Jersey Number Recognition

Github repo: https://github.com/1morshed1/jersey-number-recognition

Best Model drive link:
https://drive.google.com/drive/folders/1sxm-wz_52PH0S751joqehYl4h2UXFpu0?usp=sharing

Technical Report link:
https://github.com/1morshed1/jersey-number-recognition/blob/main/TECHNICAL_REPORT.md

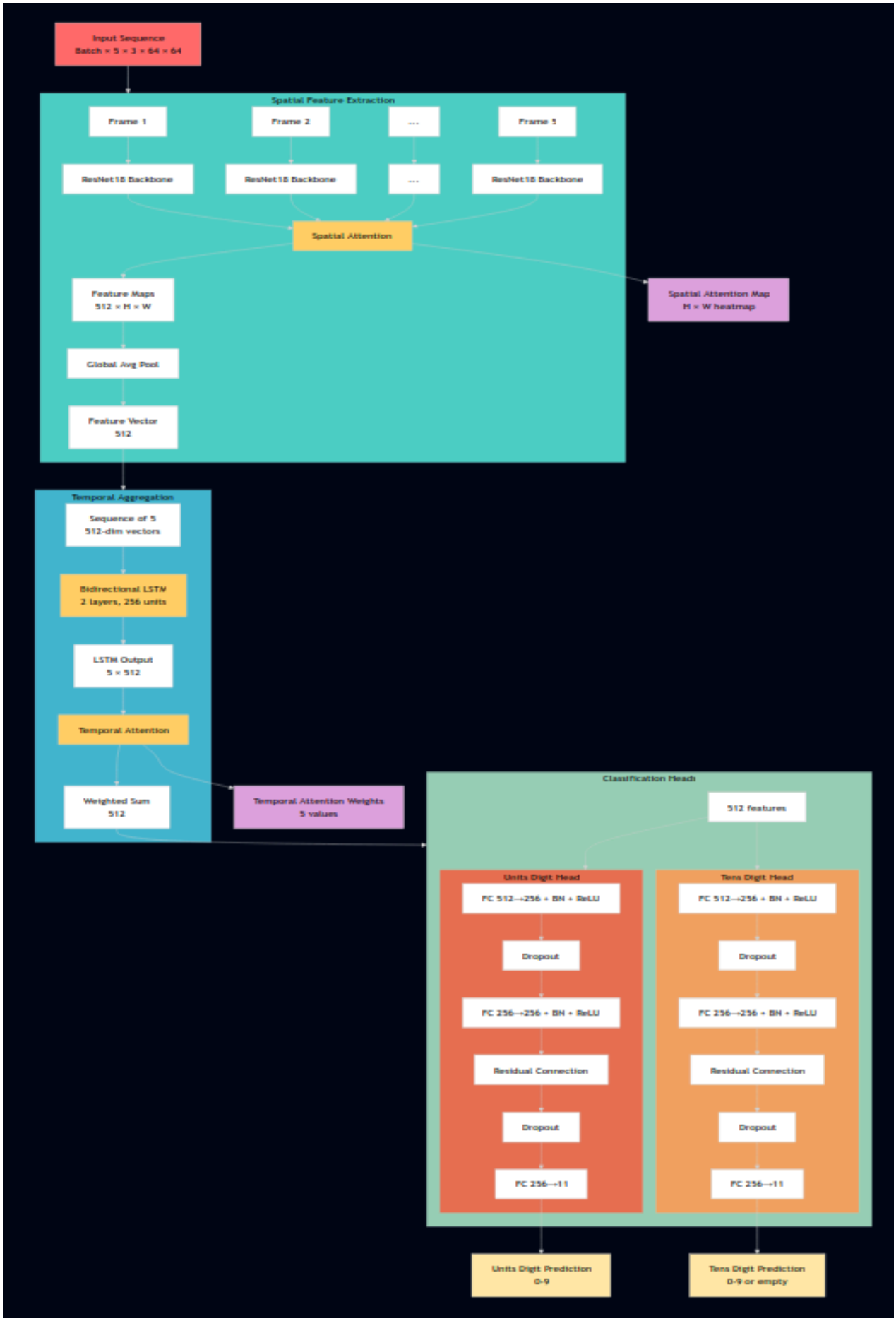
Usage of AI:

For initial planning and validation I used claude pro, deepseek, qwen, kimi, glm 4.6 and cursor agent, Then for finalizing the plan and designing the initial system claude pro, deepseek and cursor was used. Cursor along with its mcp servers(context7, sequential thinking, exa search) was used all the time for checking on the correctness of my code. Cursor's memory bank was used to always have the full context of the project, and I also setup cursorrules so that the code format remains consistent overall. For debugging I used cursor agent, claude pro and deepseek.

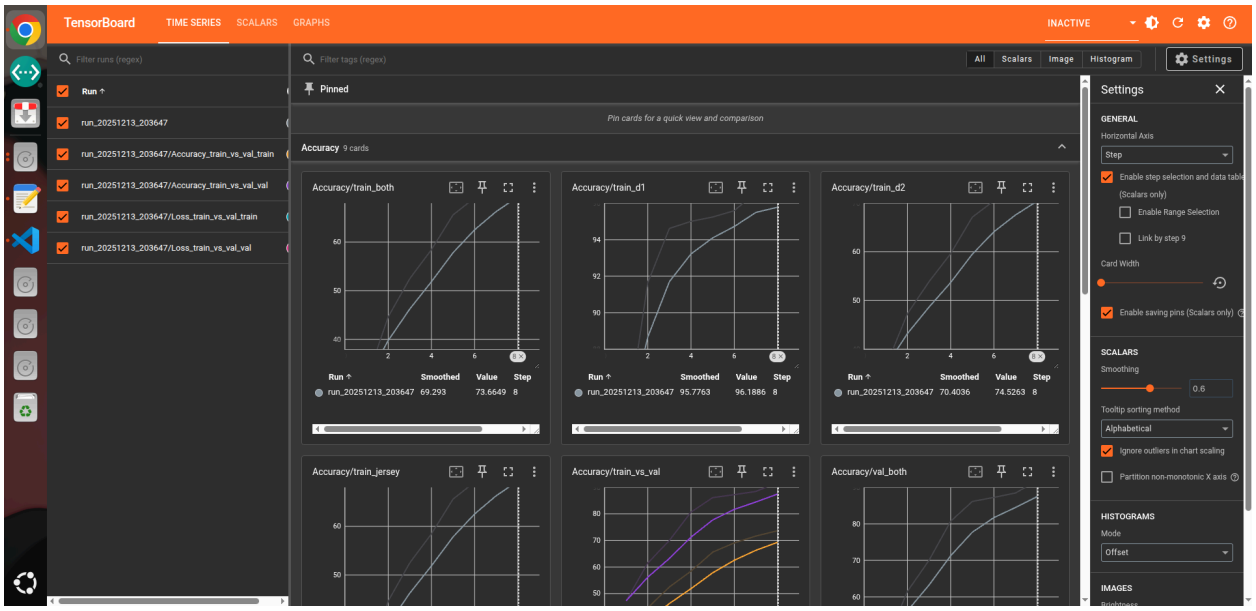
The cursor mcp servers integration helped with my knowledge gap and implementing boilerplate codes. Each script was run through over and over in claude pro, deepseek and cursor , in order to maintain quality. Finally glm 4.6 used via CLine extension was used for file indexing , so that all the AI's which do not have knowledge bank like cursor can have full context to work with each time i open them. Also Claude Desktop along with mcp tools was used for final verification.The mcp tools were integrated from the site: https://smithery.ai/

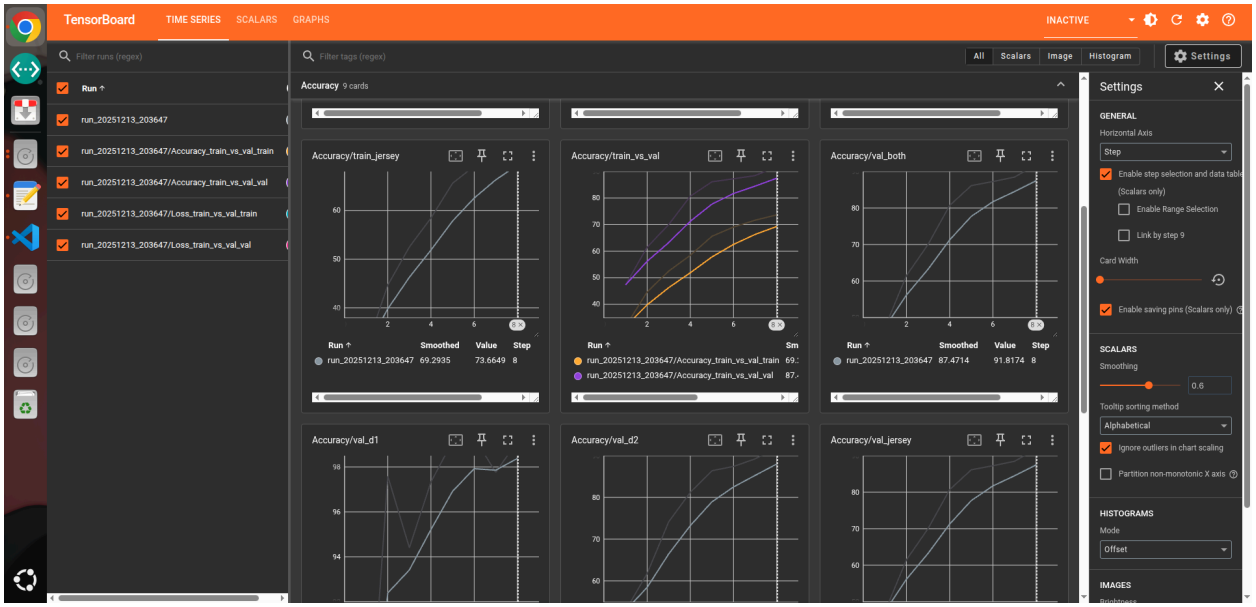
The mermaid diagrams(flowcharts) were made with the help of file indexing capabilities of glm 4.6 agent mode, cursor agent mode and deepseek. Then visualized via the site: https://mermaid.live/edit

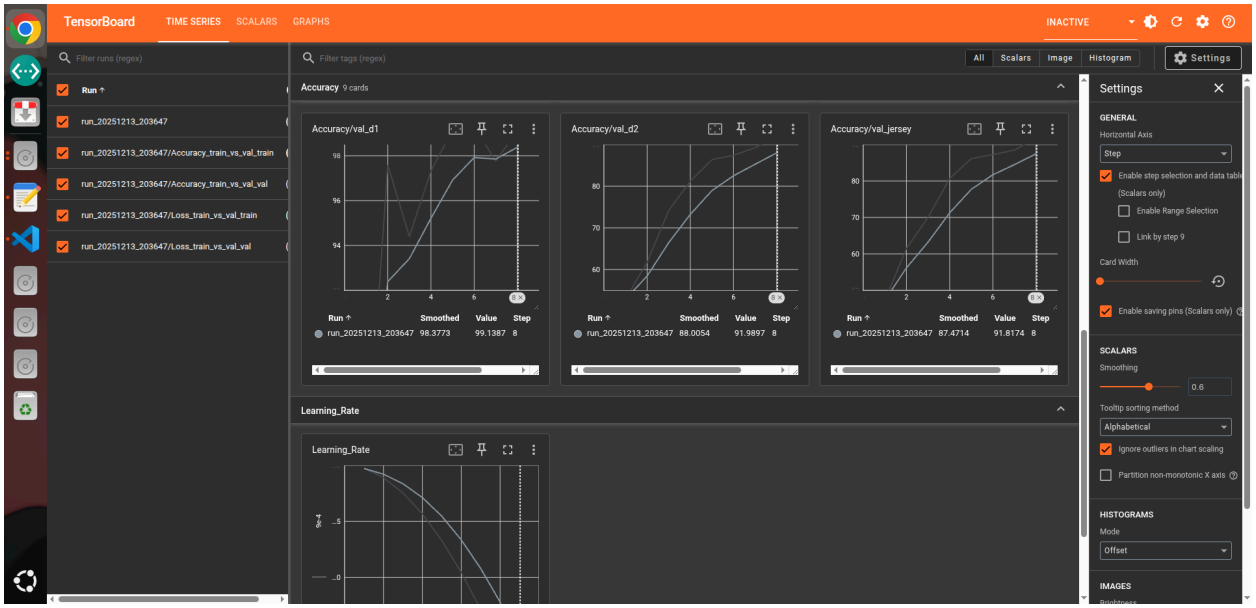
Model Architecture:

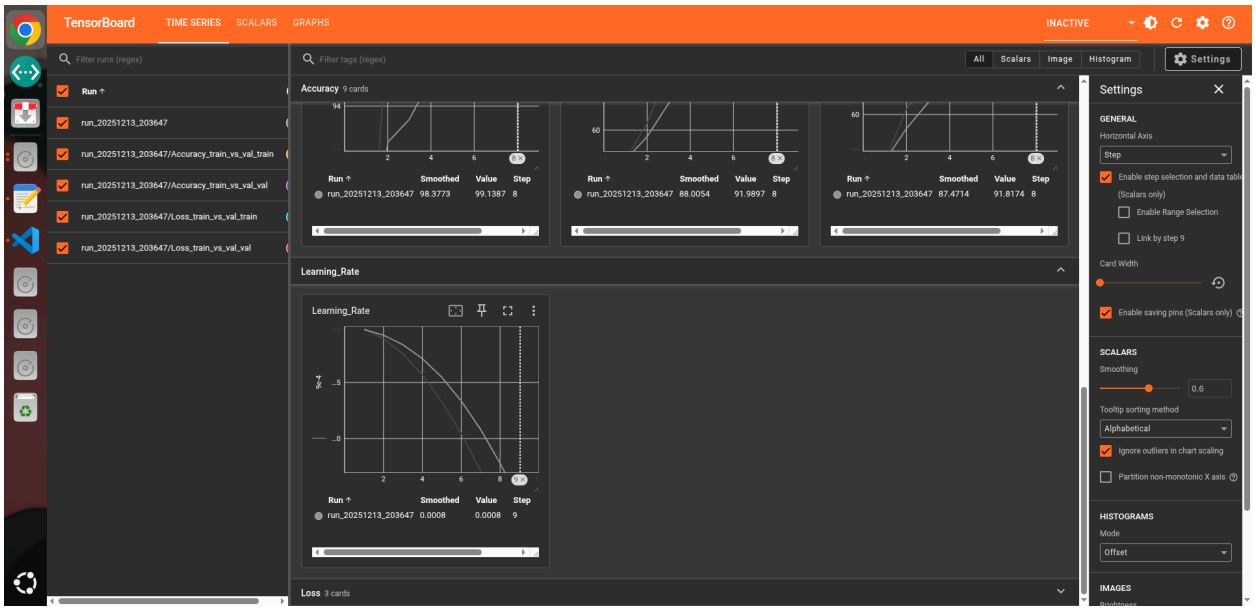

Input Sequence
Batch × 5 × 3 × 64 × 64
Spatial Feature Extraction
Frame 1
Frame 2
...
Frame 5
ResNet18 Backbone
ResNet18 Backbone
...
ResNet18 Backbone
Spatial Attention
Feature Maps
512 × H × W
Global Avg Pool
Feature Vector
512
Spatial Attention Map
H × W heatmap
Temporal Aggregation
Sequence of 5
512-dim vectors
Bidirectional LSTM
2 layers, 256 units
LSTM Output
5 × 512
Temporal Attention
Weighted Sum
512
Temporal Attention Weights
5 values
Classification Heads
512 features
Units Digit Head
FC 512→256 + BN + ReLU
Dropout
FC 256→256 + BN + ReLU
Residual Connection
Dropout
FC 256→11
Tens Digit Head
FC 512→256 + BN + ReLU
Dropout
FC 256→256 + BN + ReLU
Residual Connection
Dropout
FC 256→11
Units Digit Prediction
0-9
Tens Digit Prediction
0-9 or empty

Tensorboard Logs:

