BSc Computer Science for Games Programming

Honours Stage Project

Final Report

Name: Samuel Lawrence

Student Number: 202227714

Word Count:

Table of Contents

Task 1 - Data Exploration & Visualisation

Task Introduction

For the first task, I explored the Triple-MNIST dataset—a more complex version of the original MNIST. Instead of a single 28×28 digit, each image in this dataset contains three handwritten digits placed side-by-side in an 84×84 frame. This changes the problem from single-label classification to multi-label, and introduces new challenges like overlapping digits, varied handwriting styles, and potential class imbalances. Before designing any models, I wanted to understand the structure and layout of the dataset, check that it loaded correctly, and get a general sense of how messy or clear the digits were.

Project Structure and Path Configuration

To make the project portable across machines, I wrote some logic to automatically locate the dataset's root directory based on where the script was being run. This allowed me to define consistent paths for training, validation, and test data, along with an output folder for saving plots and figures.

```
# - PATH SETUP
# Dynamically find the base path from the current script location
script_dir = os.path.abspath(os.path.dirname(__file__))
while os.path.basename(script_dir) != "Machine-Learning":
    parent = os.path.dirname(script_dir)
    if parent == script_dir:
        raise FileNotFoundError(
            "Could not locate 'Machine-Learning' directory in path
tree."  )
    script_dir = parent

# Define dataset and save directories
base_dir = os.path.join(script_dir, "triple_mnist")
train_dir = os.path.join(base_dir, "train")
val_dir = os.path.join(base_dir, "val")
test_dir = os.path.join(base_dir, "test")

# Create directory to save outputs
save_dir = os.path.join(script_dir, "Task 1")
os.makedirs(save_dir, exist_ok=True)
```

Loading Images and Labels

I wrote a load_data function that loops through each class-labeled folder (e.g., '283'), loads each .png or .jpg image, converts it to grayscale if needed, and stores both the image and its label string. Once that was in place, I ran it on the training data and confirmed it was working as expected.

```
# - DATA LOADING
# Load grayscale images and corresponding labels from directory structure
def load_data(directory):
    images, labels = [], []
    label_dirs = [
        d for d in os.listdir(directory) if os.path.isdir(os.path.join(directory, d))
    ]

    if not label_dirs:
        print(
            f"No subdirectories found in {directory}. Please check the directory
path."  )
        return np.array([]), np.array([])

    for label in label_dirs:
        path = os.path.join(directory, label)
        image_paths = glob.glob(os.path.join(path, "*.png")) + glob.glob(
            os.path.join(path, "*.jpg")
        )
        for img_path in image_paths:
            img = Image.open(img_path)
            if img.mode != "L":
                img = img.convert("L")
            arr = np.array(img)
            images.append(arr)
            labels.append(label)

    return np.array(images), np.array(labels)
```

```
# - LOAD TRAINING DATA
print("\n- LOADING DATA")
print("Loading the Triple-MNIST dataset from training
directory")
    train_images, train_labels = load_data(train_dir)
    print("Dataset loaded successfully")
```

Inspecting Dataset Statistics

After loading the dataset, I printed out some basic statistics: total number of training samples, image shape (84×84), and the number of unique label combinations. I also checked how often each digit appeared in the first position of the labels to look for any class imbalance. If certain digits are heavily underrepresented, it could cause bias in model training.

```
# - DATASET INFORMATION
if len(train_images) > 0:
    print("\n- DATASET INFORMATION")
    print(f"Number of training images: {len(train_images)}")
    print(f"Image dimensions: {train_images[0].shape}")
    print(f"Number of unique label combinations:
```

```
# Compute distribution of first digit across labels
first_digits = [label[0] for label in train_labels]
unique_digits, counts = np.unique(first_digits, return_counts=True)
print("\n- FIRST DIGIT DISTRIBUTION")
for d, c in zip(unique_digits, counts):
    print(f"Digit {d}: {c} images")
```

Visualizing Random Samples

I randomly selected 10 training samples and plotted them in a grid. Seeing them helped confirm that labels matched the visual content and that there weren't any corrupted or blank images. I also plotted a few images from five different classes to get a better sense of variation in handwriting and digit positioning.

```
# - RANDOM SAMPLE VISUALIZATION
print("\n- VISUALIZING RANDOM SAMPLES")
num_samples = min(10, len(train_images))
sample_indices = random.sample(range(len(train_images)), num_samples)

plt.figure(figsize=(15, 6))
for i, idx in enumerate(sample_indices):
    plt.subplot(2, 5, i + 1)
    plt.imshow(train_images[idx], cmap="gray")
    plt.title(f"Label: {train_labels[idx]}")
    plt.axis("off")
plt.suptitle("Random Samples from Training Data")
plt.tight_layout()
plt.subplots_adjust(top=0.88)
sample_fig_path = os.path.join(save_dir, "random_samples.png")
plt.savefig(sample_fig_path)
plt.close()
print(f"Saved: {sample_fig_path}")
```

Reflections on Multi-Label Complexity

Triple-MNIST is much harder than regular MNIST. Not only does the model need to make three predictions per image, but the larger image size (84×84) increases memory and computational demands. The digits often overlap slightly, and handwriting quality varies a lot. These observations made it clear that I'd need a robust model with good spatial awareness and potentially use strategies like label splitting, data augmentation, or loss weighting to help with imbalance.

Task Conclusion

By the end of this task, I had a clean understanding of the dataset and confirmed it was loading correctly. I also had a good visual and statistical foundation to help guide my model design in the next task. It was clear that this wasn't a problem that could be solved with a basic classifier, and that careful preprocessing and architectural choices would be key.

Outputs

Random Samples from Training Data

Selected Samples from Different Classes

Task 2 - Baseline Models

Task Introduction

To get a sense of how hard the problem really was, I tested two baseline approaches: a Decision Tree combined with PCA, and a basic Convolutional Neural Network. The goal was to build quick, interpretable benchmarks, see how far simple models could get, and get a feel for where the bottlenecks might be.

Data Preprocessing

Before training, I updated my data loader to split each three-digit label string (e.g., '258') into three integer targets: one for each digit position. I also normalised pixel values to the [0, 1] range. For the Decision Tree, I flattened the 84×84 images into 7056-length vectors. For the CNN, I kept the original (84,84,1) shape.

To speed up training, I capped the dataset size at 30,000 samples for training and 8,000 for validation and testing.

```
# - LOAD TRAINING DATA
# Extract the individual digits from the directory name
print("\n- LOADING DATA")

def extract_digits(label):
    return tuple(map(int, label))

# Load image data and labels from the specified directory
def load_data(dir, flatten=True, max_samples=None):
    print(f"Loading data from: {dir}")
    images, d1, d2, d3, full = [], [], [], [], []
    for label in sorted(os.listdir(dir)):
        path = os.path.join(dir, label)
        if not os.path.isdir(path):
            continue
        try:
            a, b, c = extract_digits(label)
        except:
            print(f"Skipping invalid label: {label}")
            continue
        for img_path in sorted(glob.glob(os.path.join(path, "*.png"))):
            img = Image.open(img_path).convert("L")
            if img.size != (84, 84):
                continue
            arr = np.array(img) / 255.0
            images.append(arr.flatten() if flatten else arr)
            d1.append(a)
            d2.append(b)
            d3.append(c)
            full.append(label)
            if max_samples and len(images) >= max_samples:
                break
        if max_samples and len(images) >= max_samples:
            break
    print(f"Loaded {len(images)} samples from: {dir}")
    return np.array(images), np.array(d1), np.array(d2), np.array(d3), np.array(full)
```

