae_history = []

num_epochs = 10
for epoch in range(num_epochs):
    model.train()
    train_loss = 0
    train_mae = 0

    for sequences, labels in train_loader:
        sequences, labels = sequences.to(device), labels.to(d

        # Forward pass
        outputs = model(sequences)
        loss = criterion(outputs.squeeze(), labels)  # MSE lo

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

```
        train_loss += loss.item()

        # Calculate MAE manually for training
        mae = torch.abs(outputs.squeeze() - labels).mean().it
        train_mae += mae
    train_loss /= len(train_loader)
    train_mae /= len(train_loader)
    train_mae_history.append(train_mae)

    # Validation - Calculate MAE while keeping MSE as the los
    model.eval()
    val_loss = 0
    val_mae = 0
    with torch.no_grad():
        for sequences, labels in val_loader:
            sequences, labels = sequences.to(device), labels.
            outputs = model(sequences)

            # Calculate MSE Loss
            loss = criterion(outputs.squeeze(), labels)
            val_loss += loss.item()

            # Calculate MAE manually for validation
            mae = torch.abs(outputs.squeeze() - labels).mean(
            val_mae += mae
    val_loss /= len(val_loader)
    val_mae /= len(val_loader)
    val_mae_history.append(val_mae)

    print(f"Epoch [{epoch+1}/{num_epochs}], Train Loss (MSE):
     Train MAE: {train_mae:.4f}, Val MAE: {val_mae:.4f}")
```




```
Model Test MAE: 0.0269
Random Baseline Test MAE: 5.4843
The model performs better than the random baseline.
```

Generalization



Trend:

Since the model was trained on sequence lengths between 1 and 16, it is expected to perform better on sequences of these lengths, resulting in relatively low MAE values. For sequence lengths beyond this range (17–32), the model is likely to experience a higher MAE as it encounters lengths it has never seen during training.

Interpretation:

Sequence Lengths 1–16: The MAE is lower and stable as the model has learned to count within this range during training. Sequence Lengths 17–32: A progressive increase in MAE is expected as the sequence length increases, as the model has to generalize counting accurately beyond its training distribution.

Optical Character Recognition

Dataset

dispirit.png

freckly.png

hoer.png

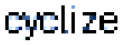
ascertainer.png

bumpily.png

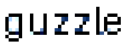
With encoded labels

Label: tensor([9, 14, 20, 5, 18, 3, 1, 18, 4, 9, 14, 1, 12, 0, 0, 0])

Label: tensor([3, 25, 3, 12, 9, 26, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0])



Label: tensor([7, 21, 26, 26, 12, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])



Architecture

```
import torch.nn.functional as F

# Sample CNN-RNN model
class CNNEncoder(nn.Module):
    def __init__(self):
        super(CNNEncoder, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride
=1, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, stri
de=1, padding=1)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        # print(x.shape)
        x = self.pool(F.relu(self.conv2(x)))
        # print(x.shape)
```

```
        # return x.permute(0, 2, 3, 1)  # Batch x Width x H
eight x Channels
        return x
class RNNDecoder(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(RNNDecoder, self).__init__()
        self.rnn = nn.LSTM(input_dim, hidden_dim, batch_fir
st=True, bidirectional=True)
        self.fc = nn.Linear(hidden_dim * 2, output_dim)  #
Bidirectional

    def forward(self, features):

        seq_len = features.size(2) * features.size(3)  # As
suming height x width gives the sequence length
        # print("Features shape before reshape:", features.
shape)

        features = features.reshape(features.size(0), seq_l
en, -1)  # Flatten height and width dimensions
        # print("Features shape after reshape:", features.s
hape)

        output, _ = self.rnn(features)
        # print(output.shape)
        output = self.fc(output)
        # print(output.shape)
        return output  # Character probabilities

class OCRModel(nn.Module):
    def __init__(self, cnn_encoder, rnn_decoder):
        super(OCRModel, self).__init__()
        self.encoder = cnn_encoder
        self.decoder = rnn_decoder

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

```
        output = self.decoder(features)
        return output
```

Training

```
def train(model, train_loader, val_loader, criterion, optim
izer, num_epochs=10):
    for epoch in range(num_epochs):
        model.train()  # Set model to training mode
        total_train_loss = 0
        batch_train_losses = []  # To store loss per batch
in training

        for batch_idx, (images, labels, label_lengths) in e
numerate(train_loader):  # Assuming dataloader provides bat
ches of images and labels
            optimizer.zero_grad()