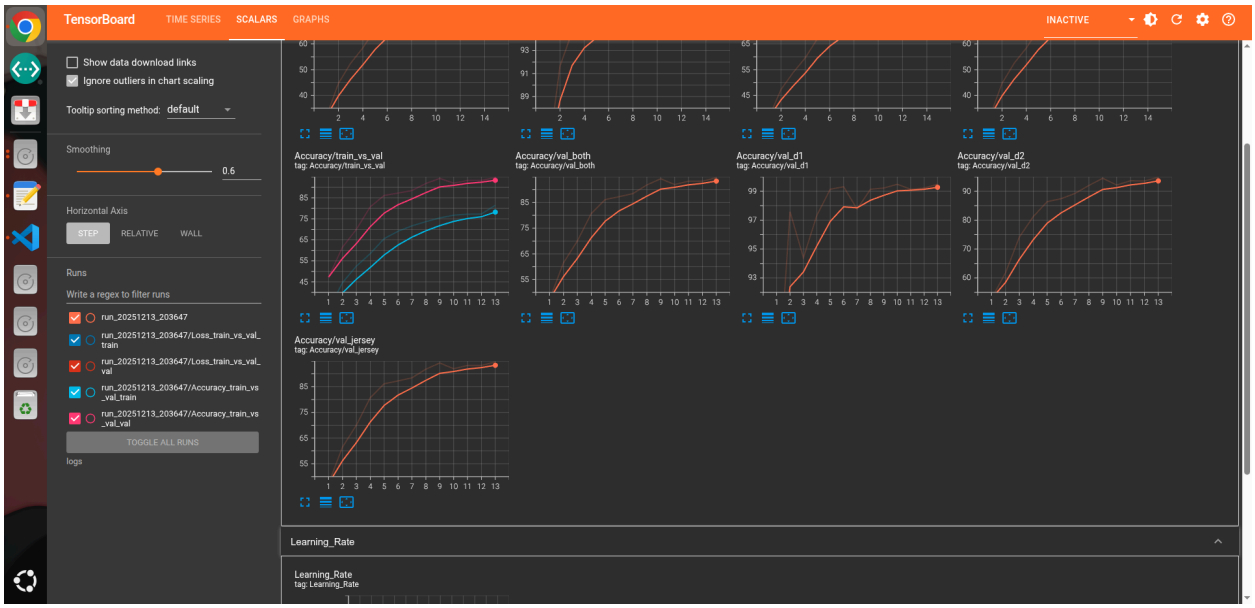


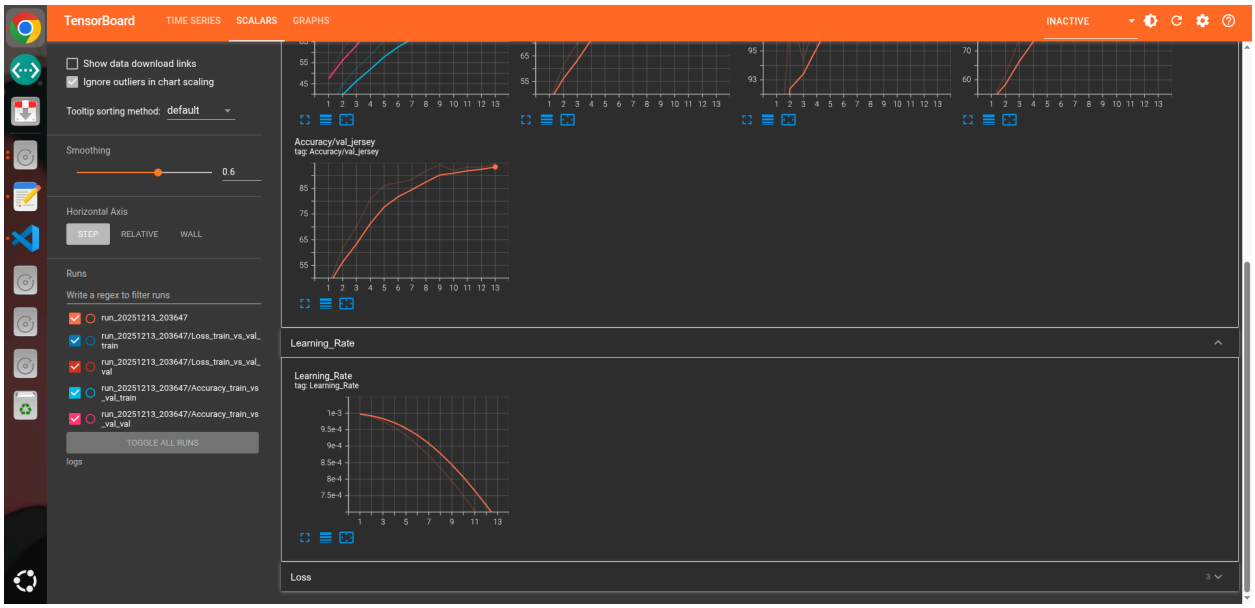


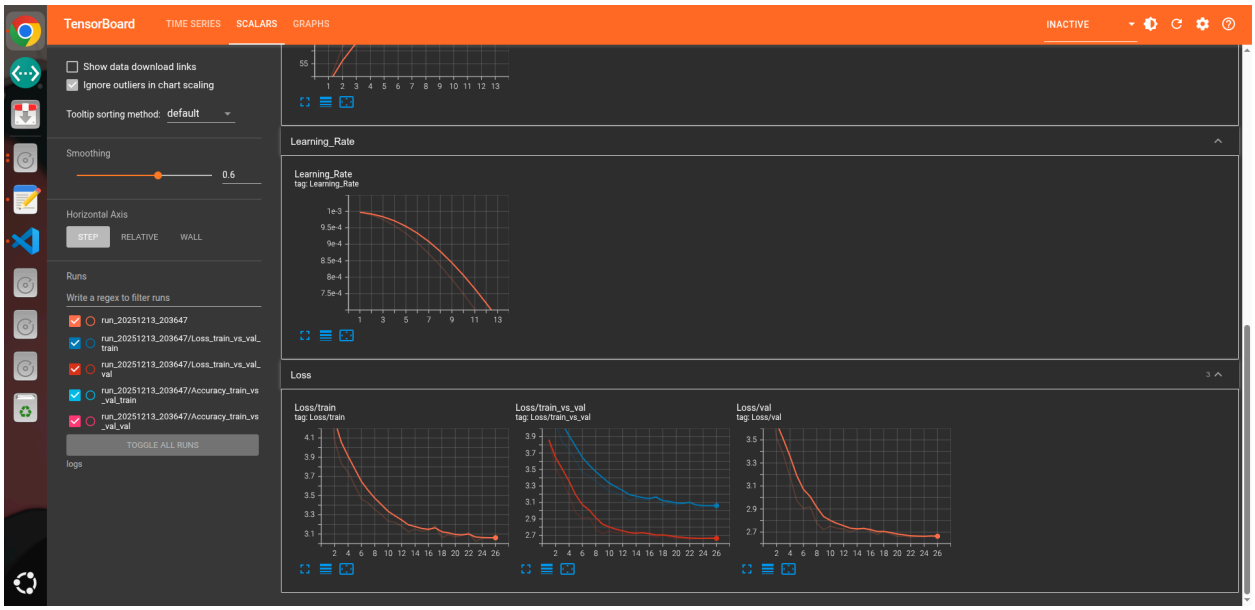

TensorBoard
TIME SERIES
SCALARS
GRAPHS
INACTIVE
Filter runs (regex)
Filter tags (regex)
All
Scalars
Image
Histogram
Settings
Run
run_20251213_203647
run_20251213_203647/Accuracy_train_vs_val_train
run_20251213_203647/Accuracy_train_vs_val_val
run_20251213_203647/Loss_train_vs_val_train
run_20251213_203647/Loss_train_vs_val_val
Accuracy 9 cards
Accuracy/val_d1
Accuracy/val_d2
Accuracy/val_jersey
Run
Smoothed
Value
Step
run_20251213_203647 98.3773 99.1387 8
run_20251213_203647 88.0054 91.9897 8
run_20251213_203647 87.4714 91.8174 8
Learning_Rate
Learning_Rate
Settings
GENERAL
Horizontal Axis
Step
Enable step selection and data table
(Scalars only)
Enable Range Selection
Link by step 9
Card Width
Enable saving pins (Scalars only)
SCALARS
Smoothing
0.6
Tooltip sorting method
Alphabetical
Ignore outliers in chart scaling
Partition non-monotonic X axis
HISTOGRAMS
Mode
Offset
IMAGES


TensorBoard
TIME SERIES
SCALARS
GRAPHS
INACTIVE
Filter runs (regex)
Filter tags (regex)
All
Scalars
Image
Histogram
Settings
Run
run_20251213_203647
run_20251213_203647/Accuracy_train_vs_val_train
run_20251213_203647/Accuracy_train_vs_val_val
run_20251213_203647/Loss_train_vs_val_train
run_20251213_203647/Loss_train_vs_val_val
Accuracy 9 cards
Run
Smoothed
Value
Step
run_20251213_203647 98.3773 99.1387 8
run_20251213_203647 88.0054 91.9897 8
run_20251213_203647 87.4714 91.8174 8
Learning_Rate
Learning_Rate
Run
Smoothed
Value
Step
run_20251213_203647 0.0008 0.0008 9
Loss 3 cards
Settings
GENERAL
Horizontal Axis
Step
Enable step selection and data table
(Scalars only)
Enable Range Selection
Link by step 9
Card Width
Enable saving pins (Scalars only)
SCALARS
Smoothing
0.6
Tooltip sorting method
Alphabetical
Ignore outliers in chart scaling
Partition non-monotonic X axis
HISTOGRAMS
Mode
Offset
IMAGES


TensorBoard
TIME SERIES
SCALARS
GRAPHS
INACTIVE
Show data download links
Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing
0.6
Horizontal Axis
STEP
RELATIVE
WALL
Runs
Write a regex to filter runs
run_20251213_203647
run_20251213_203647/Loss_train_vs_val_train
run_20251213_203647/Loss_train_vs_val_val
run_20251213_203647/Accuracy_train_vs_val_train
run_20251213_203647/Accuracy_train_vs_val_val
TOGGLE ALL RUNS
logs
Filter tags (regular expressions supported)
Accuracy
Accuracy/train_both
tag: Accuracy/train_both
Accuracy/train_d1
tag: Accuracy/train_d1
Accuracy/train_d2
tag: Accuracy/train_d2
Accuracy/train_jersey
tag: Accuracy/train_jersey
Accuracy/train_vs_val
tag: Accuracy/train_vs_val
Accuracy/val_both
tag: Accuracy/val_both
Accuracy/val_d1
tag: Accuracy/val_d1
Accuracy/val_d2
tag: Accuracy/val_d2
Accuracy/val_jersey
tag: Accuracy/val_jersey

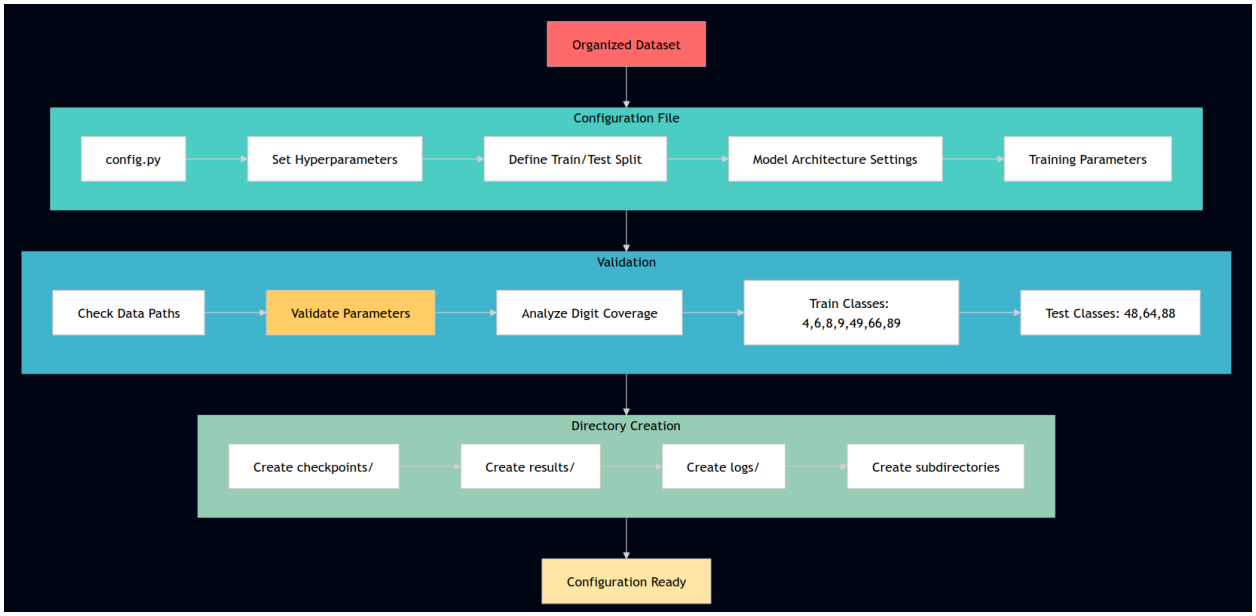

TensorBoard
TIME SERIES
SCALARS
GRAPHS
INACTIVE
Show data download links
Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing
0.6
Horizontal Axis
STEP
RELATIVE
WALL
Runs
Write a regex to filter runs
run_20251213_203647
run_20251213_203647/Loss_train_vs_val_train
run_20251213_203647/Loss_train_vs_val_val
run_20251213_203647/Accuracy_train_vs_val_train
run_20251213_203647/Accuracy_train_vs_val_val
TOGGLE ALL RUNS
logs
Accuracy/train_vs_val
tag: Accuracy/train_vs_val
Accuracy/val_both
tag: Accuracy/val_both
Accuracy/val_d1
tag: Accuracy/val_d1
Accuracy/val_d2
tag: Accuracy/val_d2
Accuracy/val_jersey
tag: Accuracy/val_jersey
Learning_Rate
Learning_Rate
tag: Learning_Rate