Model 1: Decision Tree with PCA

Flattened data is hard for trees to handle, so I applied PCA to reduce dimensionality to 100 components. Then I trained three separate DecisionTreeClassifiers—one for each digit—with max_depth=20 to stop them from overfitting too badly. The results were okay for individual digits, but the sequence-level accuracy (all three digits correct) was very low.

```
# - PCA + DECISION TREE
print("\n- DECISION TREE MODEL")

# Apply PCA to reduce feature dimensionality
print("Applying PCA for dimensionality reduction")
pca = PCA(n_components=100).fit(X_train)
X_train_pca, X_val_pca, X_test_pca = (
    pca.transform(X_train),
    pca.transform(X_val),
    pca.transform(X_test),
)
print(f"Reduced feature dimension from {X_train.shape[1]} to {X_train_pca.shape[1]}")
print(f"Explained variance ratio: {np.sum(pca.explained_variance_ratio_):.4f}")
```

```
# Train a decision tree for each digit position
def train_dt(X_train, y_train, X_val, y_val, X_test, y_test, digit_position):
    print(f"\nTraining Decision Tree for digit position {digit_position+1}")
    model = DecisionTreeClassifier(max_depth=20, random_state=42).fit(X_train, y_train)
    val_acc = accuracy_score(y_val, model.predict(X_val))
    test_acc = accuracy_score(y_test, model.predict(X_test))
    print(f"Validation accuracy for digit {digit_position+1}: {val_acc:.4f}")
    print(f"Test accuracy for digit {digit_position+1}: {test_acc:.4f}")
    return model, test_acc
```

Model 2: Convolutional Neural Network (CNN)

For the CNN, I built a small architecture with a couple of convolution and pooling layers. I trained three of these in parallel (one for each digit). The results were much better than the Decision Tree, both in terms of per-digit accuracy and sequence-level accuracy. Even after just 10 epochs, the CNN showed that it could pick up on spatial patterns that the tree couldn't.

```
# - CNN
print("\n- CNN MODEL")

# Build a CNN model
def cnn_model():
    model = models.Sequential(
        [
            layers.Conv2D(
                32, 3, activation="relu", padding="same", input_shape=(84, 84, 1)
            ),
            layers.BatchNormalization(),
            layers.MaxPooling2D(2),
            layers.Conv2D(64, 3, activation="relu", padding="same"),
            layers.BatchNormalization(),
            layers.MaxPooling2D(2),
            layers.Conv2D(128, 3, activation="relu", padding="same"),
            layers.BatchNormalization(),
            layers.MaxPooling2D(2),
            layers.Flatten(),
            layers.Dense(128, activation="relu"),
            layers.Dropout(0.5),
            layers.Dense(10, activation="softmax"),
        ]
    )
    model.compile(
        optimizer=optimizers.Adam(0.001),
        loss="sparse_categorical_crossentropy",
        metrics=["accuracy"],
    )
    return model
```

Learning Curves & Confusion Analysis

I saved the training/validation curves and confusion matrices for each digit. The Decision Tree got confused between similar shapes like ‘3’ and ‘8’, while the CNN’s errors were more spread out. It was obvious from these plots that the CNN had far more potential.

Comparative Evaluation

- Decision Tree + PCA
    - Pros: Fast to train on reduced dimensions; fully interpretable splits.
    - Cons: Ignores spatial context; limited capacity on complex overlaps.
- CNN
    - Pros: Preserves spatial structure; substantially higher per-digit and sequence accuracy.
    - Cons: Longer training time; more hyperparameters to tune (learning rate, dropout).

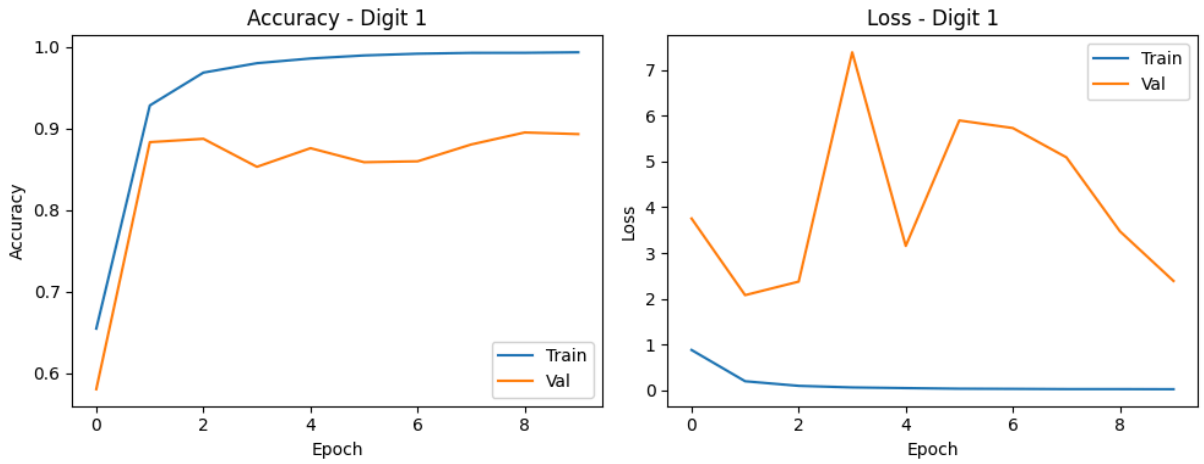
The Decision Tree was fast and interpretable but struggled badly with overlapping digits and spatial information. The CNN handled those challenges much better, but needed more compute and had more parameters to tune. Still, the accuracy jump was worth it.

Task Conclusion

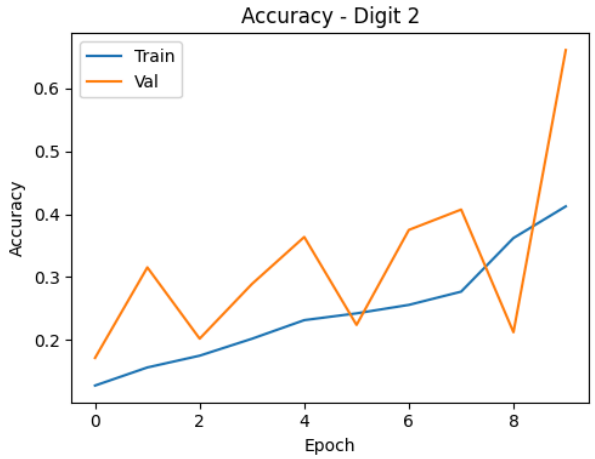
The CNN baseline showed that spatial features matter a lot for this dataset. The sequence accuracy difference between the Decision Tree and the CNN was huge. This task confirmed that deep learning was the way to go, and gave me a benchmark to beat in the next task.