            # Get the model outputs (shape: batch_size x ma
x_seq_len x num_classes)
            outputs = model(images)  # (N, T, C) where N is
batch size, T is time steps, C is num classes
            outputs = outputs.permute(1, 0, 2)  # (T, N, C)
for CTCLoss (time x batch x classes)

            # Compute the lengths of the input sequences
(i.e., the number of time steps)
            input_lengths = torch.full(size=(images.size
(0),), fill_value=outputs.size(0), dtype=torch.long)

            # Flatten the labels and create target_lengths
(length of each label sequence)
            flattened_labels = torch.cat([label for label i
n labels])  # Concatenate the label tensors
            target_lengths = torch.tensor([len(label) for l
abel in labels], dtype=torch.long)

            # Compute the CTC loss
```

```
            loss = criterion(outputs, flattened_labels, inp
ut_lengths, target_lengths)

            loss.backward()
            optimizer.step()
            total_train_loss += loss.item()  # Accumulate t
he loss for monitoring
            batch_train_losses.append(loss.item())  # Store
loss per batch

        # Print the average training loss for the entire ep
och
        avg_train_loss = abs(total_train_loss) / len(train_
loader)
        print(f"Epoch [{epoch+1}/{num_epochs}], Average Tra
in Loss: {avg_train_loss:.4f}")

        # Now calculate the validation loss
        model.eval()  # Set model to evaluation mode
        total_val_loss = 0
        batch_val_losses = []  # To store loss per batch in
validation
        with torch.no_grad():  # Disable gradients during v
alidation
            for batch_idx, (images, labels, label_lengths)
in enumerate(val_loader):  # Assuming val_loader is availab
le
                # Get the model outputs (shape: batch_size
x max_seq_len x num_classes)
                outputs = model(images)  # (N, T, C) where
N is batch size, T is time steps, C is num classes
                outputs = outputs.permute(1, 0, 2)  # (T,
N, C) for CTCLoss (time x batch x classes)

                # Compute the lengths of the input sequence
s (i.e., the number of time steps)
                input_lengths = torch.full(size=(images.siz
e(0),), fill_value=outputs.size(0), dtype=torch.long)
```

```
                # Flatten the labels and create target_leng
ths (length of each label sequence)
                flattened_labels = torch.cat([label for lab
el in labels])  # Concatenate the label tensors
                target_lengths = torch.tensor([len(label) f
or label in labels], dtype=torch.long)

                # Compute the CTC loss
                loss = criterion(outputs, flattened_labels,
input_lengths, target_lengths)
                total_val_loss += loss.item()  # Accumulate
the validation loss
                batch_val_losses.append(loss.item())  # Sto
re loss per batch

        # Print the average validation loss for the entire
epoch
        avg_val_loss = abs(total_val_loss) / len(val_loade
r)
        print(f"Epoch [{epoch+1}/{num_epochs}], Average Val
idation Loss: {avg_val_loss:.4f}")
```

```
Epoch [1/10], Average Train Loss: 25.7310
Epoch [1/10], Average Validation Loss: 24.5117
Epoch [2/10], Average Train Loss: 20.5848
Epoch [2/10], Average Validation Loss: 22.0605
Epoch [3/10], Average Train Loss: 16.4678
Epoch [3/10], Average Validation Loss: 19.8545
Epoch [4/10], Average Train Loss: 13.1743
Epoch [4/10], Average Validation Loss: 17.8690
Epoch [5/10], Average Train Loss: 10.5394
Epoch [5/10], Average Validation Loss: 16.0821
Epoch [6/10], Average Train Loss: 8.4315
Epoch [6/10], Average Validation Loss: 14.4739
Epoch [7/10], Average Train Loss: 6.7452
Epoch [7/10], Average Validation Loss: 13.0265
Epoch [8/10], Average Train Loss: 5.3962
```

```
Epoch [8/10], Average Validation Loss: 11.7239
Epoch [9/10], Average Train Loss: 4.3169
Epoch [9/10], Average Validation Loss: 10.5515
Epoch [10/10], Average Train Loss: 3.4536
Epoch [10/10], Average Validation Loss: 9.4963
```

Evaluation

Test loader ANCC = 0.6142

Random baseline ANCC = 0.0278

Decoded predictions and ground truth