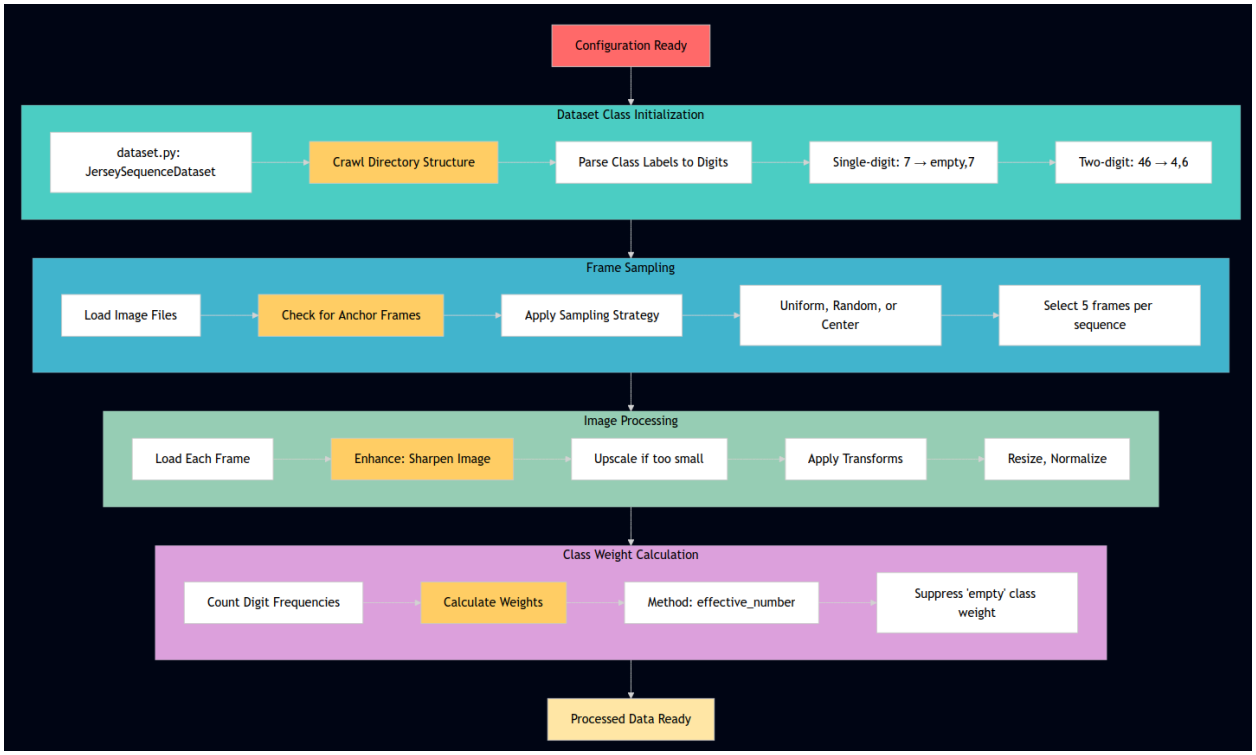


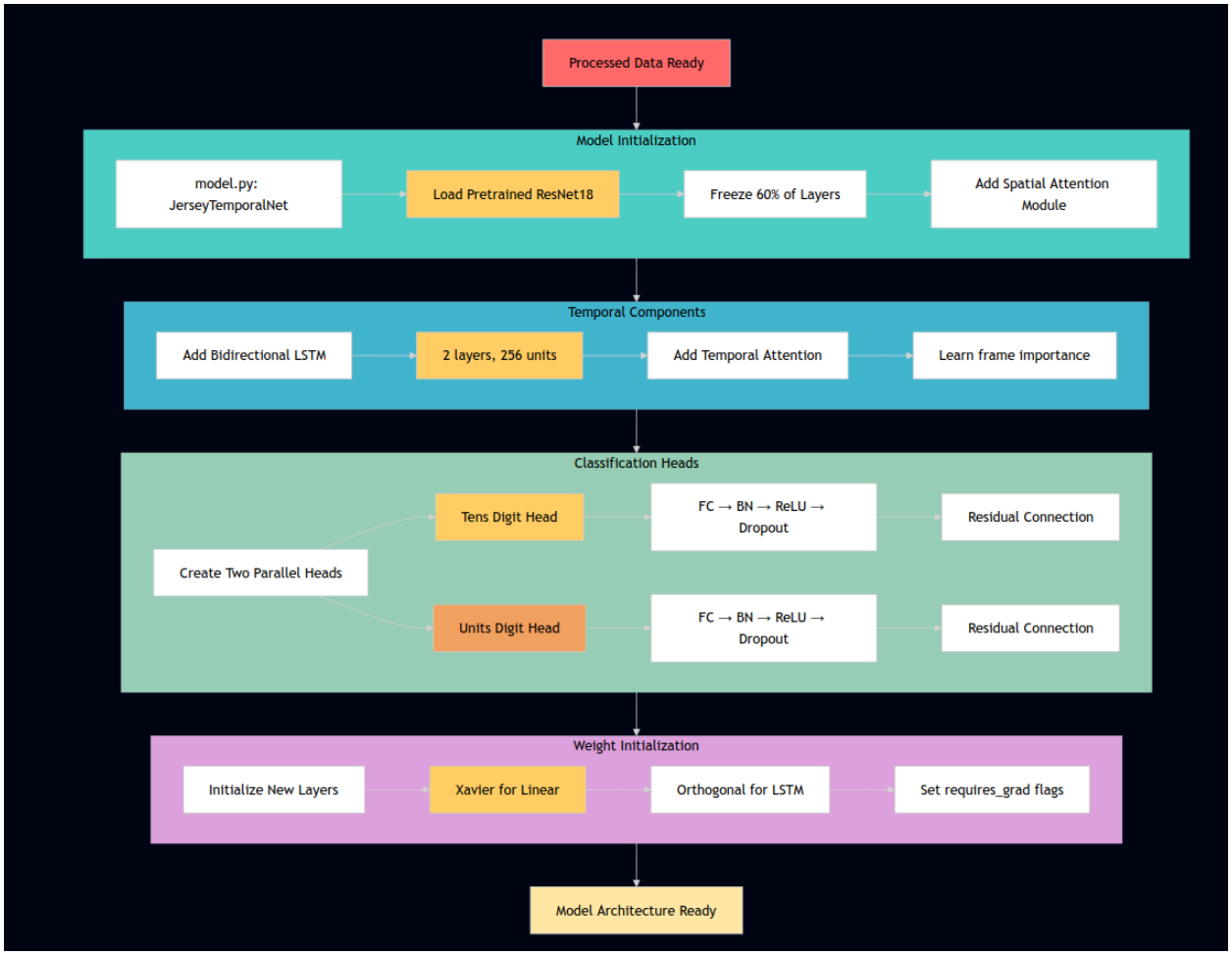


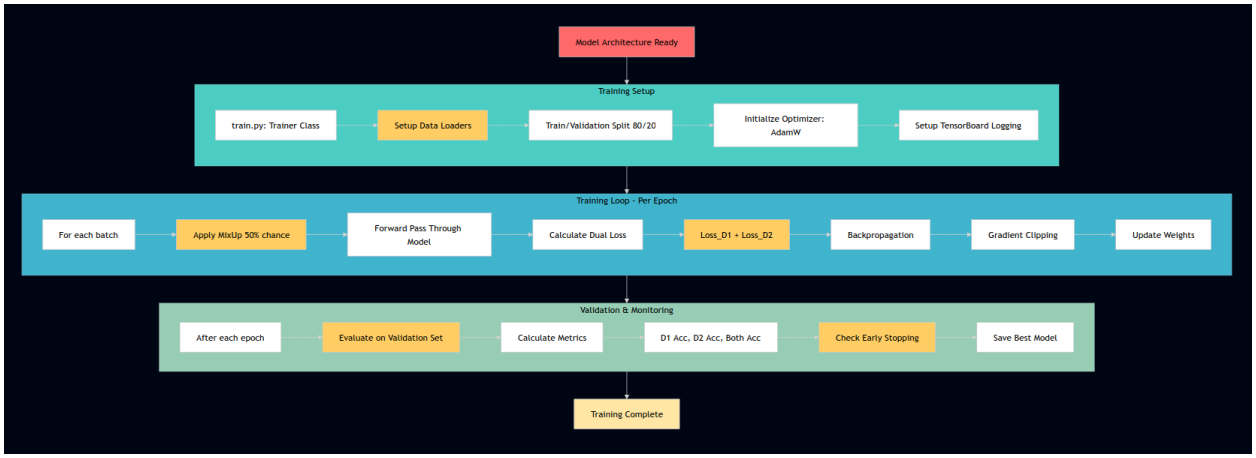
Process Flow

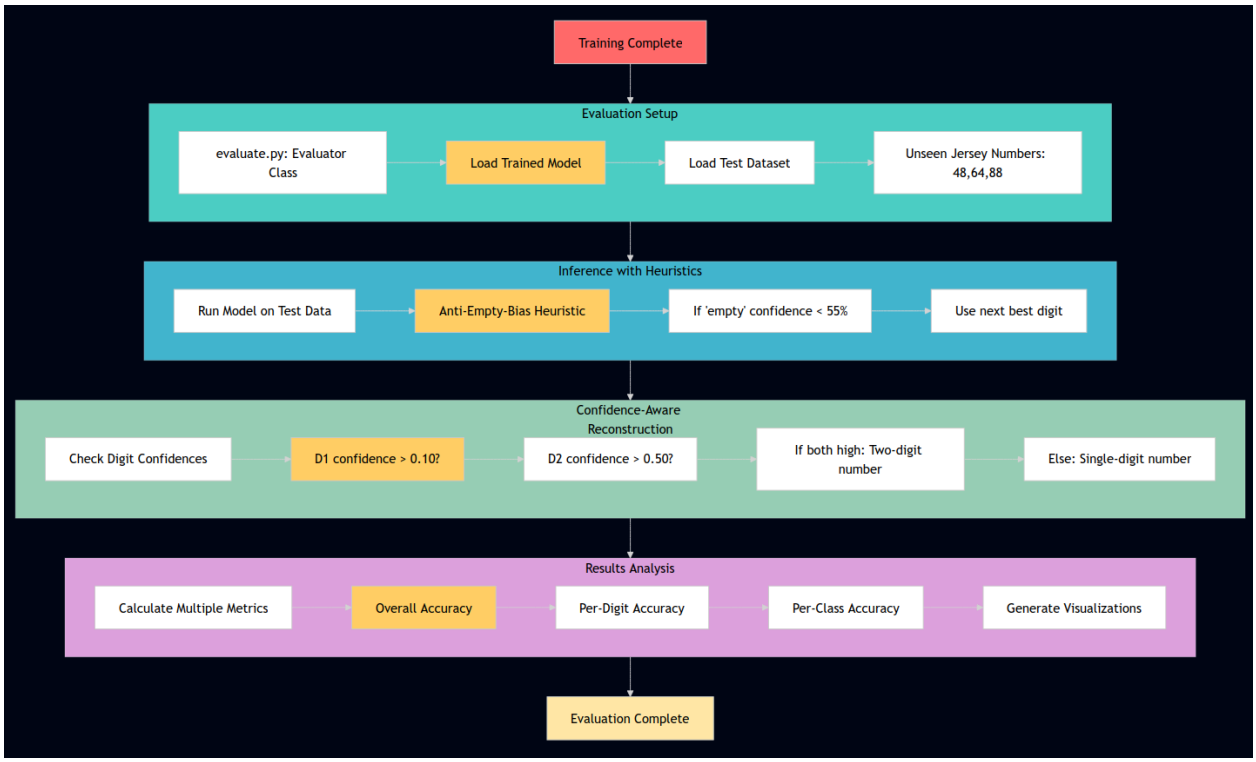
Phase 1: Configuration and Setup
Phase 2: Data Loading and Processing
Phase 3: Model Architecture Creation
Phase 4: Training Process
Phase 5: Evaluation and Testing
Phase 6: Model Analysis and Reporting

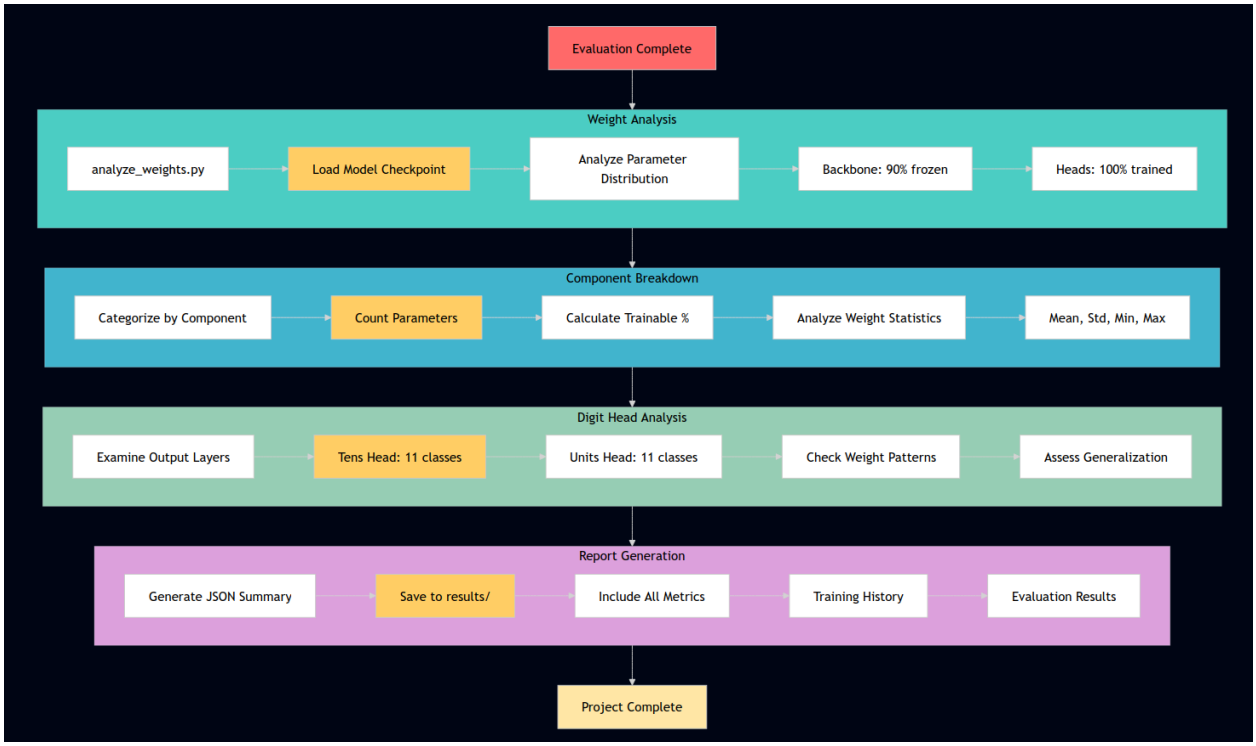

Organized Dataset
Configuration File
config.py
Set Hyperparameters
Define Train/Test Split
Model Architecture Settings
Training Parameters
Validation
Check Data Paths
Validate Parameters
Analyze Digit Coverage
Train Classes: 4,6,8,9,49,66,89
Test Classes: 48,64,88
Directory Creation
Create checkpoints/
Create results/
Create logs/
Create subdirectories
Configuration Ready


Configuration Ready
Dataset Class Initialization
dataset.py: JerseySequenceDataset
Crawl Directory Structure
Parse Class Labels to Digits
Single-digit: 7 → empty,7
Two-digit: 46 → 4,6
Frame Sampling
Load Image Files
Check for Anchor Frames
Apply Sampling Strategy
Uniform, Random, or Center
Select 5 frames per sequence
Image Processing
Load Each Frame
Enhance: Sharpen Image
Upscale if too small
Apply Transforms
Resize, Normalize
Class Weight Calculation
Count Digit Frequencies
Calculate Weights
Method: effective_number
Suppress 'empty' class weight
Processed Data Ready


Processed Data Ready
Model Initialization
model.py: JerseyTemporalNet
Load Pretrained ResNet18
Freeze 60% of Layers
Add Spatial Attention Module
Temporal Components
Add Bidirectional LSTM
2 layers, 256 units
Add Temporal Attention
Learn frame importance
Classification Heads
Create Two Parallel Heads
Tens Digit Head
FC → BN → ReLU → Dropout
Residual Connection
Units Digit Head
FC → BN → ReLU → Dropout
Residual Connection
Weight Initialization
Initialize New Layers
Xavier for Linear
Orthogonal for LSTM
Set requires_grad flags
Model Architecture Ready


Model Architecture Ready
Training Setup
train.py: Trainer Class
Setup Data Loaders
Train/Validation Split 80/20
Initialize Optimizer: AdamW
Setup TensorBoard Logging
Training Loop - Per Epoch
For each batch
Apply MixUp 50% chance
Forward Pass Through Model
Calculate Dual Loss
Loss_D1 + Loss_D2
Backpropagation
Gradient Clipping
Update Weights
Validation & Monitoring
After each epoch
Evaluate on Validation Set
Calculate Metrics
D1 Acc, D2 Acc, Both Acc
Check Early Stopping
Save Best Model
Training Complete


Training Complete
Evaluation Setup
evaluate.py: Evaluator Class
Load Trained Model
Load Test Dataset
Unseen Jersey Numbers: 48,64,88
Inference with Heuristics
Run Model on Test Data
Anti-Empty-Bias Heuristic
If 'empty' confidence < 55%
Use next best digit
Confidence-Aware Reconstruction
Check Digit Confidences
D1 confidence > 0.10?
D2 confidence > 0.50?
If both high: Two-digit number
Else: Single-digit number
Results Analysis
Calculate Multiple Metrics
Overall Accuracy
Per-Digit Accuracy
Per-Class Accuracy
Generate Visualizations
Evaluation Complete



Now codes for individuals files will be shared below:

dataset.py:

```python
# dataset.py

"""
Custom Dataset Loader for Jersey Number Recognition
Handles hierarchical directory structure:
Root/{Class}/{Jersey_Seq}/{Subdir}/.jpg
"""

import os
import glob
import random
import torch
from torch.utils.data import Dataset
from PIL import Image, ImageEnhance
import numpy as np
from config import Config
```

```
class JerseySequenceDataset(Dataset):
   def __init__(self, root_dir, transform=None, seq_length=5,
                allowed_classes=None, mode='train',
sampling_strategy='uniform'):
       """
       Args:
           root_dir (str): Root directory of dataset
           transform: torchvision transforms to apply
           seq_length (int): Number of frames to sample per sequence
           allowed_classes (list): List of class numbers to include (e.g.,
[4, 6, 46])
                                   If None, uses all available classes
           mode (str): 'train', 'val', or 'test' - for logging purposes
           sampling_strategy (str): 'uniform', 'random', or 'center'
       """
       self.root_dir = root_dir
       self.transform = transform
       self.seq_length = seq_length
       self.allowed_classes = allowed_classes
       self.mode = mode
       self.sampling_strategy = sampling_strategy
       self.samples = []  # List of (sequence_path, label_d1, label_d2,
class_name)

       self._crawl_dataset()

   def _parse_class_label(self, class_name):
       """
       Parse class folder name to digit labels.

       Args:
           class_name: Folder name

       Returns:
           (label_d1, label_d2): Tuple of digit labels
               - label_d1: Tens digit (0-9) or 10 for empty
               - label_d2: Units digit (0-9)
       """
       try:
           class_num = int(class_name)
```

```
            if class_num < 0 or class_num >= 100:
                raise ValueError(f"Class number out of range [0, 99]:
{class_num}")

            if class_num < 10:
                # Single digit: 0-9
                label_d1 = 10  # Empty tens place
                label_d2 = class_num
            else:
                # Two digits: 10-99
                label_d1 = class_num // 10
                label_d2 = class_num % 10

            return label_d1, label_d2

        except ValueError as e:
            print(f" Warning: Invalid class folder name '{class_name}':
{e}")
            return None, None

    def _crawl_dataset(self):
        """
        Crawls the directory structure:
        Root/{Class}/{Jersey_Seq_ID}/{Label_Subdir}/.jpg
        """
        if not os.path.exists(self.root_dir):
            raise ValueError(f"Dataset path does not exist:
{self.root_dir}")

        # Get all class folders
        class_folders = [d for d in os.listdir(self.root_dir)
                         if os.path.isdir(os.path.join(self.root_dir, d))]

        # Filter by allowed classes if specified
        if self.allowed_classes is not None:
            class_folders = [c for c in class_folders
                             if c.isdigit() and int(c) in
self.allowed_classes]
```

```
        class_folders = sorted(class_folders, key=lambda x: int(x) if
x.isdigit() else 999)

        total_sequences = 0
        class_distribution = {}

        for class_name in class_folders:
            class_path = os.path.join(self.root_dir, class_name)

            # Parse labels using improved function
            label_d1, label_d2 = self._parse_class_label(class_name)

            # Skip invalid class names
            if label_d1 is None or label_d2 is None:
                continue

            # Iterate over Jersey/Sequence folders
            seq_folders = [f for f in os.listdir(class_path)
                           if os.path.isdir(os.path.join(class_path, f))]

            class_seq_count = 0

            for seq_id in seq_folders:
                seq_path = os.path.join(class_path, seq_id)

                # Iterate over internal label subdirectories
                # These are sub-sequences
                sub_dirs = [s for s in os.listdir(seq_path)
                            if os.path.isdir(os.path.join(seq_path, s))]

                for sub in sub_dirs:
                    final_path = os.path.join(seq_path, sub)

                    # Check if it contains images
                    images = glob.glob(os.path.join(final_path, ".jpg")) +
\
                             glob.glob(os.path.join(final_path, ".png"))

                    if len(images) >= 1:  # At least 1 frame
```

```
                        self.samples.append((final_path, label_d1,
label_d2, class_name))
                        class_seq_count += 1

            if class_seq_count > 0:
                class_distribution[class_name] = class_seq_count
                total_sequences += class_seq_count

        # Print dataset statistics
        print(f"\n{'='60}")
        print(f"📊 {self.mode.upper()} Dataset Loaded")
        print(f"{'='60}")
        print(f"Total sequences: {total_sequences}")
        print(f"Classes found: {len(class_distribution)}")
        print("\nClass distribution:")
        for cls, count in sorted(class_distribution.items(), key=lambda x:
int(x[0])):
            print(f"  Class {cls:>2s}: {count:>4d} sequences")
        print(f"{'='60}\n")

    def _sample_frames(self, image_files):
        """
        Sample frames using different strategies, prioritizing anchor
frames.

        Args:
            image_files: List of image file paths

        Returns:
            indices: List of frame indices to use
        """
        num_frames = len(image_files)
        indices = []

        # Find anchor frames
        anchor_indices = [i for i, f in enumerate(image_files) if '_anchor'
in f]

        # Always include at least one anchor frame if available
        if anchor_indices:
```

```
            # Use the first anchor as frame 0
            indices.append(anchor_indices[0])
            # Remove it from candidate pool to avoid duplication
            remaining_indices = [i for i in range(num_frames) if i not in
anchor_indices]
        else:
            # No anchors — sample from all frames
            remaining_indices = list(range(num_frames))

        # Sample remaining frames based on strategy
        remaining_needed = self.seq_length - len(indices)

        if remaining_needed > 0:
            if self.sampling_strategy == 'uniform':
                if len(remaining_indices) >= remaining_needed:
                    step = len(remaining_indices) / remaining_needed
                    sampled = [int(i  step) for i in
range(remaining_needed)]
                    indices.extend([remaining_indices[i] for i in sampled])
                else:
                    indices.extend(remaining_indices)
                    while len(indices) < self.seq_length:
                        indices.append(remaining_indices[-1])

            elif self.sampling_strategy == 'random':
                if len(remaining_indices) >= remaining_needed:
                    sampled = np.random.choice(remaining_indices,
remaining_needed, replace=False)
                    indices.extend(sampled.tolist())
                else:
                    indices.extend(remaining_indices)
                    while len(indices) < self.seq_length:
                        indices.append(np.random.choice(remaining_indices))

            elif self.sampling_strategy == 'center':
                center = len(remaining_indices) // 2
                half_seq = remaining_needed // 2
                start = max(0, center - half_seq)
                end = min(len(remaining_indices), start + remaining_needed)
                sampled = remaining_indices[start:end]
```

```
                indices.extend(sampled)
                while len(indices) < self.seq_length:
                    indices.append(sampled[-1])

            else:
                raise ValueError(f"Unknown sampling strategy:
{self.sampling_strategy}")

        # Ensure we have exactly seq_length frames
        indices = indices[:self.seq_length]
        while len(indices) < self.seq_length:
            indices.append(indices[-1])

        return indices

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

    def _enhance_image(self, img):
        """
        Enhance image quality: sharpen and optionally upscale.

        Args:
            img: PIL.Image

        Returns:
            Enhanced PIL.Image
        """
        # Option 1: Sharpen using PIL (lightweight, no extra deps)
        enhancer = ImageEnhance.Sharpness(img)
        img = enhancer.enhance(2.0)  # Increase sharpness by 2x

        # Optional: Upscale if image is very small (e.g., < 64px)
        min_size = 64
        if img.width < min_size or img.height < min_size:
            # Use LANCZOS for best quality upsampling
            scale_factor = max(min_size / img.width, min_size / img.height)
            new_size = (int(img.width  scale_factor), int(img.height
scale_factor))
            img = img.resize(new_size, Image.LANCZOS)
```

```
        return img

    def __getitem__(self, idx):
        seq_path, d1, d2, class_name = self.samples[idx]

        # Get all image files
        image_files = sorted(
            glob.glob(os.path.join(seq_path, ".jpg")) +
            glob.glob(os.path.join(seq_path, ".png"))
        )

        # Sample frames using chosen strategy
        indices = self._sample_frames(image_files)

        # Load and Transform Images
        frames = []
        for i in indices:
            img_path = image_files[i]
            try:
                img = Image.open(img_path).convert('RGB')

                # ENHANCE IMAGE BEFORE TRANSFORM
                img = self._enhance_image(img)

                if self.transform:
                    img = self.transform(img)
                else:
                    # Always resize + normalize for consistency
                    from torchvision import transforms
                    default_transform = transforms.Compose([
                        transforms.Resize((Config.IMG_SIZE,
Config.IMG_SIZE)),
                        transforms.ToTensor(),
                        transforms.Normalize(mean=[0.485, 0.456, 0.406],
std=[0.229, 0.224, 0.225])
                    ])
                    img = default_transform(img)
                frames.append(img)
            except Exception as e:
```

```
                print(f" Warning: Failed to load {img_path}: {e}")
                if self.transform:
                    black_frame = torch.zeros(3, Config.IMG_SIZE,
Config.IMG_SIZE)
                else:
                    black_frame = torch.zeros(3, 224, 224)
                frames.append(black_frame)

        # Stack into tensor: [Seq_Len, Channels, Height, Width]
        seq_tensor = torch.stack(frames)

        return seq_tensor, torch.tensor(d1, dtype=torch.long),
torch.tensor(d2, dtype=torch.long)

    def get_class_weights(self, method='balanced',
suppress_empty_weight=True):
        """
        Calculate class weights for handling imbalanced data.

        Args:
            method: 'balanced' or 'effective_number'
            suppress_empty_weight: If True, heavily suppress the weight for
D1=10 (empty)

        Returns:
            (weights_d1, weights_d2): Tuple of weight tensors
        """
        from collections import Counter

        d1_labels = [s[1] for s in self.samples]
        d2_labels = [s[2] for s in self.samples]

        d1_counts = Counter(d1_labels)
        d2_counts = Counter(d2_labels)

        print("\n Label Distribution:")
        print(f"D1 counts: {dict(sorted(d1_counts.items()))}")
        print(f"D2 counts: {dict(sorted(d2_counts.items()))}")

        # Initialize weights
```

```
        d1_weights = torch.ones(Config.NUM_DIGIT_CLASSES)
        d2_weights = torch.ones(Config.NUM_DIGIT_CLASSES)

        total_samples = len(self.samples)

        if method == 'balanced':
            # Inverse frequency weighting
            for digit in range(Config.NUM_DIGIT_CLASSES):
                if digit in d1_counts and d1_counts[digit] > 0:
                    d1_weights[digit] = total_samples / (d1_counts[digit]
Config.NUM_DIGIT_CLASSES)
                else:
                    # Missing class: give very low weight
                    d1_weights[digit] = 0.001

            for digit in range(Config.NUM_DIGIT_CLASSES):
                if digit in d2_counts and d2_counts[digit] > 0:
                    d2_weights[digit] = total_samples / (d2_counts[digit]
Config.NUM_DIGIT_CLASSES)
                else:
                    d2_weights[digit] = 0.001

        elif method == 'effective_number':
            # Effective number of samples
            beta = 0.9999

            for digit in range(Config.NUM_DIGIT_CLASSES):
                if digit in d1_counts and d1_counts[digit] > 0:
                    effective_num = (1.0 - beta  d1_counts[digit]) / (1.0 -
beta)
                    d1_weights[digit] = 1.0 / effective_num
                else:
                    d1_weights[digit] = 0.001

            for digit in range(Config.NUM_DIGIT_CLASSES):
                if digit in d2_counts and d2_counts[digit] > 0:
                    effective_num = (1.0 - beta  d2_counts[digit]) / (1.0 -
beta)
                    d2_weights[digit] = 1.0 / effective_num
                else:
```

```
                    d2_weights[digit] = 0.001

            # Manually suppress the "empty" class (10) for D1
            if suppress_empty_weight and 10 in d1_counts:
                print(f"\n🔧 Suppressing D1 'empty' class weight by 100x")
                d1_weights[10] = d1_weights[10]  0.01

        else:
            raise ValueError(f"Unknown class weight method: {method}")

        d1_weights = torch.sqrt(d1_weights)
        d2_weights = torch.sqrt(d2_weights)

        # Print weight distribution with better formatting
        print(f"\n  Class Weights (method={method}):")
        print("\nD1 weights (Tens digit):")
        for i in range(Config.NUM_DIGIT_CLASSES):
            label = "Empty" if i == 10 else str(i)
            count = d1_counts.get(i, 0)
            print(f"  {label:>5s} (count: {count:>6d}): weight =
{d1_weights[i]:.6f}")

        print("\nD2 weights (Units digit):")
        for i in range(Config.NUM_DIGIT_CLASSES):
            label = "Empty" if i == 10 else str(i)
            count = d2_counts.get(i, 0)
            if count > 0:
                print(f"  {label:>5s} (count: {count:>6d}): weight =
{d2_weights[i]:.6f}")

        return d1_weights, d2_weights

    def set_transform(self, transform):
        """Change the transform"""
        self.transform = transform


# Test the dataset
if __name__ == "__main__":
    from torchvision import transforms
```

```
from config import Config, validate_config

validate_config()

# Define transforms
transform = transforms.Compose([
    transforms.Resize((Config.IMG_SIZE, Config.IMG_SIZE)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])

# Create dataset
dataset = JerseySequenceDataset(
    root_dir=Config.DATA_ROOT,
    transform=transform,
    seq_length=Config.SEQ_LENGTH,
    allowed_classes=Config.TRAIN_CLASSES,
    mode='train',
    sampling_strategy='uniform'
)

print("\n Dataset created successfully!")
print(f"Total samples: {len(dataset)}")

# Test loading one sample
if len(dataset) > 0:
    seq, d1, d2 = dataset[0]
    print(f"\nSample shape: {seq.shape}")
    print(f"Label D1 (tens): {d1.item()}")
    print(f"Label D2 (units): {d2.item()}")

    # Test class weights
    w1, w2 = dataset.get_class_weights(method='balanced')
    print("\n Class weights calculated!")
```

This is a custom PyTorch Dataset class for loading jersey number recognition data from video sequences.

1. WHAT THE DATASET DOES:

Loads sequences of images showing jersey numbers (like sports jerseys with numbers 0-99)

Each sequence shows the same jersey number from multiple frames

The goal is to train a model to recognize two-digit numbers

2. DIRECTORY STRUCTURE IT EXPECTS:
Root/Class/Sequence_ID/Subfolder/.jpg
Example:
Data/4/Seq_001/labeled_001/frame1.jpg
Data/4/Seq_001/labeled_001/frame2.jpg
...
Data/46/Seq_003/labeled_002/frame1.jpg

3. KEY FEATURES OF THE DATASET CLASS:

Label Parsing:

Jersey numbers are parsed into two digits

Single-digit numbers (0-9): first digit is "empty" (label 10), second digit is the number

Two-digit numbers (10-99): first digit is tens place, second digit is units place

Example: "4" becomes (10, 4), "46" becomes (4, 6)

Frame Sampling Strategies:

'uniform': evenly spaced frames from the sequence

'random': random frames

'center': frames from middle of sequence

Prioritizes "anchor" frames (frames with '_anchor' in filename) as first frame

Image Enhancement:

Sharpens images to improve clarity

Optionally upscales small images (<64px) for better quality

Applies standard transforms (resize, normalize)

Class Weight Calculation:

Calculates weights for imbalanced datasets

Two methods: 'balanced' (inverse frequency) and 'effective_number'

Can suppress weight for "empty" tens digit to avoid bias

4. WORKFLOW:

Initialization (_crawl_dataset):

Scans directory structure

Counts sequences per class

Filters by allowed classes if specified

Prints statistics about dataset

Getting an item (getitem):

Takes a sequence path

Samples frames using chosen strategy

Loads and enhances each image

Applies transforms

Returns: tensor of shape [Seq_Length, Channels, Height, Width], first digit label, second digit label

5. OUTPUT FORMAT:
For each sample returns:

seq_tensor: [5, 3, 224, 224] (if seq_length=5, image size=224)

d1_label: tensor with first digit (0-9 or 10 for empty)

d2_label: tensor with second digit (0-9)

6. USE CASE:
This dataset is designed for training models that:

Take multiple frames of a jersey number as input

Predict two separate digits (tens and units place)

Handle single-digit numbers (like "7") and two-digit numbers (like "23")

Work with potentially imbalanced class distributions

7. TESTING:
The bottom section tests the dataset by:

Creating a transform (resize, normalize)

Instantiating the dataset

Loading one sample

Calculating class weights

The main strength is handling sequential data with hierarchical directory structures and providing multiple sampling strategies for frames.

config.py:

```
# config.py

"""
Configuration file for Jersey Number Recognition
Optimized for your actual dataset
"""

import os
import warnings
from dotenv import load_dotenv


load_dotenv()  # Load environment variables from .env file if present

class Config:
    # ==========================================
```

```python
# DATA CONFIGURATION
# ============================================
# Use environment variable
DATA_ROOT = os.environ.get('DATA_ROOT')

# ============================================
#  TRAINING/TEST SPLIT
# ============================================

TRAIN_CLASSES = [4, 6, 8, 9, 49, 66, 89]
TEST_CLASSES = [48, 64, 88]

# Validation split (from training data)
VAL_SPLIT = 0.2

# ============================================
# MODEL CONFIGURATION
# ============================================
NUM_DIGIT_CLASSES = 11  # 0-9 + empty = 11
HIDDEN_DIM = 256
IMG_SIZE = 64
SEQ_LENGTH = 5

# Model architecture options
BACKBONE = 'resnet18'
USE_SPATIAL_ATTENTION = True
USE_TEMPORAL_ATTENTION = True
BIDIRECTIONAL_LSTM = True

# ============================================
# TRAINING CONFIGURATION
# ============================================
BATCH_SIZE = 16
NUM_EPOCHS = 30
LEARNING_RATE = 0.001
WEIGHT_DECAY = 1e-4
DROPOUT = 0.3

# Gradient clipping
GRADIENT_CLIP_VALUE = 1.0
```

```
    # Learning rate scheduler
    USE_SCHEDULER = True
    SCHEDULER_STEP_SIZE = 10
    SCHEDULER_GAMMA = 0.5

    # Early stopping
    EARLY_STOPPING_PATIENCE = 10
    EARLY_STOPPING_MIN_DELTA = 0.001

    # Mixed precision training
    USE_MIXED_PRECISION = True

    # Class weight method
    CLASS_WEIGHT_METHOD = 'effective_number'  # 'balanced' or
'effective_number'

    # ============================================
    # DATA AUGMENTATION
    # ============================================
    # Safe augmentations that don't change digit identity
    AUG_ROTATION_DEGREES = 5
    AUG_TRANSLATE = (0.05, 0.05)
    AUG_SCALE = (0.95, 1.05)
    AUG_BRIGHTNESS = 0.2
    AUG_CONTRAST = 0.2

    # Temporal sampling strategy
    TEMPORAL_SAMPLING = 'uniform'  # 'uniform', 'random', 'center'

    # ============================================
    # SYSTEM CONFIGURATION
    # ============================================
    NUM_WORKERS = 4
    DEVICE = 'cuda'
    SEED = 42
    PIN_MEMORY = True


    # Confidence thresholds for zero-shot jersey number reconstruction
```

```
    D1_CONF_THRESHOLD = 0.10   # Tens digit confidence threshold
    D2_CONF_THRESHOLD = 0.5    # Units digit confidence threshold

    # ============================================
    # OUTPUT PATHS
    # ============================================
    CHECKPOINT_DIR = 'checkpoints'
    RESULTS_DIR = 'results'
    LOGS_DIR = 'logs'

    @staticmethod
    def create_dirs():
        """Create all necessary output directories"""
        os.makedirs(Config.CHECKPOINT_DIR, exist_ok=True)
        os.makedirs(Config.RESULTS_DIR, exist_ok=True)
        os.makedirs(Config.LOGS_DIR, exist_ok=True)
        os.makedirs(os.path.join(Config.RESULTS_DIR, 'confusion_matrices'),
exist_ok=True)
        os.makedirs(os.path.join(Config.RESULTS_DIR, 'predictions'),
exist_ok=True)
        os.makedirs(os.path.join(Config.RESULTS_DIR, 'visualizations'),
exist_ok=True)

    @staticmethod
    def validate():
        """Validate configuration parameters"""
        errors = []
        warnings_list = []

        # Check data root exists
        if not os.path.exists(Config.DATA_ROOT):
            errors.append(f"Data root does not exist: {Config.DATA_ROOT}")

        # Check train/test split
        if Config.TRAIN_CLASSES is not None and Config.TEST_CLASSES is not
None:
            overlap = set(Config.TRAIN_CLASSES) & set(Config.TEST_CLASSES)
            if overlap:
                errors.append(f"Train and test classes overlap: {overlap}")
```

```
        # Validate hyperparameters
        if Config.BATCH_SIZE < 1:
            errors.append("Batch size must be >= 1")

        if not (32 <= Config.IMG_SIZE <= 512):
            warnings_list.append(f"IMG_SIZE={Config.IMG_SIZE} is unusual
(recommended: 128-256)")

        if Config.SEQ_LENGTH < 1:
            errors.append("SEQ_LENGTH must be >= 1")

        if not (0.0 < Config.VAL_SPLIT < 1.0):
            errors.append("VAL_SPLIT must be between 0 and 1")

        if not (0.0 <= Config.DROPOUT < 1.0):
            errors.append("DROPOUT must be between 0 and 1")

        # Check GPU availability
        if Config.DEVICE == 'cuda':
            try:
                import torch
                if not torch.cuda.is_available():
                    warnings_list.append("CUDA requested but not available,
will fall back to CPU")
                    Config.DEVICE = 'cpu'
            except ImportError:
                errors.append("PyTorch not installed")

        # Print errors
        if errors:
            print("\n CONFIGURATION ERRORS:")
            for error in errors:
                print(f"  • {error}")
            raise ValueError("Configuration validation failed")

        # Print warnings
        if warnings_list:
            print("\n CONFIGURATION WARNINGS:")
            for warning in warnings_list:
                print(f"  • {warning}")
```

```
        return True


def validate_config():
    """Check if configuration is valid and analyze digit coverage"""
    print("\n" + "="70)
    print("VALIDATING CONFIGURATION")
    print("="70)

    # Validate parameters
    Config.validate()

    # Validate digit coverage
    print("\n🔍 Analyzing Digit Coverage:")

    if Config.TRAIN_CLASSES is None:
        print(" No specific training classes defined")
        return

    train_tens = set()
    train_units = set()

    for cls in Config.TRAIN_CLASSES:
        if cls < 10:
            train_units.add(cls)
        else:
            train_tens.add(cls // 10)
            train_units.add(cls % 10)

    print(f"  Training tens digits: {sorted(train_tens)} + empty")
    print(f"  Training units digits: {sorted(train_units)}")

    # Check generalization to test classes
    if Config.TEST_CLASSES:
        print(f"\n  Checking generalization to test classes:")
        can_generalize_all = True

        for test_cls in Config.TEST_CLASSES:
            if test_cls < 10:
```

```
                if test_cls in train_units:
                    print(f" Class {test_cls}: covered (single digit)")
                else:
                    print(f" Class {test_cls}: NOT covered")
                    can_generalize_all = False
            else:
                tens = test_cls // 10
                units = test_cls % 10

                tens_covered = tens in train_tens
                units_covered = units in train_units

                if tens_covered and units_covered:
                    print(f" Class {test_cls} = {tens}{units}: covered")
                else:
                    missing = []
                    if not tens_covered:
                        missing.append(f"tens={tens}")
                    if not units_covered:
                        missing.append(f"units={units}")
                    print(f" Class {test_cls}: NOT covered ({',
'.join(missing)})")
                    can_generalize_all = False

        if can_generalize_all:
            print("\n Model should generalize to all test classes!")
        else:
            print("\n WARNING: Model may struggle with some test classes")

    print("\n" + "="70)
    print("CONFIGURATION SUMMARY")
    print("="70)
    print(f" Data root: {Config.DATA_ROOT}")
    print(f" Train classes: {Config.TRAIN_CLASSES}")
    print(f" Test classes: {Config.TEST_CLASSES}")
    print(f" Validation split: {Config.VAL_SPLIT100:.0f}%")
    print(f" Backbone: {Config.BACKBONE}")
    print(f" Image size: {Config.IMG_SIZE}x{Config.IMG_SIZE}")
    print(f" Sequence length: {Config.SEQ_LENGTH} frames")
    print(f" Batch size: {Config.BATCH_SIZE}")
```

```
    print(f" Epochs: {Config.NUM_EPOCHS}")
    print(f" Learning rate: {Config.LEARNING_RATE}")
    print(f" Checkpoints: {Config.CHECKPOINT_DIR}/")
    print(f" Results: {Config.RESULTS_DIR}/")
    print("="70 + "\n")


if __name__ == "__main__":
    validate_config()
```

This is a configuration file that sets all the parameters for training a jersey number recognition model.