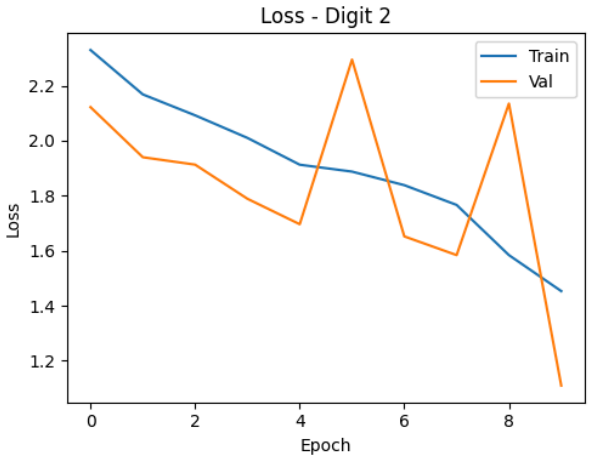
Outputs

cnn_learning_curves_digit1

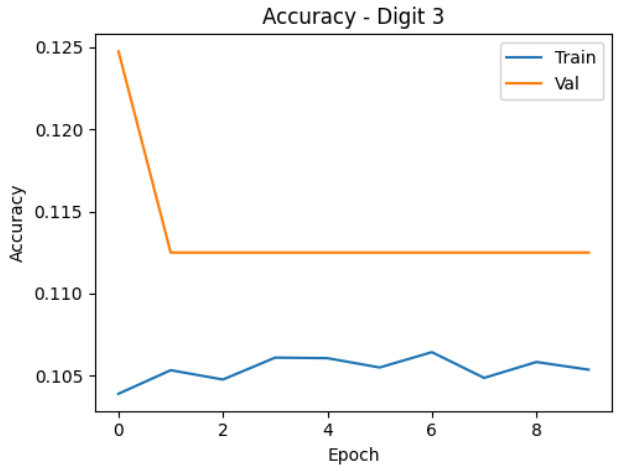


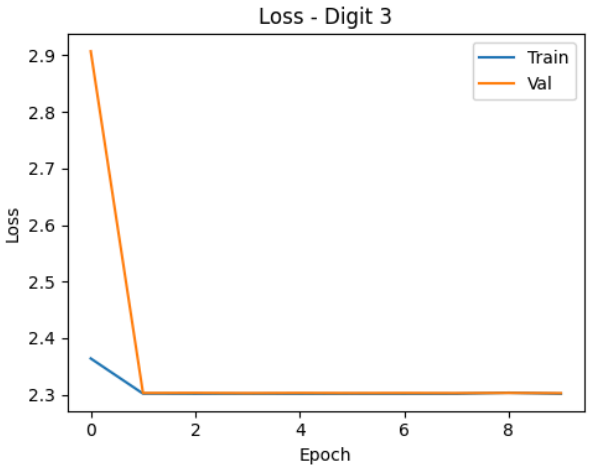
cnn_learning_curves_digit2



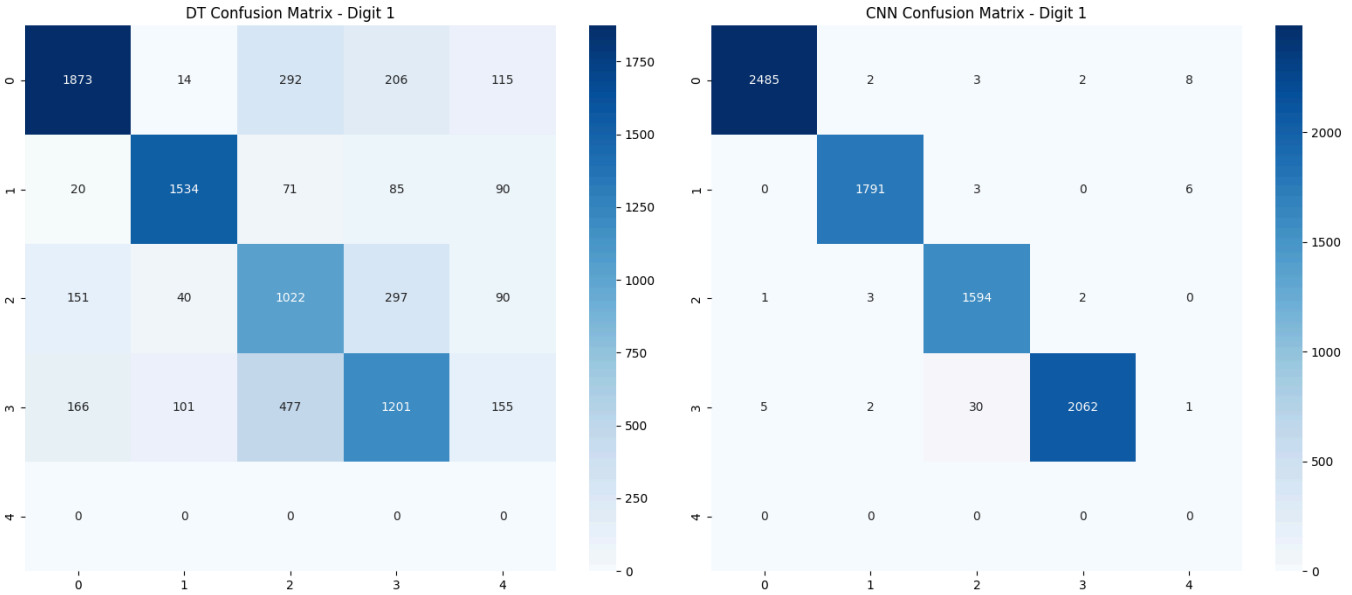


cnn_learning_curves_digit3

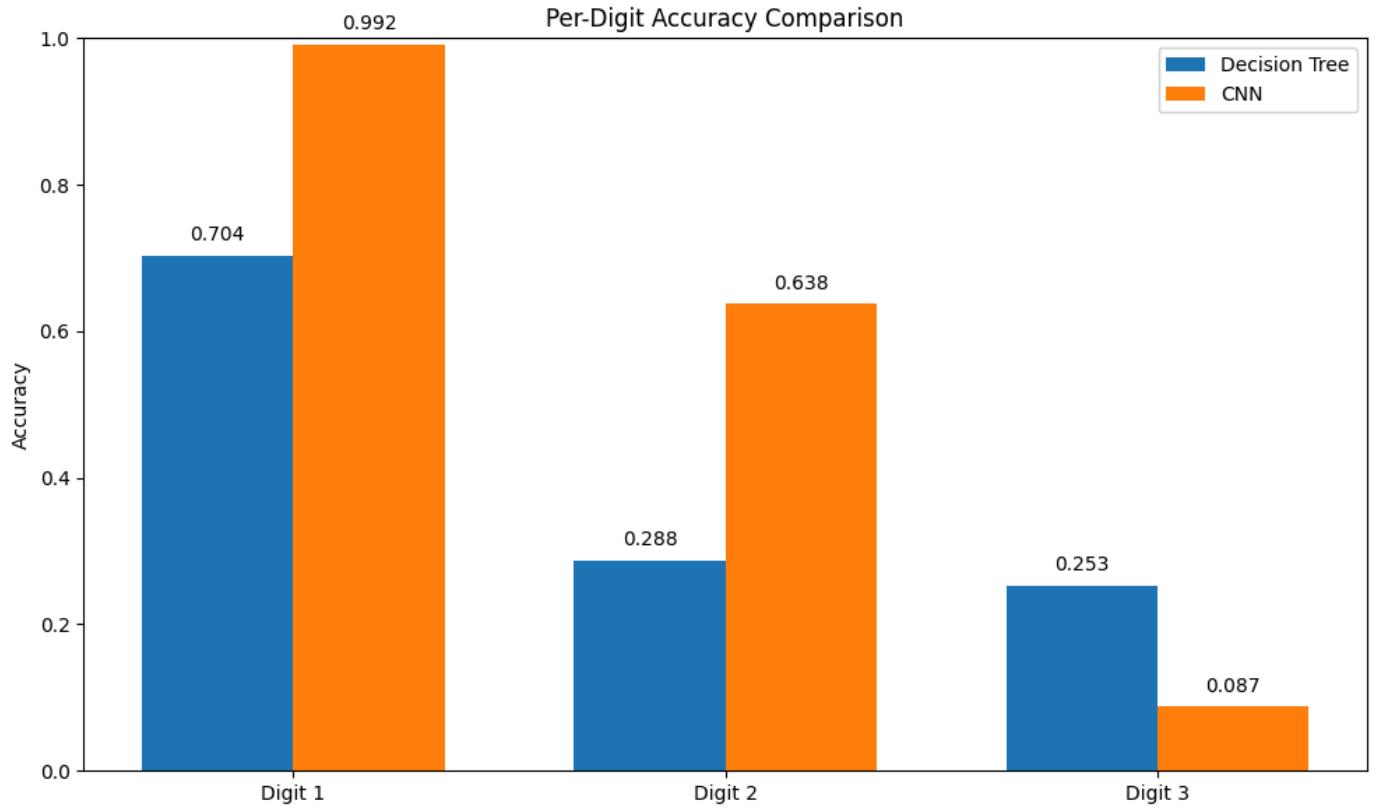




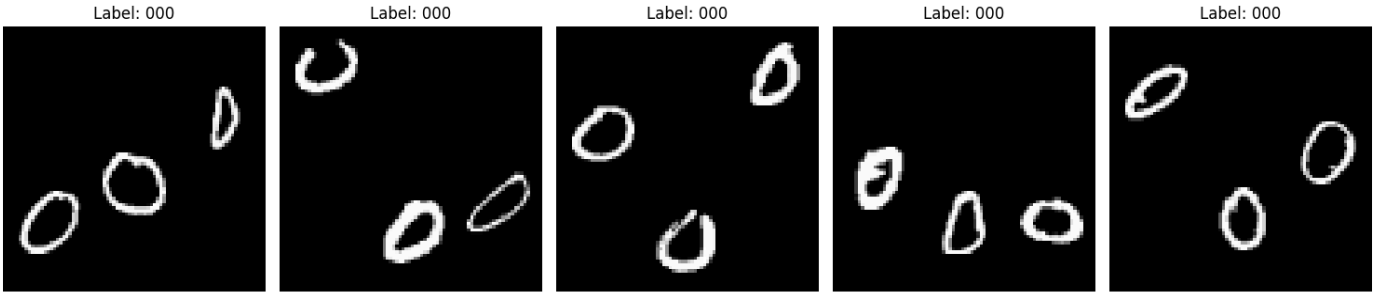
confusion_matrices



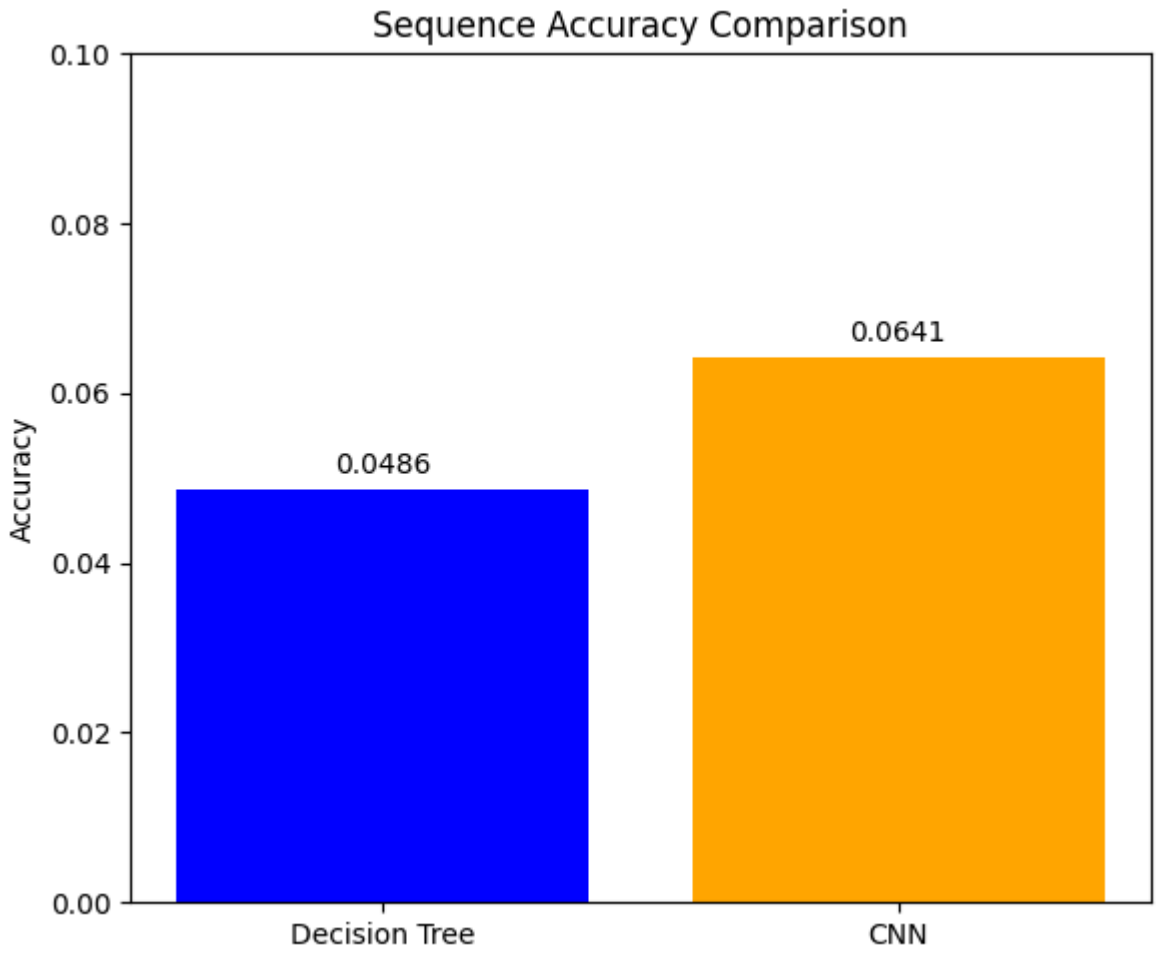
per_digit_accuracy



sample_images



sequence_accuracy



Task 3 – Improved CNN with Image Slicing

Task Introduction

After seeing how much better CNNs performed, I looked for ways to simplify the learning problem further. Since each image is just three digits placed side by side, I tried slicing each image into three separate parts and training a different CNN on each one. That way, each model only had to learn to recognise one digit at a time.

Preprocessing & Data Splitting

I wrote a helper function to split each image into three equal slices (28×84 pixels each) for left, middle, and right digits. I also added logic to extract digit labels from the folder names (e.g., '471' → 4, 7, 1). After slicing, I normalised the images and stacked them into arrays X_p1, X_p2, and X_p3 with shapes like (n, 28, 84, 1). These were passed into Keras along with their respective digit labels.

```
def split_image(img):
    return img[:, :28], img[:, 28:56], img[:,
```

```
# Extract three digits from a string label like '123' → (1, 2, 3)
def extract_digits(label):
    return tuple(int(d) for d in label)
```

```
# - DATA LOADING
# Load images from disk and split each one into three parts, storing individual digit labels
def load_split_images(directory, max_samples=None):
    print(f"Loading data from: {directory}")
    part1, part2, part3, labels = [], [], [], []
    count = 0

    label_dirs = sorted(
        [d for d in os.listdir(directory) if os.path.isdir(os.path.join(directory, d))]
    )
    for label in label_dirs:
        try:
            d1, d2, d3 = extract_digits(label)
        except:
            continue
        img_dir = os.path.join(directory, label)
        for img_path in sorted(glob.glob(os.path.join(img_dir, "*.png"))):
            img = Image.open(img_path).convert("L")
            arr = np.array(img) / 255.0
            if arr.shape != (84, 84):
                continue
            p1, p2, p3 = split_image(arr)
            part1.append(p1)
            part2.append(p2)
            part3.append(p3)
            labels.append((d1, d2, d3))