WHAT THIS FILE DOES:

1. Loads environment variables

   Reads a .env file if it exists

   Gets the DATA_ROOT path from environment variable (so you don't hardcode paths)

2. Defines configuration parameters in categories:

DATA CONFIGURATION:

   DATA_ROOT: Where your dataset is located

   TRAIN_CLASSES: Which jersey numbers to use for training (7 numbers: 4, 6, 8, 9, 49, 66, 89)

   TEST_CLASSES: Which jersey numbers to use for testing (3 numbers: 48, 64, 88)

   VAL_SPLIT: 20% of training data used for validation

MODEL CONFIGURATION:

   NUM_DIGIT_CLASSES: 11 (digits 0-9 + "empty" for no tens digit)

   IMG_SIZE: 64x64 pixels (small images)

   SEQ_LENGTH: 5 frames per sequence

BACKBONE: Uses ResNet18 as the base model

Uses both spatial and temporal attention

Uses bidirectional LSTM

TRAINING CONFIGURATION:

BATCH_SIZE: 16 images per batch

NUM_EPOCHS: 30 training cycles

LEARNING_RATE: 0.001

Uses mixed precision training (faster, less memory)

Uses class weights to handle imbalanced data

DATA AUGMENTATION:

Small rotations (5 degrees)

Small translations

Brightness/contrast adjustments

All safe augmentations that don't change the digit identity

SYSTEM CONFIGURATION:

NUM_WORKERS: 4 parallel data loaders

Uses GPU (cuda) if available

Fixed random seed (42) for reproducibility

OUTPUT PATHS:

Creates directories for checkpoints, results, logs

3. Key features of this config:

Class Coverage Analysis:

The validate_config function checks if the training data covers all digits needed for test numbers:

- Training has digits: 4, 6, 8, 9, 4, 6, 6, 8, 9 (from 49, 66, 89)
- Testing has digits: 4, 8, 6, 4, 8, 8 (from 48, 64, 88)
- Since test digits (4, 6, 8) are all in training, the model should generalize

Example:
Test number 48:

- Tens digit 4: Seen in training (from 49)
- Units digit 8: Seen in training (from 8 and 89)
- Both digits are covered, so model can predict 48

4. Configuration Validation:
The validate() method checks for:

- Data path exists
- No overlap between train and test classes
- Hyperparameters are reasonable
- GPU availability

5. Directory Creation:
The create_dirs() method makes sure all necessary folders exist for saving:

- Model checkpoints
- Results
- Logs
- Confusion matrices
- Visualizations

6. Main Idea:
This config separates train/test by jersey numbers:

Train on some numbers (4, 6, 8, 9, 49, 66, 89)

Test on different numbers (48, 64, 88)

But all individual digits (4, 6, 8, 9) appear in both sets

This tests if the model learns to recognize individual digits, not just memorizing specific jersey numbers

The validation function shows if this approach will work by checking digit coverage between train and test sets.

model.py

```
# model.py

"""
Enhanced Two-Head Temporal Jersey Recognition Model (CRNN)
Architecture: ResNet18/34 (CNN) + Bidirectional LSTM + Temporal Attention
+ Dual Classification Heads

Key Features:
- Temporal attention mechanism (no information bottleneck)
- Bidirectional LSTM for better context
- Spatial attention for focusing on jersey region
- Flexible backbone selection
- Better classification heads with batch normalization
- Support for variable-length sequences with padding masks
- Gradient clipping support
"""

import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import models
from typing import Tuple, Optional


class SpatialAttention(nn.Module):
```

```
    """
    Spatial attention module to focus on jersey number regions
    Generates attention weights across spatial dimensions
    """
    def __init__(self, in_channels):
        super(SpatialAttention, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_channels, in_channels // 8, kernel_size=1),
            nn.BatchNorm2d(in_channels // 8),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels // 8, 1, kernel_size=1),
            nn.Sigmoid()
        )

    def forward(self, x):
        """
        Args:
            x: [Batch, Channels, H, W]
        Returns:
            Attention-weighted features
        """
        attention = self.conv(x)  # [Batch, 1, H, W]
        return x  attention


class TemporalAttention(nn.Module):
    """
    Temporal attention mechanism to weight different frames
    Learns which frames are most informative for digit recognition
    """
    def __init__(self, hidden_dim):
        super(TemporalAttention, self).__init__()
        self.attention = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim // 2),
            nn.Tanh(),
            nn.Linear(hidden_dim // 2, 1)
        )

    def forward(self, lstm_out, mask=None):
        """
```

```
        Args:
            lstm_out: [Batch, Seq, Hidden]
            mask: [Batch, Seq] - Optional padding mask (1 for valid, 0 for
padding)
        Returns:
            context: [Batch, Hidden] - Weighted sum of temporal features
            weights: [Batch, Seq] - Attention weights
        """
        # Compute attention scores
        scores = self.attention(lstm_out)  # [Batch, Seq, 1]
        scores = scores.squeeze(-1)  # [Batch, Seq]

        # Apply mask if provided
        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)

        # Softmax to get attention weights
        weights = F.softmax(scores, dim=1)  # [Batch, Seq]

        # Weighted sum
        context = torch.bmm(weights.unsqueeze(1), lstm_out)  # [Batch, 1,
Hidden]
        context = context.squeeze(1)  # [Batch, Hidden]

        return context, weights


class EnhancedDigitHead(nn.Module):
    """
    Enhanced classification head with batch normalization and residual
connection
    """
    def __init__(self, input_dim, num_classes, dropout=0.3):
        super(EnhancedDigitHead, self).__init__()

        hidden_dim = input_dim // 2

        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.bn1 = nn.BatchNorm1d(hidden_dim)
        self.dropout1 = nn.Dropout(dropout)
```

```
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)
        self.bn2 = nn.BatchNorm1d(hidden_dim)
        self.dropout2 = nn.Dropout(dropout)

        self.fc_out = nn.Linear(hidden_dim, num_classes)

        # Residual connection projection if needed
        self.shortcut = nn.Linear(input_dim, hidden_dim) if input_dim !=
hidden_dim else None

    def forward(self, x):
        """
        Args:
            x: [Batch, input_dim]
        Returns:
            logits: [Batch, num_classes]
        """
        identity = x

        # First block
        out = self.fc1(x)
        out = self.bn1(out)
        out = F.relu(out)
        out = self.dropout1(out)

        # Second block with residual
        if self.shortcut is not None:
            identity = self.shortcut(identity)

        out = self.fc2(out)
        out = self.bn2(out)
        out = out + identity  # Residual connection
        out = F.relu(out)
        out = self.dropout2(out)

        # Output
        logits = self.fc_out(out)

        return logits
```

```python
class JerseyTemporalNet(nn.Module):
    """
    Enhanced Two-Head Temporal Jersey Recognition Model
    """
    def __init__(
        self,
        num_classes=11,
        hidden_dim=256,
        dropout=0.3,
        backbone='resnet18',
        use_spatial_attention=True,
        use_temporal_attention=True,
        bidirectional=True
    ):
        """
        Args:
            num_classes (int): Number of digit classes (0-9 + empty = 11)
            hidden_dim (int): LSTM hidden dimension
            dropout (float): Dropout probability
            backbone (str): CNN backbone ('resnet18', 'resnet34',
'resnet50')
            use_spatial_attention (bool): Whether to use spatial attention
            use_temporal_attention (bool): Whether to use temporal
attention
            bidirectional (bool): Whether to use bidirectional LSTM
        """
        super(JerseyTemporalNet, self).__init__()

        self.use_spatial_attention = use_spatial_attention
        self.use_temporal_attention = use_temporal_attention
        self.bidirectional = bidirectional

        # ============================================
        # A. SPATIAL FEATURE EXTRACTOR (CNN BACKBONE)
        # ============================================
        self.backbone_name = backbone
        self.backbone, self.feature_dim = self._build_backbone(backbone)
```

```
        # Spatial attention module
        if use_spatial_attention:
            # Get the number of channels before global pooling
            # For ResNet: layer4 outputs different channels per variant
            if 'resnet18' in backbone or 'resnet34' in backbone:
                spatial_channels = 512
            else:
                spatial_channels = 2048
            self.spatial_attention = SpatialAttention(spatial_channels)

        # ============================================
        # B. TEMPORAL AGGREGATOR (LSTM)
        # ============================================
        self.lstm = nn.LSTM(
            input_size=self.feature_dim,
            hidden_size=hidden_dim,
            num_layers=2,
            batch_first=True,
            dropout=dropout if dropout > 0 else 0,
            bidirectional=bidirectional
        )

        # Effective hidden dimension after LSTM
        lstm_output_dim = hidden_dim  2 if bidirectional else hidden_dim

        # Temporal attention module
        if use_temporal_attention:
            self.temporal_attention = TemporalAttention(lstm_output_dim)
            self.aggregation_method = 'attention'
        else:
            self.aggregation_method = 'mean'  # Fallback to mean pooling

        # ============================================
        # C. CLASSIFICATION HEADS
        # ============================================
        self.dropout = nn.Dropout(dropout)

        # Head 1: Tens Place Digit (0-9, or 10 for "empty")
        self.head_digit1 = EnhancedDigitHead(lstm_output_dim, num_classes,
dropout)
```

```
        # Head 2: Units Place Digit (0-9)
        self.head_digit2 = EnhancedDigitHead(lstm_output_dim, num_classes,
dropout)

        # Initialize weights
        self._initialize_weights()

    def _freeze_backbone_layers(self, backbone, freeze_percentage=0.6):
        """
        Freeze backbone layers intelligently.

        Args:
            backbone: The CNN backbone module
            freeze_percentage: Percentage of layers to freeze (0.0 to 1.0)

        Returns:
            Number of frozen parameters, total parameters
        """
        # Get all parameters with their names
        params = list(backbone.named_parameters())
        total_layers = len(params)
        freeze_until_idx = int(total_layers  freeze_percentage)

        frozen_params = 0
        total_params = 0

        print(f"\n Freezing {freeze_percentage100:.0f}% of backbone layers
({freeze_until_idx}/{total_layers}):")

        for idx, (name, param) in enumerate(params):
            total_params += param.numel()

            if idx < freeze_until_idx:
                param.requires_grad = False
                frozen_params += param.numel()
                if idx < 3:
                    print(f" Frozen: {name} ({param.numel():,} params)")
            else:
                param.requires_grad = True
```

```
                if idx == freeze_until_idx:  # Print first unfrozen
                    print(f"  ... ({freeze_until_idx - 3} more frozen
layers)")
                    print(f" Trainable: {name} ({param.numel():,} params)")
                elif idx < freeze_until_idx + 2:
                    print(f" Trainable: {name} ({param.numel():,} params)")

        print(f"\n  Total: {frozen_params:,} / {total_params:,} frozen
({frozen_params/total_params100:.1f}%)")

        return frozen_params, total_params

    def _build_backbone(self, backbone_name):
        """Build and configure the CNN backbone"""
        if backbone_name == 'resnet18':
            resnet = models.resnet18(pretrained=True)
        elif backbone_name == 'resnet34':
            resnet = models.resnet34(pretrained=True)
        elif backbone_name == 'resnet50':
            resnet = models.resnet50(pretrained=True)
        else:
            raise ValueError(f"Unsupported backbone: {backbone_name}")

        feature_dim = resnet.fc.in_features

        # Remove final FC and global pooling
        if self.use_spatial_attention:
            # Keep layer4 for spatial attention, remove avgpool and fc
            backbone = nn.Sequential(list(resnet.children())[:-2])
        else:
            # Remove only the fc layer
            backbone = nn.Sequential(list(resnet.children())[:-1])

        # Freeze layers intelligently
        self._freeze_backbone_layers(backbone, freeze_percentage=0.6)

        return backbone, feature_dim

    def _initialize_weights(self):
        """Initialize weights for new layers"""
```

```
        for m in self.modules():
            if isinstance(m, nn.Linear):
                nn.init.xavier_uniform_(m.weight)
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.LSTM):
                for name, param in m.named_parameters():
                    if 'weight_ih' in name:
                        nn.init.xavier_uniform_(param.data)
                    elif 'weight_hh' in name:
                        nn.init.orthogonal_(param.data)
                    elif 'bias' in name:
                        nn.init.constant_(param.data, 0)

    def unfreeze_backbone(self, percentage=0.3):
        """
        Gradually unfreeze more backbone layers

        Args:
            percentage (float): Percentage of layers to unfreeze (0.0 to
1.0)
        """
        params = list(self.backbone.named_parameters())
        num_to_unfreeze = int(len(params)  percentage)

        print(f"\n Unfreezing {percentage100:.0f}% of backbone layers
({num_to_unfreeze} layers):")

        for idx, (name, param) in enumerate(params[-num_to_unfreeze:]):
            param.requires_grad = True
            if idx < 3:  # Print first 3
                print(f" Unfrozen: {name}")

    def forward(self, x, mask=None):
        """
        Forward pass

        Args:
            x: Input tensor [Batch, Seq_Length, Channels, Height, Width]
```

```
            mask: Optional padding mask [Batch, Seq_Length] (1 for valid, 0
for padding)

        Returns:
            d1_logits: Logits for tens place [Batch, num_classes]
            d2_logits: Logits for units place [Batch, num_classes]
            attention_weights: Temporal attention weights (if enabled)
[Batch, Seq]
        """
        batch_size, seq_length, c, h, w = x.size()

        # ================================================
        # Step 1: Extract spatial features from each frame
        # ================================================
        x = x.view(batch_size  seq_length, c, h, w)

        # Pass through CNN backbone
        features = self.backbone(x)

        # Apply spatial attention if enabled
        if self.use_spatial_attention and features.dim() == 4:
            features = self.spatial_attention(features)
            # Global average pooling
            features = F.adaptive_avg_pool2d(features, (1, 1))

        # Flatten spatial dimensions
        features = features.view(batch_size  seq_length, -1)

        # Reshape back to sequence
        features = features.view(batch_size, seq_length, -1)

        # ================================================
        # Step 2: Temporal modeling with LSTM
        # ================================================
        if mask is not None:
            # Pack padded sequence for efficiency
            lengths = mask.sum(dim=1).cpu()
            packed = nn.utils.rnn.pack_padded_sequence(
                features, lengths, batch_first=True, enforce_sorted=False
            )
```

```
            lstm_out, (h_n, c_n) = self.lstm(packed)
            lstm_out, _ = nn.utils.rnn.pad_packed_sequence(lstm_out,
batch_first=True)
        else:
            lstm_out, (h_n, c_n) = self.lstm(features)

        # ============================================
        # Step 3: Temporal aggregation
        # ============================================
        attention_weights = None

        if self.use_temporal_attention:
            # Use attention mechanism
            aggregated, attention_weights =
self.temporal_attention(lstm_out, mask)
        else:
            # Fallback to mean pooling
            if mask is not None:
                # Masked mean
                mask_expanded = mask.unsqueeze(-1).expand_as(lstm_out)
                sum_out = (lstm_out  mask_expanded).sum(dim=1)
                lengths = mask.sum(dim=1, keepdim=True).float()
                aggregated = sum_out / lengths
            else:
                aggregated = lstm_out.mean(dim=1)

        # Apply dropout
        aggregated = self.dropout(aggregated)

        # ============================================
        # Step 4: Digit classification
        # ============================================
        d1_logits = self.head_digit1(aggregated)
        d2_logits = self.head_digit2(aggregated)

        if attention_weights is not None:
            return d1_logits, d2_logits, attention_weights
        else:
            return d1_logits, d2_logits
```

```
    def predict(self, x, mask=None, return_confidence=True):
        """
        Predict digit labels (for inference)

        Args:
            x: Input tensor [Batch, Seq_Length, Channels, Height, Width]
            mask: Optional padding mask [Batch, Seq_Length]
            return_confidence: Whether to return confidence scores

        Returns:
            d1_pred: Predicted tens digit [Batch]
            d2_pred: Predicted units digit [Batch]
            confidence: Average confidence scores [Batch] (if
return_confidence=True)
            attention_weights: Temporal attention weights [Batch, Seq] (if
enabled)
        """
        self.eval()
        with torch.no_grad():
            forward_out = self.forward(x, mask)

            if len(forward_out) == 3:
                d1_logits, d2_logits, attention_weights = forward_out
            else:
                d1_logits, d2_logits = forward_out
                attention_weights = None

            # Get probabilities
            d1_probs = F.softmax(d1_logits, dim=1)
            d2_probs = F.softmax(d2_logits, dim=1)

            # Get predictions
            d1_pred = torch.argmax(d1_probs, dim=1)
            d2_pred = torch.argmax(d2_probs, dim=1)

            if return_confidence:
                # Get confidence (max probability)
                d1_conf, _ = torch.max(d1_probs, dim=1)
                d2_conf, _ = torch.max(d2_probs, dim=1)
```

```
                # Average confidence
                confidence = (d1_conf + d2_conf) / 2

                if attention_weights is not None:
                    return d1_pred, d2_pred, confidence, attention_weights
                else:
                    return d1_pred, d2_pred, confidence
            else:
                if attention_weights is not None:
                    return d1_pred, d2_pred, attention_weights
                else:
                    return d1_pred, d2_pred

    def get_num_params(self, trainable_only=True):
        """Return the number of parameters"""
        if trainable_only:
            return sum(p.numel() for p in self.parameters() if
p.requires_grad)
        else:
            return sum(p.numel() for p in self.parameters())


if __name__ == "__main__":
    print("="*70)
    print("Testing Enhanced JerseyTemporalNet Model")
    print("="*70)

    # Test configuration
    batch_size = 4
    seq_length = 8
    img_size = 224
    num_classes = 11

    # Create model
    print("\n1. Creating model...")
    model = JerseyTemporalNet(
        num_classes=num_classes,
        hidden_dim=256,
        dropout=0.3,
        backbone='resnet18',
```

```
        use_spatial_attention=True,
        use_temporal_attention=True,
        bidirectional=True
    )

    print("   Model created!")
    print(f"   Total parameters:
{model.get_num_params(trainable_only=False):,}")
    print(f"   Trainable parameters:
{model.get_num_params(trainable_only=True):,}")

    # Create dummy input
    dummy_input = torch.randn(batch_size, seq_length, 3, img_size,
img_size)
    print("\n2. Testing forward pass...")
    print(f"   Input shape: {dummy_input.shape}")

    # Forward pass
    model.eval()
    output = model(dummy_input)

    if len(output) == 3:
        d1_logits, d2_logits, attention = output
        print(f"   Output D1 shape: {d1_logits.shape}")
        print(f"   Output D2 shape: {d2_logits.shape}")
        print(f"   Attention weights shape: {attention.shape}")
    else:
        d1_logits, d2_logits = output
        print(f"   Output D1 shape: {d1_logits.shape}")
        print(f"   Output D2 shape: {d2_logits.shape}")

    print("\n" + "="70)
    print(" All tests passed!")
    print("="70)
```

This is a neural network model for recognizing jersey numbers from video sequences.

WHAT THIS MODEL DOES:

It takes sequences of images (multiple frames showing a jersey number) and predicts two digits:
1. First digit (tens place) - can be 0-9 or "empty" (10) for single-digit numbers
2. Second digit (units place) - 0-9

MODEL ARCHITECTURE HAS 4 MAIN PARTS:

1. SPATIAL FEATURE EXTRACTOR (CNN BACKBONE)
- Uses ResNet18/34/50 as the base
- Takes each image frame and extracts visual features
- Includes Spatial Attention module to focus on jersey number region
- Example: If frame shows a player, attention helps focus on the jersey number area

2. TEMPORAL AGGREGATOR (LSTM)
- Takes features from all frames in the sequence
- Uses LSTM (Long Short-Term Memory) to understand temporal patterns
- Can be bidirectional - looks at frames both forward and backward in time
- Example: If number is partially visible in some frames, LSTM combines information across all frames

3. TEMPORAL ATTENTION
- Weights different frames based on importance
- Some frames might be clearer than others
- Gives higher weight to frames where number is most visible
- Example: Frame where player is facing camera gets more weight than side view

4. TWO CLASSIFICATION HEADS
- Head 1: Predicts first digit (tens place)
- Head 2: Predicts second digit (units place)
- Both heads use same features but have separate classifiers
- Example: For number "46": Head 1 predicts "4", Head 2 predicts "6"

KEY FEATURES:

Freezing Backbone Layers:
- ResNet starts with pretrained weights (trained on ImageNet)
- Freezes 60% of backbone layers initially (keeps them fixed)
- Allows fine-tuning only higher layers specific to jersey numbers
- Prevents overfitting when dataset is small

Spatial Attention:
- Learns to focus on relevant parts of the image
- Creates a "heatmap" highlighting jersey number region
- Helps ignore irrelevant background

Temporal Attention:
- Learns which frames are most informative
- Outputs weights showing importance of each frame
- Can see which frames the model considered important

Handling Variable Sequences:
- Uses masking for sequences with different lengths
- Ignores padding frames (empty frames added to make sequences same length)

TRAINING STRATEGY:

Two Separate Losses:
- Loss for first digit prediction
- Loss for second digit prediction
- Total loss = sum of both losses

Gradual Unfreezing:
- Starts with most backbone layers frozen
- Gradually unfreezes layers during training
- Allows model to adapt while keeping useful pretrained features

MODULE BREAKDOWN:

SpatialAttention class:
- Takes CNN features (e.g., 512 channels)
- Learns which spatial locations are important
- Multiplies features by attention weights

TemporalAttention class:
- Takes LSTM outputs for each frame
- Learns weights for each frame
- Combines frames using weighted average

EnhancedDigitHead class:
- Classification head for digit prediction
- Uses residual connections (helps training)
- Has dropout for regularization

JerseyTemporalNet class:
- Main model combining all components
- Has forward() method for training
- Has predict() method for inference

WORKFLOW EXAMPLE:

Input: 5 frames showing jersey number "23"

1. Each frame goes through ResNet → extract features
2. Spatial attention focuses on number region in each frame
3. LSTM processes all 5 frames together
4. Temporal attention weights frames (e.g., frame 3 gets 0.4 weight, frame 1 gets 0.1, etc.)
5. Weighted features go to two heads:
   - Head 1 predicts: "2" (tens digit)
   - Head 2 predicts: "3" (units digit)

OUTPUT FORMAT:
- Two sets of probabilities (11 classes each: 0-9 + "empty")
- Attention weights showing frame importance
- Confidence scores for predictions

TESTING SECTION:
The bottom code tests the model with dummy data to ensure:
- Model can be created
- Forward pass works
- Output shapes are correct
- Parameters are counted correctly

WHY THIS DESIGN:
- Uses temporal information: Multiple frames help when some frames are blurry/occluded
- Attention mechanisms: Focus on what matters (both spatially and temporally)
- Two-head design: Natural for two-digit numbers
- Pretrained backbone: Leverages general image knowledge
- Flexibility: Can handle different sequence lengths, different backbones

train.py

```
# train.py
import os
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Subset
from torch.cuda.amp import autocast, GradScaler
from torch.utils.tensorboard import SummaryWriter
from torchvision import transforms
from tqdm import tqdm
import numpy as np
```

```
import json
import random
from datetime import datetime
import warnings
warnings.filterwarnings('ignore')

from config import Config
from dataset import JerseySequenceDataset
from model import JerseyTemporalNet


# ======================= MIXUP ======================
def mixup_data(x, d1_target, d2_target, alpha=1.0):
    if alpha > 0:
        lam = np.random.beta(alpha, alpha)
    else:
        lam = 1.0
    batch_size = x.size(0)
    index = torch.randperm(batch_size).to(x.device)
    mixed_x = lam  x + (1 - lam)  x[index]
    return mixed_x, d1_target, d1_target[index], d2_target,
d2_target[index], lam

def mixup_criterion(criterion, pred, y_a, y_b, lam):
    return lam  criterion(pred, y_a) + (1 - lam)  criterion(pred, y_b)
# ============================================================


class EarlyStopping:
    def __init__(self, patience=10, min_delta=0.001, verbose=True):
        self.patience = patience
        self.min_delta = min_delta
        self.counter = 0
        self.best_loss = None
        self.early_stop = False
        self.verbose = verbose

    def __call__(self, val_loss):
        if self.best_loss is None:
            self.best_loss = val_loss
```

```
        elif val_loss > self.best_loss - self.min_delta:
            self.counter += 1
            if self.verbose:
                print(f"Early stopping counter:
{self.counter}/{self.patience}")
            if self.counter >= self.patience:
                self.early_stop = True
        else:
            self.best_loss = val_loss
            self.counter = 0
        return self.early_stop


class Trainer:
    def __init__(self):
        self.device = torch.device('cuda' if torch.cuda.is_available() else
'cpu')
        print(f"Using device: {self.device}")
        Config.create_dirs()
        self.use_amp = Config.USE_MIXED_PRECISION and self.device.type ==
'cuda'
        self.scaler = GradScaler() if self.use_amp else None

        # Initialize TensorBoard
        timestamp = datetime.now().strftime('%Y%m%d_%H%M%S')
        self.writer = SummaryWriter(os.path.join(Config.LOGS_DIR,
f'run_{timestamp}'))
        print(f" TensorBoard logging to:
{Config.LOGS_DIR}/run_{timestamp}")

        self.setup_data()
        self.setup_model()

        self.early_stopping =
EarlyStopping(patience=Config.EARLY_STOPPING_PATIENCE)
        self.history = {k: [] for k in ['train_loss', 'val_loss',
'train_acc', 'val_acc',
                                        'val_d1_acc', 'val_d2_acc',
'val_jersey_acc',
                                        'epoch', 'learning_rate']}
```

```
        self.best_val_acc = 0.0

    def setup_data(self):
        print("\n" + "="60 + "\nSETTING UP DATA\n" + "="60)

        self.train_transform = transforms.Compose([
            transforms.Resize((Config.IMG_SIZE, Config.IMG_SIZE)),
            transforms.RandomAffine(degrees=Config.AUG_ROTATION_DEGREES,
                                    translate=Config.AUG_TRANSLATE,
scale=Config.AUG_SCALE),
            transforms.ColorJitter(brightness=Config.AUG_BRIGHTNESS,
contrast=Config.AUG_CONTRAST),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
0.224, 0.225]),
        ])
        self.val_transform = transforms.Compose([
            transforms.Resize((Config.IMG_SIZE, Config.IMG_SIZE)),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
0.224, 0.225])
        ])

        temp_ds = JerseySequenceDataset(Config.DATA_ROOT, transform=None,
                                        seq_length=Config.SEQ_LENGTH,

allowed_classes=Config.TRAIN_CLASSES, mode='full')
        indices =
np.random.RandomState(Config.SEED).permutation(len(temp_ds))
        split = int(Config.VAL_SPLIT  len(indices))
        val_idx, train_idx = indices[:split], indices[split:]

        train_ds = Subset(JerseySequenceDataset(Config.DATA_ROOT,
self.train_transform,

seq_length=Config.SEQ_LENGTH,

allowed_classes=Config.TRAIN_CLASSES,
                                                mode='train',
sampling_strategy=Config.TEMPORAL_SAMPLING),
```

```
                          train_idx)
        val_ds = Subset(JerseySequenceDataset(Config.DATA_ROOT,
self.val_transform,
                                              seq_length=Config.SEQ_LENGTH,

allowed_classes=Config.TRAIN_CLASSES,
                                              mode='val',
sampling_strategy='uniform'),
                        val_idx)

        full_train_ds = JerseySequenceDataset(Config.DATA_ROOT,
self.train_transform,
                                              seq_length=Config.SEQ_LENGTH,

allowed_classes=Config.TRAIN_CLASSES, mode='train')
        self.w_d1, self.w_d2 =
full_train_ds.get_class_weights(method='effective_number')

        self.train_loader = DataLoader(train_ds,
batch_size=Config.BATCH_SIZE, shuffle=True,
                                       num_workers=Config.NUM_WORKERS,
pin_memory=True,

persistent_workers=Config.NUM_WORKERS > 0)
        self.val_loader = DataLoader(val_ds, batch_size=Config.BATCH_SIZE,
shuffle=False,
                                     num_workers=Config.NUM_WORKERS,
pin_memory=True,
                                     persistent_workers=Config.NUM_WORKERS
> 0)

        print(f"Train batches: {len(self.train_loader)} | Val batches:
{len(self.val_loader)}")

    def setup_model(self):
        print("\n" + "="60 + "\nSETTING UP MODEL\n" + "="60)
        self.model = JerseyTemporalNet(
            num_classes=Config.NUM_DIGIT_CLASSES,
            hidden_dim=Config.HIDDEN_DIM,
            dropout=Config.DROPOUT,
```

```
            backbone=Config.BACKBONE,
            use_spatial_attention=Config.USE_SPATIAL_ATTENTION,
            use_temporal_attention=Config.USE_TEMPORAL_ATTENTION,
            bidirectional=Config.BIDIRECTIONAL_LSTM
        ).to(self.device)

        self.criterion_d1 =
nn.CrossEntropyLoss(weight=self.w_d1.to(self.device), label_smoothing=0.1)
        self.criterion_d2 =
nn.CrossEntropyLoss(weight=self.w_d2.to(self.device), label_smoothing=0.1)

        self.optimizer = optim.AdamW(self.model.parameters(),
lr=Config.LEARNING_RATE, weight_decay=1e-4)
        self.scheduler =
optim.lr_scheduler.CosineAnnealingLR(self.optimizer,
T_max=Config.NUM_EPOCHS)

        # Log model graph to TensorBoard
        dummy_input = torch.randn(1, Config.SEQ_LENGTH, 3, Config.IMG_SIZE,
Config.IMG_SIZE).to(self.device)
        try:
            self.writer.add_graph(self.model, dummy_input)
            print(" Model graph saved to TensorBoard")
        except Exception as e:
            print(f" Could not save model graph: {e}")

        print("Model ready — MixUp + effective_number weights with
suppressed empty class")

    def forward_with_output(self, x):
        output = self.model(x)
        if len(output) == 3:
            d1_logit, d2_logit, _ = output
        else:
            d1_logit, d2_logit = output
        return d1_logit, d2_logit

    def calculate_metrics(self, d1p, d2p, d1t, d2t):
        both = ((d1p == d1t) & (d2p == d2t)).float().mean().item()  100
        d1_acc = (d1p == d1t).float().mean().item()  100
```

```
        d2_acc = (d2p == d2t).float().mean().item()  100
        jersey_pred = torch.where(d1p == 10, d2p, d1p  10 + d2p)
        jersey_true = torch.where(d1t == 10, d2t, d1t  10 + d2t)
        jersey_acc = (jersey_pred == jersey_true).float().mean().item()
100
        return {'both_acc': both, 'd1_acc': d1_acc, 'd2_acc': d2_acc,
'jersey_acc': jersey_acc}

    def train_epoch(self, epoch):
        self.model.train()
        total_loss = 0.0
        all_d1_pred, all_d2_pred, all_d1_true, all_d2_true = [], [], [], []

        for seq, d1_true, d2_true in tqdm(self.train_loader, desc=f"Epoch
{epoch} [Train]"):
            seq = seq.to(self.device)
            d1_true = d1_true.to(self.device)
            d2_true = d2_true.to(self.device)

            self.optimizer.zero_grad()

            if random.random() < 0.5:  # MixUp
                seq_mix, d1a, d1b, d2a, d2b, lam = mixup_data(seq, d1_true,
d2_true, alpha=1.0)
                with autocast(enabled=self.use_amp):
                    d1_logit, d2_logit = self.forward_with_output(seq_mix)
                    loss = (mixup_criterion(self.criterion_d1, d1_logit,
d1a, d1b, lam) +
                            mixup_criterion(self.criterion_d2, d2_logit,
d2a, d2b, lam))
            else:
                with autocast(enabled=self.use_amp):
                    d1_logit, d2_logit = self.forward_with_output(seq)
                    loss = self.criterion_d1(d1_logit, d1_true) +
self.criterion_d2(d2_logit, d2_true)

            if self.use_amp:
                self.scaler.scale(loss).backward()
                self.scaler.unscale_(self.optimizer)
```

```
                torch.nn.utils.clip_grad_norm_(self.model.parameters(),
Config.GRADIENT_CLIP_VALUE)
                self.scaler.step(self.optimizer)
                self.scaler.update()
            else:
                loss.backward()
                torch.nn.utils.clip_grad_norm_(self.model.parameters(),
Config.GRADIENT_CLIP_VALUE)
                self.optimizer.step()

            total_loss += loss.item()
            all_d1_pred.append(d1_logit.argmax(1).cpu())
            all_d2_pred.append(d2_logit.argmax(1).cpu())
            all_d1_true.append(d1_true.cpu())
            all_d2_true.append(d2_true.cpu())

        metrics = self.calculate_metrics(
            torch.cat(all_d1_pred), torch.cat(all_d2_pred),
            torch.cat(all_d1_true), torch.cat(all_d2_true)
        )

        avg_loss = total_loss / len(self.train_loader)

        # Log to TensorBoard
        self.writer.add_scalar('Loss/train', avg_loss, epoch)
        self.writer.add_scalar('Accuracy/train_both', metrics['both_acc'],
epoch)
        self.writer.add_scalar('Accuracy/train_d1', metrics['d1_acc'],
epoch)
        self.writer.add_scalar('Accuracy/train_d2', metrics['d2_acc'],
epoch)
        self.writer.add_scalar('Accuracy/train_jersey',
metrics['jersey_acc'], epoch)

        return avg_loss, metrics

    def validate(self, epoch):
        self.model.eval()
        total_loss = 0.0
        all_d1_pred, all_d2_pred, all_d1_true, all_d2_true = [], [], [], []
```

```
        with torch.no_grad():
            for seq, d1_true, d2_true in tqdm(self.val_loader, desc=f"Epoch
{epoch} [Val]  "):
                seq, d1_true, d2_true = seq.to(self.device),
d1_true.to(self.device), d2_true.to(self.device)
                with autocast(enabled=self.use_amp):
                    d1_logit, d2_logit = self.forward_with_output(seq)
                    loss = self.criterion_d1(d1_logit, d1_true) +
self.criterion_d2(d2_logit, d2_true)

                total_loss += loss.item()
                all_d1_pred.append(d1_logit.argmax(1).cpu())
                all_d2_pred.append(d2_logit.argmax(1).cpu())
                all_d1_true.append(d1_true.cpu())
                all_d2_true.append(d2_true.cpu())

        metrics = self.calculate_metrics(
            torch.cat(all_d1_pred), torch.cat(all_d2_pred),
            torch.cat(all_d1_true), torch.cat(all_d2_true)
        )

        avg_loss = total_loss / len(self.val_loader)

        # Log to TensorBoard
        self.writer.add_scalar('Loss/val', avg_loss, epoch)
        self.writer.add_scalar('Accuracy/val_both', metrics['both_acc'],
epoch)
        self.writer.add_scalar('Accuracy/val_d1', metrics['d1_acc'], epoch)
        self.writer.add_scalar('Accuracy/val_d2', metrics['d2_acc'], epoch)
        self.writer.add_scalar('Accuracy/val_jersey',
metrics['jersey_acc'], epoch)

        return avg_loss, metrics

    def train(self):
        print("\n" + "="60)
        print("TRAINING WITH MIXUP")
        print("="60)
```

```
        for epoch in range(1, Config.NUM_EPOCHS + 1):
            train_loss, train_metrics = self.train_epoch(epoch)
            val_loss, val_metrics = self.validate(epoch)
            self.scheduler.step()

            self.history['epoch'].append(epoch)
            self.history['train_loss'].append(train_loss)
            self.history['val_loss'].append(val_loss)
            self.history['train_acc'].append(train_metrics['both_acc'])
            self.history['val_acc'].append(val_metrics['both_acc'])
            self.history['val_d1_acc'].append(val_metrics['d1_acc'])
            self.history['val_d2_acc'].append(val_metrics['d2_acc'])

self.history['val_jersey_acc'].append(val_metrics['jersey_acc'])

self.history['learning_rate'].append(self.optimizer.param_groups[0]['lr'])

            # Log learning rate
            self.writer.add_scalar('Learning_Rate',
self.optimizer.param_groups[0]['lr'], epoch)

            # Log train vs val comparison
            self.writer.add_scalars('Loss/train_vs_val', {
                'train': train_loss,
                'val': val_loss
            }, epoch)

            self.writer.add_scalars('Accuracy/train_vs_val', {
                'train': train_metrics['both_acc'],
                'val': val_metrics['both_acc']
            }, epoch)

            print(f"\nEpoch {epoch:2d} | Val Acc:
{val_metrics['both_acc']:5.2f}% | "
                  f"D1: {val_metrics['d1_acc']:5.2f}% | D2:
{val_metrics['d2_acc']:5.2f}%")

            if val_metrics['both_acc'] > self.best_val_acc:
                self.best_val_acc = val_metrics['both_acc']
                torch.save({
```

```
                    'epoch': epoch,
                    'model_state_dict': self.model.state_dict(),
                    'optimizer_state_dict': self.optimizer.state_dict(),
                    'best_val_acc': self.best_val_acc,
                    'history': self.history
                }, os.path.join(Config.CHECKPOINT_DIR, 'best_model.pth'))
                print(f"NEW BEST MODEL SAVED: {self.best_val_acc:.2f}%")

            if self.early_stopping(val_loss):
                print("Early stopping triggered")
                break

        # Close TensorBoard writer
        self.writer.close()

        print(f"\nTRAINING COMPLETE! Best Val Acc:
{self.best_val_acc:.2f}%")
        print(f" View TensorBoard: tensorboard --logdir={Config.LOGS_DIR}")
        print("Now run: python evaluate.py --checkpoint
checkpoints/best_model.pth")


if __name__ == "__main__":
    torch.manual_seed(Config.SEED)
    np.random.seed(Config.SEED)
    random.seed(Config.SEED)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(Config.SEED)
        torch.backends.cudnn.deterministic = True
        torch.backends.cudnn.benchmark = False

    Trainer().train()
```