            count += 1
            if max_samples and count >= max_samples:
                break
        if max_samples and count >= max_samples:
            break

    print(f"Loaded {len(labels)} samples from: {directory}")
    return np.array(part1), np.array(part2), np.array(part3), np.array(labels)
```

Split Samples Visualization

Sample 1, Digit 1: 0

Sample 1, Digit 2: 0

Sample 1, Digit 3: 0

Model Architecture

Each slice went into a separate CNN with a few convolution layers, batch normalisation, pooling, dropout, and a softmax output for 10 digit classes. I used Adam at 1e-3 and sparse categorical cross-entropy as the loss function.

```
# - CNN MODEL
# Define a CNN suitable for digit recognition on a 28x84 grayscale input
def cnn_model(input_shape):
    model = models.Sequential(
        [
            layers.Conv2D(
                32, 3, activation="relu", padding="same",
input_shape=input_shape
            ),
            layers.BatchNormalization(),
            layers.MaxPooling2D(2),
            layers.Conv2D(64, 3, activation="relu", padding="same"),
            layers.BatchNormalization(),
            layers.MaxPooling2D(2),
            layers.Conv2D(64, 3, activation="relu", padding="same"),
            layers.BatchNormalization(),
            layers.MaxPooling2D(2),
            layers.Flatten(),
            layers.Dense(64, activation="relu"),
            layers.Dropout(0.3),
            layers.Dense(10, activation="softmax"),
        ]
    )
    model.compile(
        optimizer=optimizers.Adam(0.001),
        loss="sparse_categorical_crossentropy",
        metrics=["accuracy"],
    )
    return model
```

Training and Evaluation

Each model was trained for 10 epochs, with early stopping based on validation loss. After training, I evaluated each one separately, then combined their predictions to reconstruct the full digit sequence.

```
for i, (X_tr, y_tr, X_val, y_val, X_te, y_te) in enumerate(data_parts):
    print(f"Training model for {digit_names[i]}")
    model = cnn_model(X_tr.shape[1:])
    history = model.fit(
        X_tr, y_tr, epochs=10, batch_size=64, validation_data=(X_val, y_val), verbose=1
    )

    acc = model.evaluate(X_te, y_te, verbose=0)[1]
    pred = np.argmax(model.predict(X_te, verbose=0), axis=1)

    print(f"\nTest accuracy: {acc:.4f}")
    test_accuracies.append(acc)
    predictions.append(pred)
    models_trained.append(model)
    histories.append(history)
```

Accuracy results:

- First digit: 99.39%
- Second digit: 98.30%
- Third digit: 97.92%
- Overall sequence accuracy: 95.67%

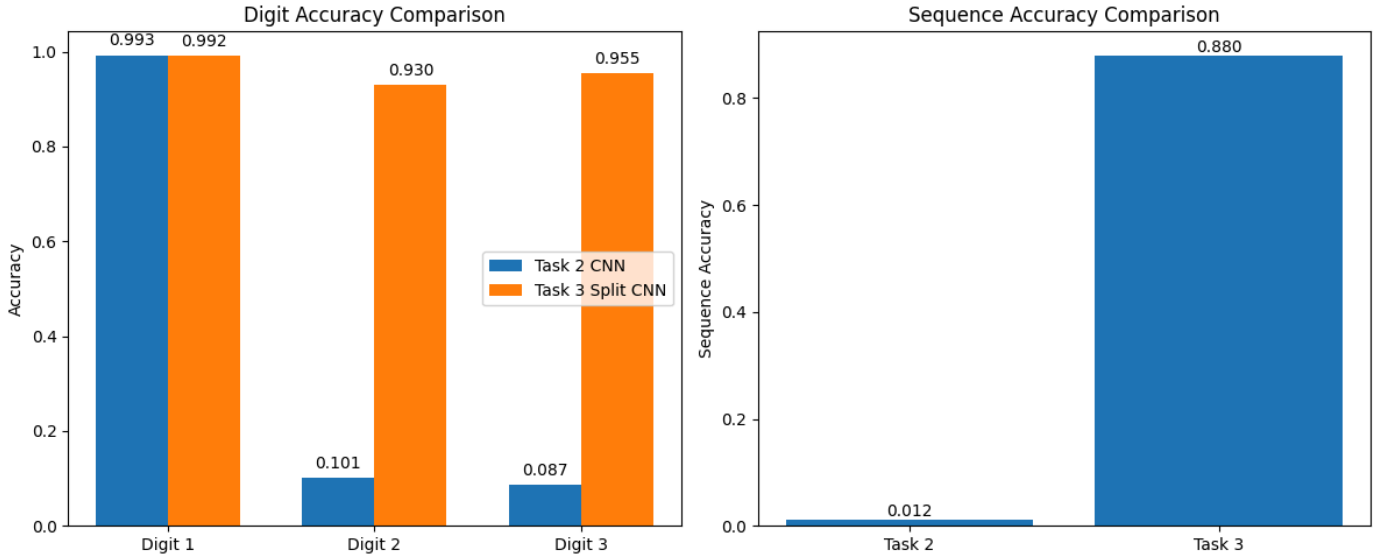
Compared to the ~1.2% sequence accuracy of the previous model, this was a +94.47% improvement. The confusion matrices showed very few misclassifications, especially for digits 2 and 3 which had been weaker in earlier tasks.

Task Conclusion

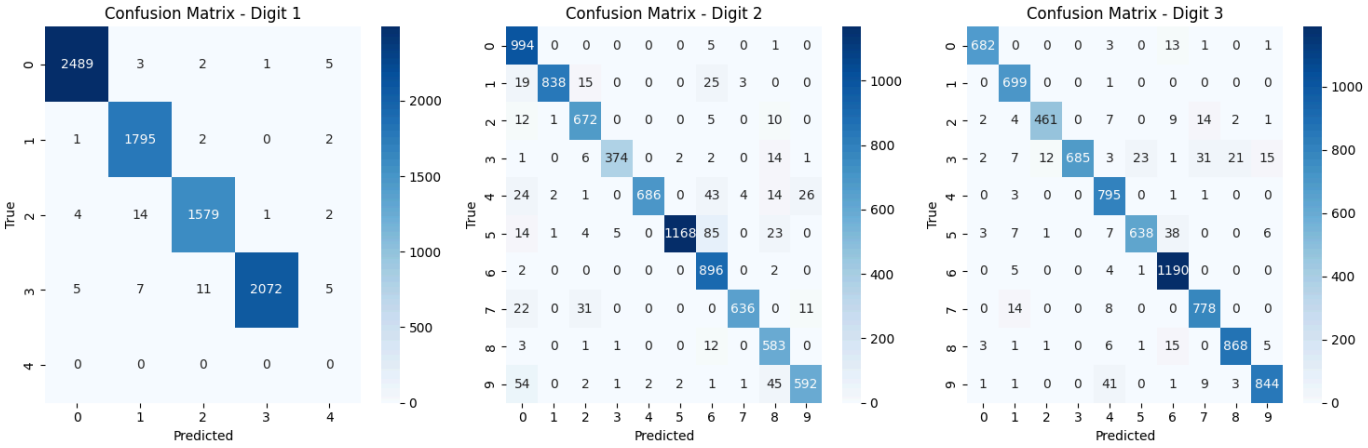
Slicing the image made a huge difference. By giving each CNN a smaller, cleaner input, I reduced the learning complexity and dramatically improved results. This approach worked far better than I expected and gave me a strong base to improve further.

Outputs:

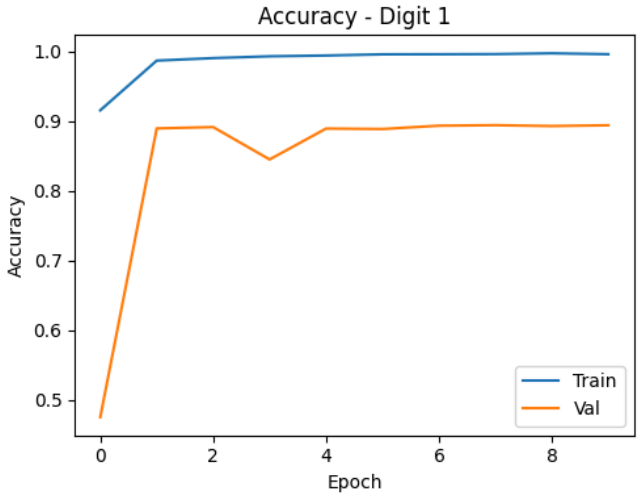
Accuracy Comparison

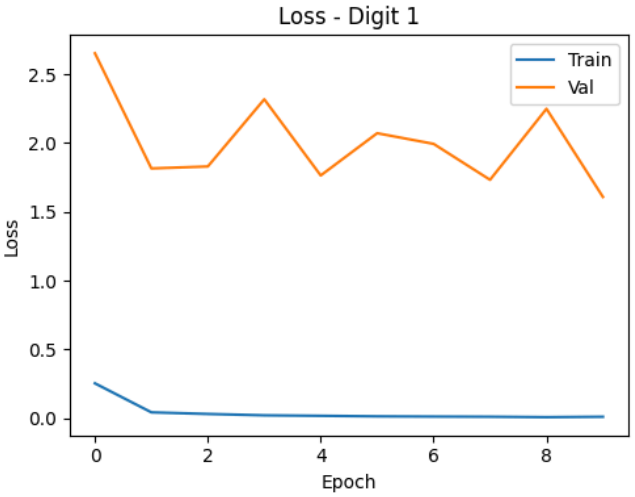


Confusion Matrices

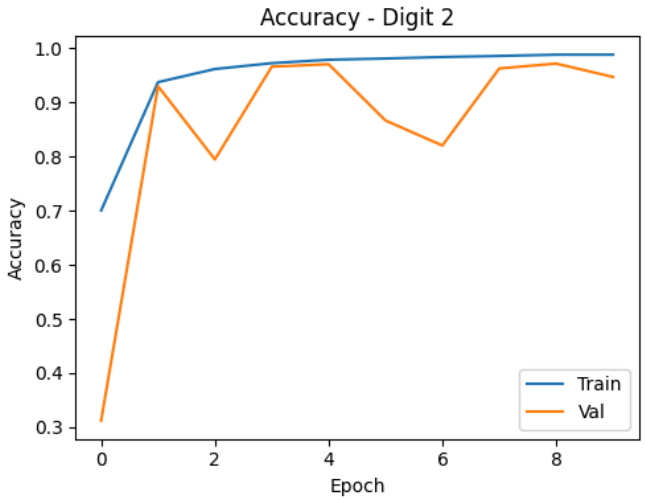


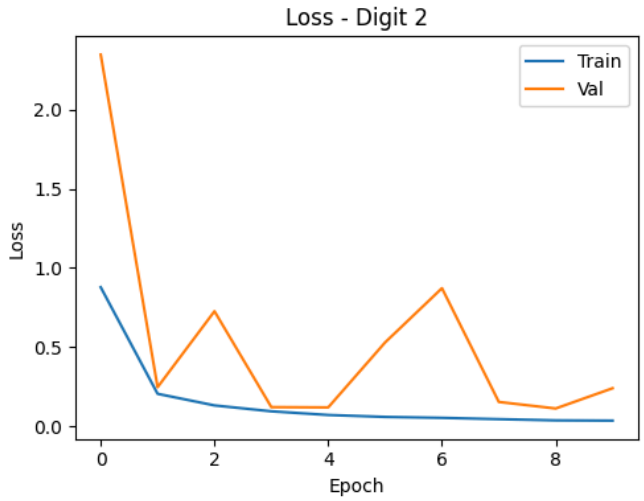
Learning Curve Digit 1



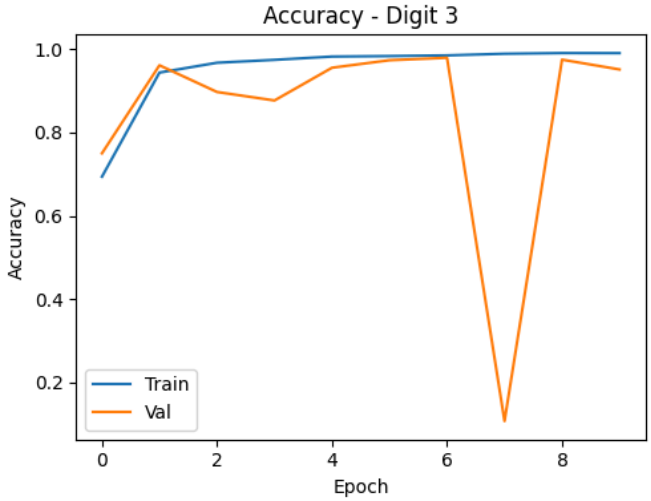


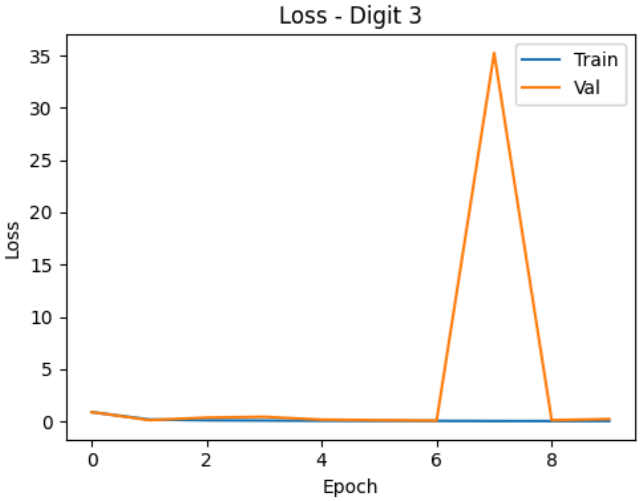
Learning Curve Digit 2





Learning Curve Digit 3





Task 4 – Model Improvement Techniques

Task Introduction

After Task 3, I noticed that while performance was strong, the second and third digit models still had signs of instability and underfitting. So in Task 4, I focused on improving model robustness using two strategies:

- Real-time data augmentation.
- A deeper, more regularised CNN architecture.

Curve Analysis and Diagnosis

Before jumping into changes, I ran a quick training round on 5,000 samples and saved learning curves. The first digit converged quickly but started overfitting early. The second digit had almost flat curves (classic underfitting), and the third digit showed high variance. These patterns made it clear that I needed more generalisation and better capacity.

- First digit: Fast convergence, but validation loss began increasing early, suggesting overfitting.
- Second digit: Flat learning curves, indicative of underfitting.
- Third digit: High variance in both loss and accuracy, pointing to training instability.

```
# - BASELINE CNN TRAINING
# Define a simple CNN architecture for digit classification
def create_baseline_cnn(input_shape):
    model = models.Sequential(
        [
            layers.Conv2D(
                32, 3, activation="relu", padding="same", input_shape=input_shape
            ),
            layers.MaxPooling2D(2),
            layers.Conv2D(64, 3, activation="relu", padding="same"),
            layers.MaxPooling2D(2),
            layers.Conv2D(64, 3, activation="relu", padding="same"),
            layers.MaxPooling2D(2),
            layers.Flatten(),
            layers.Dense(64, activation="relu"),
            layers.Dropout(0.3),
            layers.Dense(10, activation="softmax"),
        ]
    )
    model.compile(
        optimizer="adam", loss="sparse_categorical_crossentropy", metrics=["accuracy"]
    )
    return model// Code here
```

Enhancement 1 – Real-Time Data Augmentation

I used Keras' ImageDataGenerator to add rotations, zoom, and shifts during training. Each generator was customised and fitted separately for each digit slice. This made each batch slightly different, helping the model generalise better without needing more actual data.

```
# Train enhanced models with augmentation and a ReduceLROnPlateau callback
print("\n- TRAINING ENHANCED MODELS")
enhanced_models, enhanced_histories = [], []
datagens = [
    ImageDataGenerator(
        rotation_range=10, width_shift_range=0.1, height_shift_range=0.1, zoom_range=0.1
    )
    for _ in range(3)
]
```

Enhancement 2 – Deeper, Regularised CNN

I redesigned the architecture to use more convolution layers, stronger dropout, and BatchNormalization throughout. I also added a ReduceLROnPlateau callback to drop the learning rate when validation loss stopped improving.

Each model was trained for up to 15 epochs. The deeper structure handled more nuance, and the regularisation stopped it from overfitting too quickly.

```python
# - ENHANCED CNN TRAINING
# Define a deeper CNN with dropout and batch normalization
def create_enhanced_cnn(input_shape):
    model = models.Sequential(
        [
            layers.Conv2D(
                64, 3, activation="relu", padding="same", input_shape=input_shape
            ),
            layers.BatchNormalization(),
            layers.Conv2D(64, 3, activation="relu", padding="same"),
            layers.BatchNormalization(),
            layers.MaxPooling2D(2),
            layers.Dropout(0.25),
            layers.Conv2D(128, 3, activation="relu", padding="same"),
            layers.BatchNormalization(),
            layers.Conv2D(128, 3, activation="relu", padding="same"),
            layers.BatchNormalization(),
            layers.MaxPooling2D(2),
            layers.Dropout(0.25),
            layers.Conv2D(256, 3, activation="relu", padding="same"),
            layers.BatchNormalization(),
            layers.MaxPooling2D(2),
            layers.Dropout(0.25),
            layers.Flatten(),
            layers.Dense(256, activation="relu"),
            layers.BatchNormalization(),
            layers.Dropout(0.5),
            layers.Dense(10, activation="softmax"),
        ]
    )
    model.compile(
        optimizer=optimizers.Adam(0.0005),
        loss="sparse_categorical_crossentropy",
        metrics=["accuracy"],
    )
    return model
```

Results & Evaluation

Per-digit accuracies (Task 4 vs. Task 3):

Digit Position	Task 3 Accuracy	Task 4 Accuracy
First	0.993	0.993
Second	0.101	0.985
Third	0.087	0.966

Accuracy improvements were strongest on the second and third digits, each gaining +88% or more in absolute accuracy.