This is the main training script that trains the jersey number recognition model.

WHAT THIS SCRIPT DOES:

It trains the neural network using a combination of techniques to recognize jersey numbers from video sequences.

MAIN COMPONENTS:

1. MixUp Augmentation:
- Creates new training samples by mixing two images
- Example: Mixes 30% of image with number "23" + 70% of image with number "45"
- Helps model generalize better and reduces overfitting
- Used randomly during training (50% chance per batch)

2. Early Stopping:
- Stops training if validation loss doesn't improve for 10 epochs
- Prevents overtraining
- Saves the best model

3. Trainer Class:
Manages the entire training process with these steps:

SETUP PHASE:

A. Data Preparation:
- Creates two different transforms:
  - Train transform: Adds augmentation (rotation, color jitter, etc.)
  - Val transform: Simple resize and normalize (no augmentation)
- Splits data into train (80%) and validation (20%)
- Calculates class weights to handle imbalanced data
- Creates data loaders for batching

B. Model Setup:
- Creates the JerseyTemporalNet model
- Sets up loss functions with class weights
- Uses AdamW optimizer with weight decay
- Uses cosine annealing learning rate scheduler
- Sets up TensorBoard for visualization

TRAINING LOOP:

For each epoch (training cycle):

1. Training Phase:
- Sets model to training mode
- For each batch:
  - Randomly applies MixUp (50% chance)
  - Passes sequences through model
  - Calculates loss for both digits
  - If MixUp was used: calculates combined loss from both mixed samples

  - Backpropagates error
  - Uses gradient clipping to prevent exploding gradients
  - Updates model weights
  - If mixed precision training is enabled: uses less memory for faster training
- Logs metrics to TensorBoard

2. Validation Phase:
- Sets model to evaluation mode
- No gradient calculation (faster)
- No augmentation
- Passes validation data through model
- Calculates accuracy metrics:
  - D1 accuracy (tens digit)
  - D2 accuracy (units digit)
  - Both digits correct
  - Jersey number accuracy (reconstructs full number from both digits)
- Logs metrics to TensorBoard

3. Model Saving:
- Saves model if validation accuracy improves
- Saves checkpoint with:
  - Model weights
  - Optimizer state
  - Training history
  - Best accuracy

4. Learning Rate Adjustment:
- Updates learning rate using cosine annealing
- Gradually decreases learning rate over time

5. Early Stopping Check:
- Checks if validation loss hasn't improved
- Stops training if no improvement for 10 epochs

KEY METRICS CALCULATED:

Accuracy Types:
1. D1 Accuracy: How often tens digit is correct (or correctly predicted as "empty" for single digits)
2. D2 Accuracy: How often units digit is correct
3. Both Digits Accuracy: Both D1 and D2 correct simultaneously
4. Jersey Number Accuracy: Full number reconstructed correctly
   - For single digits: matches units digit
   - For two digits: (D1 10 + D2) matches actual number

TRAINING TECHNIQUES USED:

MixUp Augmentation:
- Randomly mixes two training samples
- Helps model learn smoother decision boundaries
- Reduces memorization of training data

Class Weighting:
- Gives more weight to rare classes
- Prevents model from ignoring rare digits

Label Smoothing:
- Makes labels slightly "fuzzy" (not exactly 1.0 for correct class)
- Helps prevent overconfidence
- Improves generalization

Gradient Clipping:
- Limits gradient values
- Prevents exploding gradients during training

Mixed Precision Training:
- Uses lower precision (FP16) where possible
- Faster training, less GPU memory
- Automatically managed by PyTorch

Cosine Annealing Learning Rate:
- Learning rate follows cosine curve
- Starts high, decreases smoothly, may increase slightly at end
- Helps escape local minima

TENSORBOARD VISUALIZATION:
- Logs training/validation losses
- Logs all accuracy metrics
- Logs learning rate changes
- Can visualize model graph
- View with: tensorboard --logdir=logs/

OUTPUT:
- Best model saved as: checkpoints/best_model.pth
- TensorBoard logs in: logs/run_TIMESTAMP/
- Training history saved in checkpoint

FINISHING:

- Prints best validation accuracy
- Suggests running evaluation script
- Closes TensorBoard writer

REPRODUCIBILITY:
- Sets random seeds for Python, NumPy, PyTorch
- Makes training deterministic (same results each run)

The script implements modern training techniques to create a robust model that can recognize jersey numbers even with limited training data, handling class imbalance and preventing overfitting.

evaluate.py

```
# evaluate.py
import os
import torch
import torch.nn as nn
from torch.utils.data import DataLoader
from torchvision import transforms
from tqdm import tqdm
import numpy as np
import json
from collections import defaultdict
from datetime import datetime
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix

from config import Config
from dataset import JerseySequenceDataset
from model import JerseyTemporalNet


class Evaluator:
    def __init__(self, checkpoint_path):
        self.device = torch.device('cuda' if torch.cuda.is_available() else
'cpu')
        print(f"Using device: {self.device}")
        self.checkpoint_path = checkpoint_path
        self.load_checkpoint()
```

```
        self.setup_data()
        self.results = {'overall': {}, 'per_class': {}, 'predictions': [],
'confusion_matrices': {}}

    def load_checkpoint(self):
        print("\n" + "="60)
        print("LOADING MODEL")
        print("="60)

        if not os.path.exists(self.checkpoint_path):
            raise FileNotFoundError(f"Checkpoint not found:
{self.checkpoint_path}")

        checkpoint = torch.load(self.checkpoint_path,
map_location=self.device)


        if 'config' in checkpoint:
            cfg = checkpoint['config']
        else:
            # New format: vars(Config) saved directly
            cfg = {k: v for k, v in checkpoint.items() if k in [
                'num_digit_classes', 'hidden_dim', 'backbone',
'train_classes', 'test_classes'
            ]}

        # Reconstruct model
        self.model = JerseyTemporalNet(
            num_classes=cfg.get('num_digit_classes',
Config.NUM_DIGIT_CLASSES),
            hidden_dim=cfg.get('hidden_dim', Config.HIDDEN_DIM),
            dropout=Config.DROPOUT,
            backbone=cfg.get('backbone', Config.BACKBONE),
            use_spatial_attention=Config.USE_SPATIAL_ATTENTION,
            use_temporal_attention=Config.USE_TEMPORAL_ATTENTION,
            bidirectional=Config.BIDIRECTIONAL_LSTM
        ).to(self.device)

        self.model.load_state_dict(checkpoint['model_state_dict'])
        self.model.eval()
```

```
        print(f"Model loaded from: {self.checkpoint_path}")
        print(f"Train classes: {cfg.get('train_classes',
Config.TRAIN_CLASSES)}")
        print(f"Best val acc: {checkpoint.get('best_val_acc',
'N/A'):.2f}%")
        print(f"Epoch: {checkpoint.get('epoch', 'N/A')}")
        print(f"Parameters: {self.model.get_num_params():,}")

    def setup_data(self):
        print("\n" + "="60)
        print("SETTING UP TEST DATA")
        print("="60)

        self.test_transform = transforms.Compose([
            transforms.Resize((Config.IMG_SIZE, Config.IMG_SIZE)),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
0.224, 0.225])
        ])

        self.test_dataset = JerseySequenceDataset(
            root_dir=Config.DATA_ROOT,
            transform=self.test_transform,
            seq_length=Config.SEQ_LENGTH,
            allowed_classes=Config.TEST_CLASSES,
            mode='test',
            sampling_strategy='uniform'
        )

        self.test_loader = DataLoader(
            self.test_dataset,
            batch_size=Config.BATCH_SIZE,
            shuffle=False,
            num_workers=Config.NUM_WORKERS,
            pin_memory=Config.PIN_MEMORY and self.device.type == 'cuda'
        )

        print(f"Test samples: {len(self.test_dataset)} | Batches:
{len(self.test_loader)}")
```

```
   def evaluate(self):
       print("\n" + "="60)
       print("RUNNING EVALUATION ON HELD-OUT CLASSES:",
Config.TEST_CLASSES)
       print("="60)

       self.model.eval()
       all_d1_true, all_d2_true = [], []
       all_d1_pred, all_d2_pred = [], []
       all_d1_conf, all_d2_conf = [], []

       class_total = defaultdict(int)
       class_correct = defaultdict(int)

       # Anti-empty-bias heuristic parameters
       EMPTY_CLASS = 10
       EMPTY_THRESHOLD = 0.55  # If "empty" has < 55% confidence, use next
best

       total_modified = 0
       total_samples = 0

       with torch.no_grad():
           for seq, d1_true, d2_true in tqdm(self.test_loader,
desc="Evaluating"):
               seq = seq.to(self.device)
               d1_true = d1_true.to(self.device)
               d2_true = d2_true.to(self.device)

               # Safe forward (handles 2 or 3 return values)
               out = self.model(seq)
               if len(out) == 3:
                   d1_logit, d2_logit, _ = out
               else:
                   d1_logit, d2_logit = out

               # D2 prediction
               d2_pred = d2_logit.argmax(1)
```

```
                # D1 prediction with anti-empty-bias heuristic
                d1_probs_softmax = d1_logit.softmax(1)
                d1_pred_list = []
                batch_modified = 0

                for i in range(len(d1_probs_softmax)):
                    probs = d1_probs_softmax[i]

                    if probs[EMPTY_CLASS] < EMPTY_THRESHOLD:

                        masked_probs = probs.clone()
                        masked_probs[EMPTY_CLASS] = 0
                        prediction = masked_probs.argmax().item()
                        d1_pred_list.append(prediction)

                        # Count if we changed from what argmax would give
                        if probs.argmax().item() == EMPTY_CLASS:
                            batch_modified += 1
                    else:

                        d1_pred_list.append(EMPTY_CLASS)

                d1_pred = torch.tensor(d1_pred_list,
device=d1_logit.device)

                total_modified += batch_modified
                total_samples += len(d1_pred_list)

                # Get confidences for the predictions we made
                d1_prob = d1_probs_softmax
                d2_prob = d2_logit.softmax(1)
                d1_conf = d1_prob.gather(1,
d1_pred.unsqueeze(1)).squeeze(1)
                d2_conf = d2_prob.gather(1,
d2_pred.unsqueeze(1)).squeeze(1)

                all_d1_true.extend(d1_true.cpu().numpy())
                all_d2_true.extend(d2_true.cpu().numpy())
                all_d1_pred.extend(d1_pred.cpu().numpy())
                all_d2_pred.extend(d2_pred.cpu().numpy())
```

```
                all_d1_conf.extend(d1_conf.cpu().numpy())
                all_d2_conf.extend(d2_conf.cpu().numpy())

                # Per-class stats — CONFIDENCE-AWARE COMPOSITION
                for i in range(len(d1_true)):
                    # True jersey number
                    true_cls = int(d2_true[i].item() if d1_true[i] == 10
else d1_true[i].item()  10 + d2_true[i].item())

                    # Predicted jersey number (adaptive rule)
                    if (d1_pred[i] != 10 and
                            d1_conf[i] > Config.D1_CONF_THRESHOLD and
                            d2_conf[i] > Config.D2_CONF_THRESHOLD):
                        pred_cls = int(d1_pred[i].item()  10 +
d2_pred[i].item())  # two-digit
                    else:
                        pred_cls = int(d2_pred[i].item())  # fallback to
single-digit

                    class_total[true_cls] += 1
                    if pred_cls == true_cls:
                        class_correct[true_cls] += 1

        print(f"\n Anti-empty-bias heuristic modified
{total_modified}/{total_samples} predictions " +
              f"({100total_modified/total_samples:.1f}%,
threshold={EMPTY_THRESHOLD})")

        # Convert to numpy
        d1_true = np.array(all_d1_true)
        d2_true = np.array(all_d2_true)
        d1_pred = np.array(all_d1_pred)
        d2_pred = np.array(all_d2_pred)
        d1_conf = np.array(all_d1_conf)
        d2_conf = np.array(all_d2_conf)

        # Overall accuracy (confidence-aware)
        overall_correct = 0
        for i in range(len(d1_true)):
```

```
            true_num = int(d2_true[i] if d1_true[i] == 10 else d1_true[i]
10 + d2_true[i])
            if (d1_pred[i] != 10 and
                    d1_conf[i] > Config.D1_CONF_THRESHOLD and
                    d2_conf[i] > Config.D2_CONF_THRESHOLD):
                pred_num = int(d1_pred[i]  10 + d2_pred[i])
            else:
                pred_num = int(d2_pred[i])
            if pred_num == true_num:
                overall_correct += 1

        total = len(d1_true)
        acc = 100.0  overall_correct / total
        d1_acc = 100.0  (d1_true == d1_pred).mean()
        d2_acc = 100.0  (d2_true == d2_pred).mean()
        avg_conf = np.mean((d1_conf + d2_conf) / 2)

        self.results['overall'] = {
            'accuracy': float(acc),
            'd1_accuracy': float(d1_acc),
            'd2_accuracy': float(d2_acc),
            'avg_confidence': float(avg_conf),
            'total_samples': int(total),
            'correct_samples': int(overall_correct),
            'empty_threshold': float(EMPTY_THRESHOLD),
            'predictions_modified': int(total_modified)
        }

        # Per-class
        for cls in sorted(class_total.keys()):
            self.results['per_class'][int(cls)] = {
                'total': class_total[cls],
                'correct': class_correct[cls],
                'accuracy': 100.0  class_correct[cls] / class_total[cls]
            }

        # Confusion matrices (digit-wise only — reconstruction is post-hoc)
        self.results['confusion_matrices'] = {
            'd1': confusion_matrix(d1_true, d1_pred).tolist(),
            'd2': confusion_matrix(d2_true, d2_pred).tolist()
```

```
        }

        self.print_results()
        self.save_results()
        self.generate_visualizations(d1_true, d2_true, d1_pred, d2_pred)

    def print_results(self):
        print("\n" + "="60)
        print("GENERALIZATION TEST RESULTS (with Anti-Empty-Bias
Heuristic)")
        print("="60)
        o = self.results['overall']
        print(f"Overall Accuracy: {o['accuracy']:.2f}%")
        print(f"Tens Digit Acc:   {o['d1_accuracy']:.2f}%")
        print(f"Units Digit Acc:  {o['d2_accuracy']:.2f}%")
        print(f"Avg Confidence:   {o['avg_confidence']:.4f}")
        print(f"Total Samples:    {o['total_samples']}")
        print(f"Predictions Modified:
{o['predictions_modified']}/{o['total_samples']}")

        print("\nPer-Class Accuracy:")
        for cls, stats in sorted(self.results['per_class'].items()):
            print(f"  Class {cls:2d} → {stats['accuracy']:6.2f}%
({stats['correct']}/{stats['total']})")

        if self.results['per_class']:
            best = max(self.results['per_class'].items(), key=lambda x:
x[1]['accuracy'])
            worst = min(self.results['per_class'].items(), key=lambda x:
x[1]['accuracy'])
            print(f"\nBest:  Class {best[0]} → {best[1]['accuracy']:.2f}%")
            print(f"Worst: Class {worst[0]} → {worst[1]['accuracy']:.2f}%")

    def save_results(self):
        path = os.path.join(Config.RESULTS_DIR, 'evaluation_results.json')
        result_data = {
            'metadata': {
                'checkpoint': self.checkpoint_path,
                'test_classes': Config.TEST_CLASSES,
                'timestamp': datetime.now().isoformat(),
```

```
                'empty_threshold':
self.results['overall']['empty_threshold'],
                'reconstruction_rule': f"d1_conf >
{Config.D1_CONF_THRESHOLD} and d2_conf > {Config.D2_CONF_THRESHOLD}"
            },
            'results': self.results
        }
        with open(path, 'w') as f:
            json.dump(result_data, f, indent=4)
        print(f"\nResults saved to {path}")

    def generate_visualizations(self, d1t, d2t, d1p, d2p):
        print("\nGenerating plots...")
        os.makedirs(os.path.join(Config.RESULTS_DIR, 'confusion_matrices'),
exist_ok=True)

        # Confusion matrices
        fig, ax = plt.subplots(1, 2, figsize=(16, 6))
        sns.heatmap(confusion_matrix(d1t, d1p), annot=True, fmt='d',
cmap='Blues', ax=ax[0])
        ax[0].set_title('Tens Digit Confusion')
        sns.heatmap(confusion_matrix(d2t, d2p), annot=True, fmt='d',
cmap='Greens', ax=ax[1])
        ax[1].set_title('Units Digit Confusion')
        plt.tight_layout()
        plt.savefig(os.path.join(Config.RESULTS_DIR, 'confusion_matrices',
'confusion_matrices.png'), dpi=300)
        plt.close()

        # Per-class bar chart
        if self.results['per_class']:
            classes = sorted(self.results['per_class'].keys())
            accs = [self.results['per_class'][c]['accuracy'] for c in
classes]
            plt.figure(figsize=(10, 6))
            bars = plt.bar(classes, accs, color=['green' if a >= 80 else
'orange' if a >= 60 else 'red' for a in accs])
            plt.axhline(y=self.results['overall']['accuracy'], color='red',
linestyle='--',
```

```
                        label=f"Overall:
{self.results['overall']['accuracy']:.2f}%")
            plt.title('Per-Class Accuracy on Test Set (with Anti-Empty-Bias
Heuristic)')
            plt.xlabel('Jersey Number')
            plt.ylabel('Accuracy (%)')
            plt.legend()
            plt.grid(axis='y', alpha=0.3)
            plt.savefig(os.path.join(Config.RESULTS_DIR,
'per_class_accuracy.png'), dpi=300, bbox_inches='tight')
            plt.close()

        print("All plots saved!")


def main():
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument('--checkpoint', type=str,
default='checkpoints/best_model.pth')
    args = parser.parse_args()

    evaluator = Evaluator(args.checkpoint)
    evaluator.evaluate()

    print("\n" + "="60)
    print("EVALUATION COMPLETE!")
    print("="60)


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

This is the evaluation script that tests the trained model on new data to see how well it generalizes.

WHAT THIS SCRIPT DOES:

It loads a trained model and tests it on held-out jersey numbers (numbers the model hasn't seen during training) to evaluate generalization.

MAIN STEPS:

1. Load the trained model:
- Loads checkpoint file saved during training
- Reconstructs the model architecture
- Loads the trained weights
- Prints model info (accuracy from training, parameters, etc.)

2. Prepare test data:
- Uses different jersey numbers than training (from Config.TEST_CLASSES)
- Applies only basic transforms (no augmentation)
- Creates test data loader

3. Run evaluation with special heuristics:

The key innovation here is the "anti-empty-bias" heuristic and confidence-aware reconstruction.

Problem being solved:
The model tends to predict "empty" (10) for the tens digit too often, even for two-digit numbers. This script fixes that.

Anti-empty-bias heuristic:
- When model predicts "empty" for tens digit with low confidence (<55%)
- Ignore that prediction and choose the next most likely digit
- Example: Model predicts "empty" with 50% confidence, "4" with 45% confidence
- Heuristic chooses "4" instead of "empty"
- This fixes many wrong single-digit predictions

Confidence-aware reconstruction:
- Only reconstruct two-digit numbers if both digits have high confidence
- Uses thresholds: D1 confidence > 0.10, D2 confidence > 0.50
- Example: Model predicts "4" (confidence: 0.15) and "8" (confidence: 0.60)
- Both above thresholds → reconstructed number: 48
- If either digit has low confidence → use only units digit as single-digit number

4. Calculate multiple accuracy metrics:
- Overall accuracy: Full jersey number correct (using reconstruction rules)
- D1 accuracy: Tens digit correct (or correctly "empty")
- D2 accuracy: Units digit correct
- Per-class accuracy: Accuracy for each jersey number separately
- Average confidence: How confident the model is in its predictions

5. Generate visualizations:
- Confusion matrices: Show which digits get confused with which

  - One for tens digit
  - One for units digit
- Per-class bar chart: Shows accuracy for each jersey number
  - Green bars: Good accuracy (≥80%)
  - Orange bars: Medium accuracy (60-80%)
  - Red bars: Poor accuracy (<60%)

6. Save results:
- Saves all metrics to JSON file
- Saves plots to results folder
- Includes metadata about the evaluation

KEY FEATURES:

Handling model outputs:
The script safely handles whether model returns 2 outputs (just logits) or 3 outputs (logits + attention weights)

Per-class statistics:
Tracks accuracy for each jersey number separately to identify:
- Which numbers are easy/hard
- Patterns in errors

Digit-level vs number-level accuracy:
- D1/D2 accuracy: Individual digit prediction
- Overall accuracy: Full number reconstructed correctly
- Jersey number accuracy: Using reconstruction rules

PRACTICAL EXAMPLE:

Suppose test data has jersey number "48" (not in training, but digits "4" and "8" are in training numbers).

Without heuristics:
- Model might predict: "empty" (10) for tens, "8" for units
- Reconstructed as: "8" (wrong)
- Reason: Model is biased toward predicting "empty" for tens digit

With heuristics:
- Model predicts: "empty" with 50% confidence, "4" with 45% confidence, "8" with 85% confidence
- Anti-empty-bias: Choose "4" (not "empty") because empty confidence <55%
- Confidence check: D1 confidence=0.45 > 0.10 ✓, D2 confidence=0.85 > 0.50 ✓
- Reconstructed as: "48" ✓

OUTPUT FORMAT:

Console output shows:
- Overall accuracy percentage
- Individual digit accuracies
- Per-class accuracy breakdown
- Best/worst performing numbers
- How many predictions were modified by heuristics

Saved files:
- results/evaluation_results.json (all metrics)
- results/confusion_matrices/confusion_matrices.png
- results/per_class_accuracy.png

WHY THIS MATTERS:

This evaluation tests generalization:
- Train on numbers: 4, 6, 8, 9, 49, 66, 89
- Test on numbers: 48, 64, 88
- All test digits (4, 6, 8) appear in training, just in different combinations
- Tests if model learned to recognize individual digits, not just memorized specific numbers

COMMAND LINE USE:
```
python evaluate.py --checkpoint checkpoints/best_model.pth
```

The script provides a comprehensive evaluation with diagnostic tools (confusion matrices, per-class breakdown) to understand exactly how the model performs and where it makes errors.

analyze_weights.py:

```python
# analyze_weights.py
import torch
import json
import os
from config import Config
from model import JerseyTemporalNet

def analyze_model_weights(checkpoint_path):
    """Generate comprehensive weight summary"""
```

```
    print("="70)
    print("TRAINED WEIGHTS SUMMARY")
    print("="70)

    # Load checkpoint
    checkpoint = torch.load(checkpoint_path, map_location='cpu')

    print(f"\n Checkpoint: {checkpoint_path}")
    print(f" Best Validation Accuracy: {checkpoint.get('best_val_acc',
'N/A'):.2f}%")
    print(f" Training Epoch: {checkpoint.get('epoch', 'N/A')}")

    # Create model
    model = JerseyTemporalNet(
        num_classes=Config.NUM_DIGIT_CLASSES,
        hidden_dim=Config.HIDDEN_DIM,
        dropout=Config.DROPOUT,
        backbone=Config.BACKBONE,
        use_spatial_attention=Config.USE_SPATIAL_ATTENTION,
        use_temporal_attention=Config.USE_TEMPORAL_ATTENTION,
        bidirectional=Config.BIDIRECTIONAL_LSTM
    )

    model.load_state_dict(checkpoint['model_state_dict'])

    # Analyze weights by component
    print("\n" + "="70)
    print("MODEL ARCHITECTURE SUMMARY")
    print("="70)

    components = {
        'backbone': [],
        'spatial_attention': [],
        'lstm': [],
        'temporal_attention': [],
        'head_digit1': [],
        'head_digit2': [],
        'other': []
    }
```

```
    # Categorize parameters
    for name, param in model.named_parameters():
        if 'backbone' in name:
            components['backbone'].append((name, param))
        elif 'spatial_attention' in name:
            components['spatial_attention'].append((name, param))
        elif 'lstm' in name:
            components['lstm'].append((name, param))
        elif 'temporal_attention' in name:
            components['temporal_attention'].append((name, param))
        elif 'head_digit1' in name:
            components['head_digit1'].append((name, param))
        elif 'head_digit2' in name:
            components['head_digit2'].append((name, param))
        else:
            components['other'].append((name, param))

    # Print summary for each component
    total_params = 0
    trainable_params = 0

    for component_name, params in components.items():
        if not params:
            continue

        component_total = sum(p.numel() for _, p in params)
        component_trainable = sum(p.numel() for _, p in params if
p.requires_grad)

        total_params += component_total
        trainable_params += component_trainable

        print(f"\n🔹 {component_name.upper().replace('_', ' ')}")
        print(f"   Total Parameters: {component_total:,}")
        print(f"   Trainable: {component_trainable:,}
({100component_trainable/component_total:.1f}%)")
        print(f"   Layers: {len(params)}")

        # Show first 3 layers
        for i, (name, param) in enumerate(params[:3]):
```

```
            print(f"      {name}: {list(param.shape)}")
        if len(params) > 3:
            print(f"      ... ({len(params)-3} more layers)")

    print("\n" + "="70)
    print("OVERALL STATISTICS")
    print("="70)
    print(f"Total Parameters: {total_params:,}")
    print(f"Trainable Parameters: {trainable_params:,}
({100trainable_params/total_params:.1f}%)")
    print(f"Frozen Parameters: {total_params - trainable_params:,}")
    print(f"Model Size: {total_params  4 / (10242):.2f} MB (FP32)")

    # Analyze digit heads specifically
    print("\n" + "="70)
    print("DIGIT HEAD ANALYSIS (KEY FOR GENERALIZATION)")
    print("="70)

    for head_name in ['head_digit1', 'head_digit2']:
        print(f"\n {head_name.upper()} (Tens Digit)" if '1' in head_name
else f"\n🎯 {head_name.upper()} (Units Digit)")
        head_params = components[head_name]

        for name, param in head_params:
            if 'fc_out.weight' in name:
                print(f"   Output Layer Shape: {list(param.shape)}")
                print(f"   → Maps {param.shape[1]} features →
{param.shape[0]} digit classes")
                print("   Weight Statistics:")
                print(f"      Mean: {param.data.mean().item():.4f}")
                print(f"      Std:  {param.data.std().item():.4f}")
                print(f"      Min:  {param.data.min().item():.4f}")
                print(f"      Max:  {param.data.max().item():.4f}")

    # Save summary to JSON
    summary = {
        'checkpoint_path': checkpoint_path,
        'best_val_acc': float(checkpoint.get('best_val_acc', 0)),
        'epoch': int(checkpoint.get('epoch', 0)),
        'total_parameters': int(total_params),
```

```
        'trainable_parameters': int(trainable_params),
        'model_size_mb': float(total_params  4 / (10242)),
        'components': {
            name: {
                'total_params': int(sum(p.numel() for _, p in params)),
                'trainable_params': int(sum(p.numel() for _, p in params if
p.requires_grad)),
                'num_layers': len(params)
            }
            for name, params in components.items() if params
        }
    }

    output_path = os.path.join(Config.RESULTS_DIR, 'weight_summary.json')
    with open(output_path, 'w') as f:
        json.dump(summary, f, indent=4)

    print(f"\n Summary saved to: {output_path}")
    print("="70)

if __name__ == "__main__":
    analyze_model_weights('checkpoints/best_model.pth')
```

This script analyzes the trained model's weights to understand how the model learned and what it learned.

WHAT THIS SCRIPT DOES:

It loads a trained model checkpoint and provides a detailed analysis of the model's internal weights - showing which parts of the model were most important, how many parameters were trained, and giving insights into what the model learned.

MAIN ANALYSIS STEPS:

1. Load the checkpoint:
- Loads the saved model file
- Shows training accuracy and epoch number

2. Create the model architecture:
- Recreates the exact same model structure

- Loads the trained weights into it

3. Categorize all parameters by component:
Groups weights into 7 categories:
- backbone: ResNet CNN layers (extract visual features)
- spatial_attention: Layers that focus on jersey number region
- lstm: Temporal modeling layers
- temporal_attention: Layers that weight different frames
- head_digit1: Classifier for tens digit
- head_digit2: Classifier for units digit
- other: Any remaining layers

4. Generate detailed statistics for each component:
For each component shows:
- Total parameters: How many numbers/weights
- Trainable parameters: How many were actually updated during training
- Percentage trainable: How much of this component was trained vs frozen
- Number of layers: How many distinct layers
- Example layer shapes: Shows the size of first few layers

5. Special focus on digit classification heads:
This is the most important analysis for understanding generalization:

Digit head output layer analysis:
- Shows weight matrix shape: [11 classes × number of features]
- For head_digit1 (tens digit): Maps features → 11 classes (0-9 + "empty")
- For head_digit2 (units digit): Maps features → 11 classes (0-9 + "empty")
- Shows weight statistics: mean, standard deviation, min, max

Why this matters:
The digit heads are what actually make the digit predictions. Their weights show:
- Which features are important for each digit
- How confident/uncertain the model is
- Whether it learned clear patterns or is confused

6. Calculate overall statistics:
- Total parameters: All weights in the model
- Trainable parameters: Weights that were updated
- Frozen parameters: Weights kept fixed (from pretrained ResNet)
- Model size: How much memory/disk space needed

7. Save results to JSON file:
- Saves all analysis to a file for future reference
- Includes component breakdown

- Includes training accuracy

KEY INSIGHTS YOU GET FROM THIS ANALYSIS:

1. Understanding what was trained:
- Example: If backbone has 90% frozen, only 10% was fine-tuned
- This tells you if the model reused existing visual knowledge vs learned everything from scratch

2. Parameter distribution:
- Which parts of model have most parameters
- Example: Backbone might have millions of parameters, heads only thousands
- Shows where model complexity is concentrated

3. Training efficiency:
- Percentage of trainable parameters shows training efficiency
- Low percentage (e.g., 20% trainable) means leveraged pretrained knowledge well

4. Digit head weight patterns:
- Weight magnitude: Large weights → strong feature importance
- Weight spread: High standard deviation → specialized features for different digits
- Extreme values: Very high/low weights → confident decisions

EXAMPLE OUTPUT INTERPRETATION:

If head_digit2 output weights show:
- Mean: 0.001 (small weights overall)
- Std: 0.050 (moderate variation)
- Min: -0.100, Max: 0.150 (reasonable range)

This suggests:
- Model doesn't have extreme confidence
- Learned moderate distinctions between digits
- Should generalize reasonably

If instead weights showed:
- Mean: 0.500 (very large)
- Std: 2.000 (huge variation)
- Min: -5.000, Max: 8.000 (extreme values)

This would suggest:
- Overconfident model
- Possibly overfitted to training data
- Might not generalize well

PRACTICAL USE CASES:

1. Model debugging:
- If accuracy is poor, check if backbone was properly unfrozen
- Check if digit heads have reasonable weight ranges

2. Model comparison:
- Compare different checkpoints to see training progress
- See which components changed most during training

3. Optimization:
- Identify components with most parameters for potential pruning
- See if certain heads could be made smaller without losing accuracy

4. Understanding generalization:
- Well-trained digit heads should show clear patterns
- Confused heads might show similar weights for all digits

TECHNICAL DETAILS:

Why analyze weights?
- Weights encode what the model learned
- Pattern in weights = pattern in learning
- Can reveal overfitting, underfitting, or good generalization

Model size calculation:
- Each parameter is a 32-bit float (4 bytes)
- Total parameters × 4 bytes = size in bytes
- Divided by 1024^2 to get MB

Frozen vs trainable:
- Frozen: From pretrained ResNet, kept unchanged
- Trainable: Updated during jersey number training
- Typically: Backbone mostly frozen, heads fully trainable

COMMAND LINE USE:
Just run: `python analyze_weights.py`

The script provides a "health check" for the trained model, showing what was learned, how complex the model is, and giving hints about whether it will generalize well to new data.