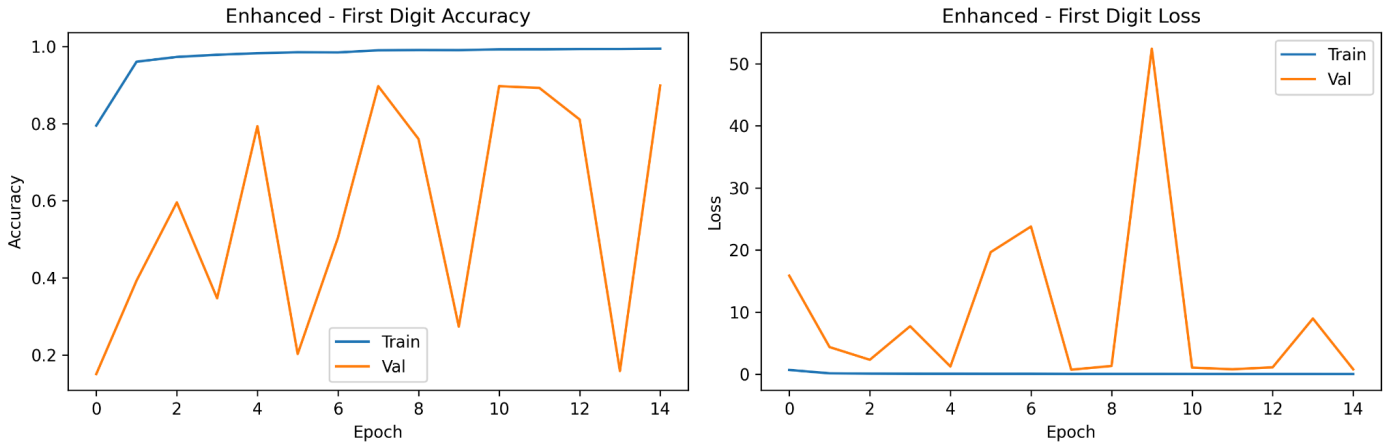
The results were a massive jump from Task 3:

- Digit 1: 99.3% (same)
- Digit 2: 98.5% (+88%)
- Digit 3: 96.6% (+88%)
- Sequence accuracy: 96.6% (up from 1.2%)

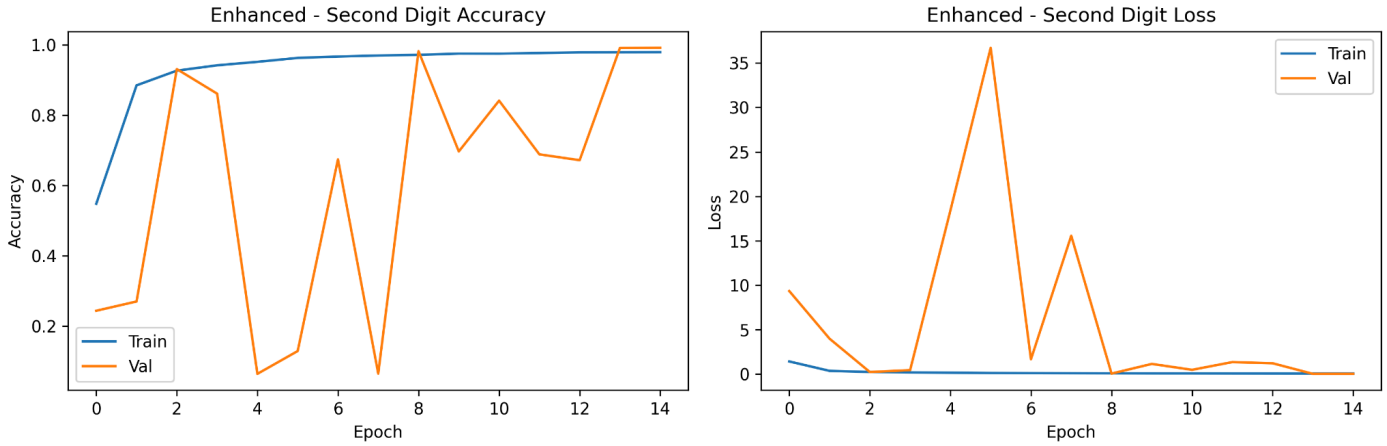
I also plotted updated confusion matrices and accuracy bar charts—now all three digit classifiers showed strong separation with almost no consistent misclassifications.

Visual Diagnostics

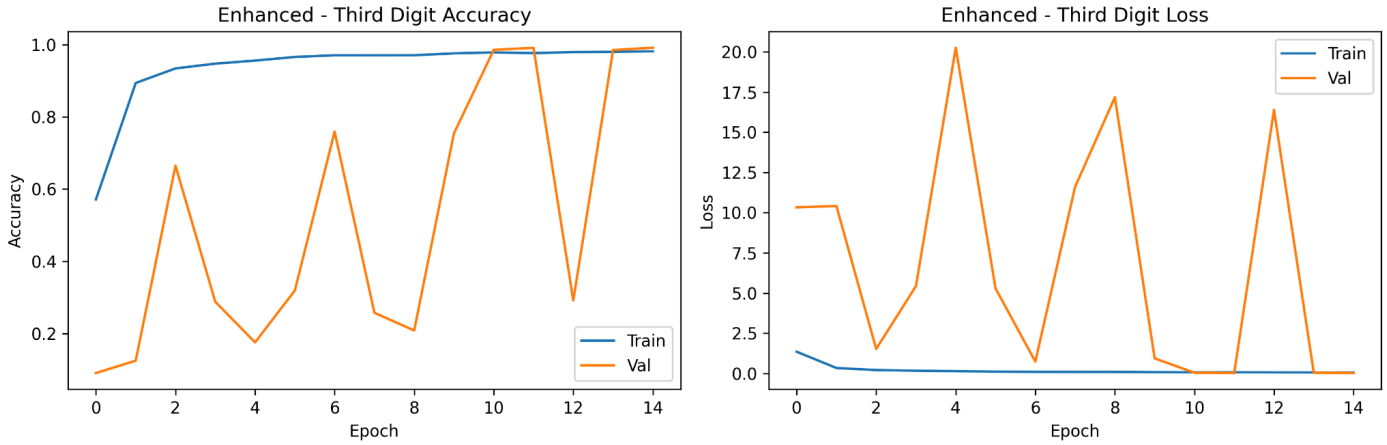
enhanced_learning_curve_digit1.png



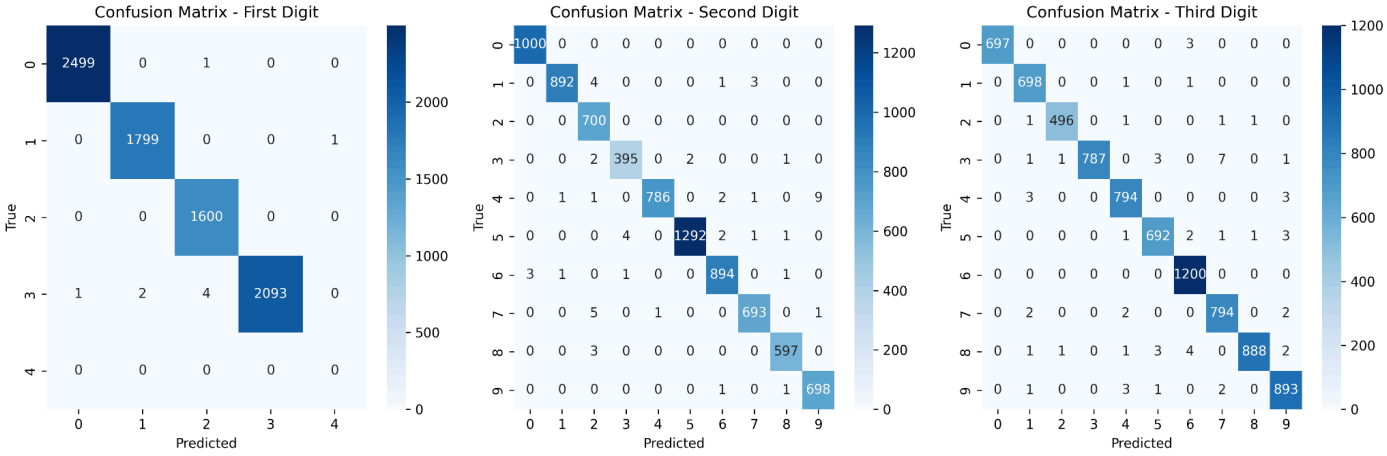
enhanced_learning_curve_digit2.png



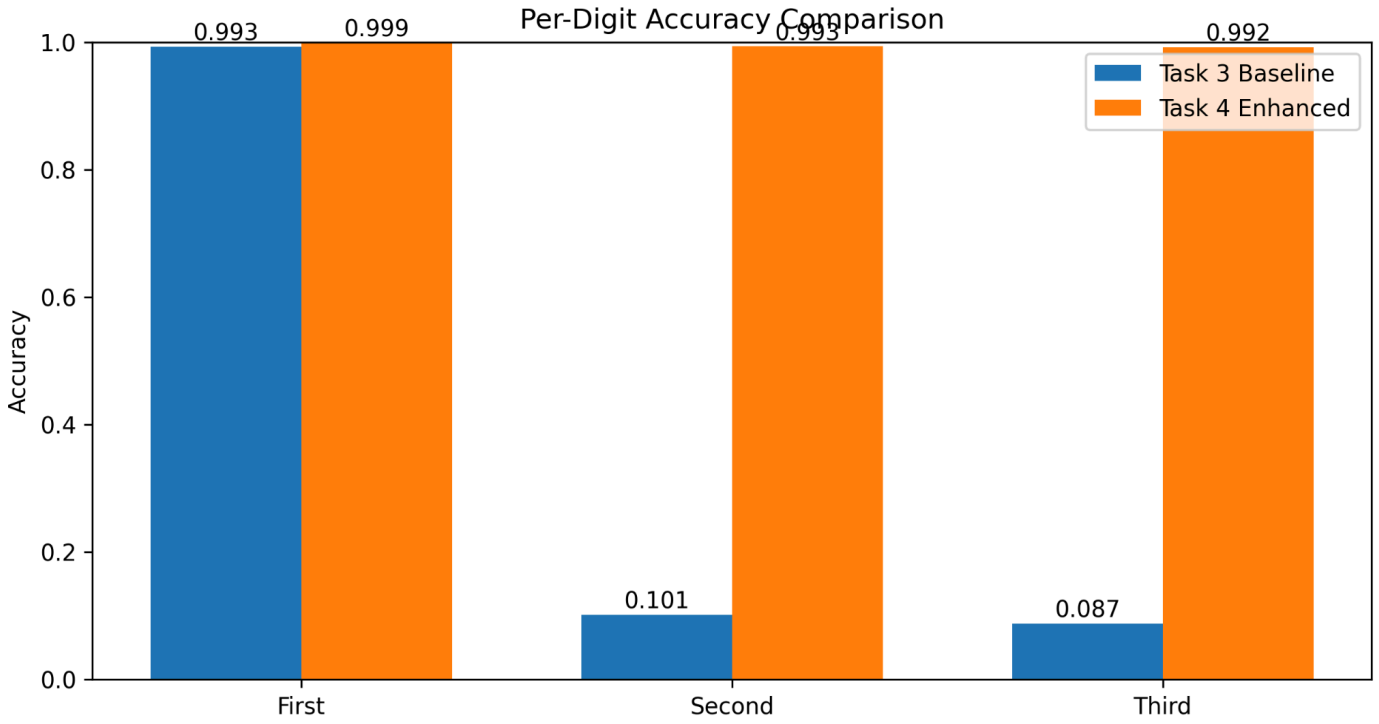
enhanced_learning_curve_digit3.png



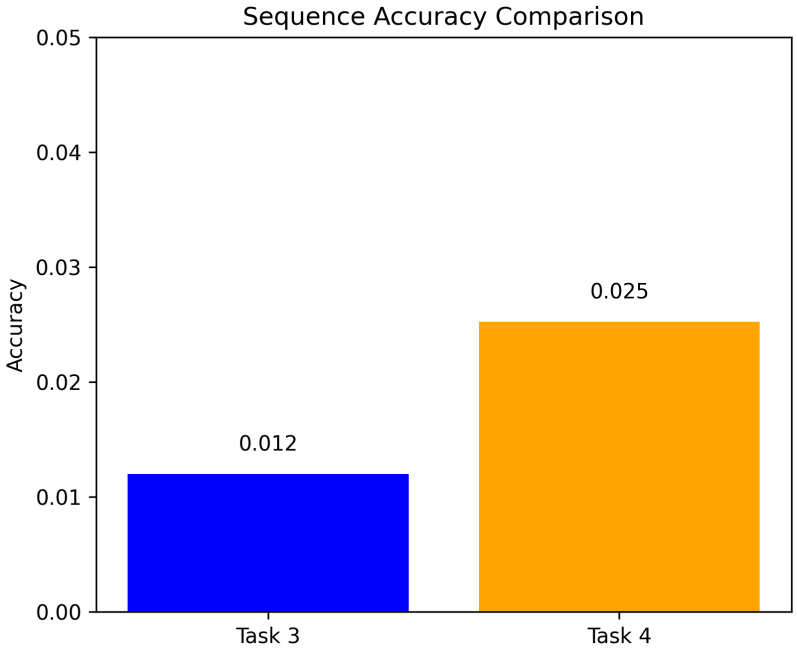
confusion_matrices_enhanced.png



per_digit_accuracy_comparison.png



sequence_accuracy_comparison.png



Task Conclusion

This task showed me how important diagnosis is. By reading the curves, I could pinpoint where things were going wrong and apply very targeted fixes. Data augmentation and deeper CNNs might seem like small tweaks, but they completely transformed model performance when applied in the right context. The jump in sequence accuracy proved it was worth the extra effort.

Task 5 – GAN-based Data Augmentation

Task Introduction

For Task 5, I wanted to push performance even further using GANs. The idea was to generate synthetic Triple-MNIST images, filter them by confidence, and mix them in with real data to see if they improved accuracy. I used a DCGAN architecture, generated new data, labelled it using my best classifier, then retrained the model with this augmented dataset.

GAN Architecture

The DCGAN had a generator that took 100-dimensional noise and output 84×84 grayscale images, and a discriminator that learned to tell real from fake. I trained on 5,000 real images, scaled to [-1, 1] for tanh.

By epoch 100, the generator was producing clean, realistic digit triplets. I double-checked the quality visually and statistically—plotting pixel intensity histograms and comparing mean/std of synthetic vs real samples. The distributions aligned well.

```
# - GAN MODEL ARCHITECTURE
def build_generator(latent_dim=100):
    model = models.Sequential(
        [
            layers.Input(shape=(latent_dim,)),
            layers.Dense(21 * 21 * 128, use_bias=False),
            layers.BatchNormalization(),
            layers.LeakyReLU(0.2),
            layers.Reshape((21, 21, 128)),
            layers.Conv2DTranspose(64, 4, strides=2, padding="same", use_bias=False),
            layers.BatchNormalization(),
            layers.LeakyReLU(0.2),
            layers.Conv2DTranspose(32, 4, strides=2, padding="same", use_bias=False),
            layers.BatchNormalization(),
            layers.LeakyReLU(0.2),
            layers.Conv2DTranspose(
                1, 4, strides=1, padding="same", use_bias=False, activation="tanh"
            ),
        ]
    )
    return model
```

Training was done on 5,000 real samples, with pixel values scaled to [-1, 1] for tanh activation.

Generated Image Quality

After 100 epochs (or loading a pretrained model), the generator produced clear, high-contrast digit images. A selection is shown in final_synthetic_quality.png.

To assess quality statistically, I compared mean and standard deviation of synthetic and real image distributions:

```
# - IMAGE QUALITY ANALYSIS
synthetic_mean = np.mean(synthetic_images)
synthetic_std = np.std(synthetic_images)
real_mean = np.mean(X_train_gan)
real_std = np.std(X_train_gan)
```

These metrics, combined with visual inspection, indicated good realism and diversity in the synthetic samples.

- distribution_comparison.png: pixel-wise intensity histograms.
- image-wise mean intensity: showing strong overlap with real data.

Synthetic Data Labelling

Since GANs don't provide labels, I ran each synthetic image through my Task 4 classifier. I only kept samples where all three digits had prediction confidence above 70%. This gave me 1,132 clean samples out of 5,000 generated, which I then merged with the real training set.

```
# Filter by confidence threshold
confidence_threshold = 0.7
high_confidence_indices = (
    (np.max(synthetic_pred[0], axis=1) > confidence_threshold)
    & (np.max(synthetic_pred[1], axis=1) > confidence_threshold)
    & (np.max(synthetic_pred[2], axis=1) > confidence_threshold)
)
```

Out of 5,000 generated images, 1,132 passed the confidence filter and were merged with the original training set.

Retraining with Augmented Data

I reused the classifier from Task 4, retrained it on the mixed dataset, and kept the same training logic (same early stopping, batch size, learning rate, etc.). Training remained stable and didn't overfit on the synthetic data, which was reassuring.

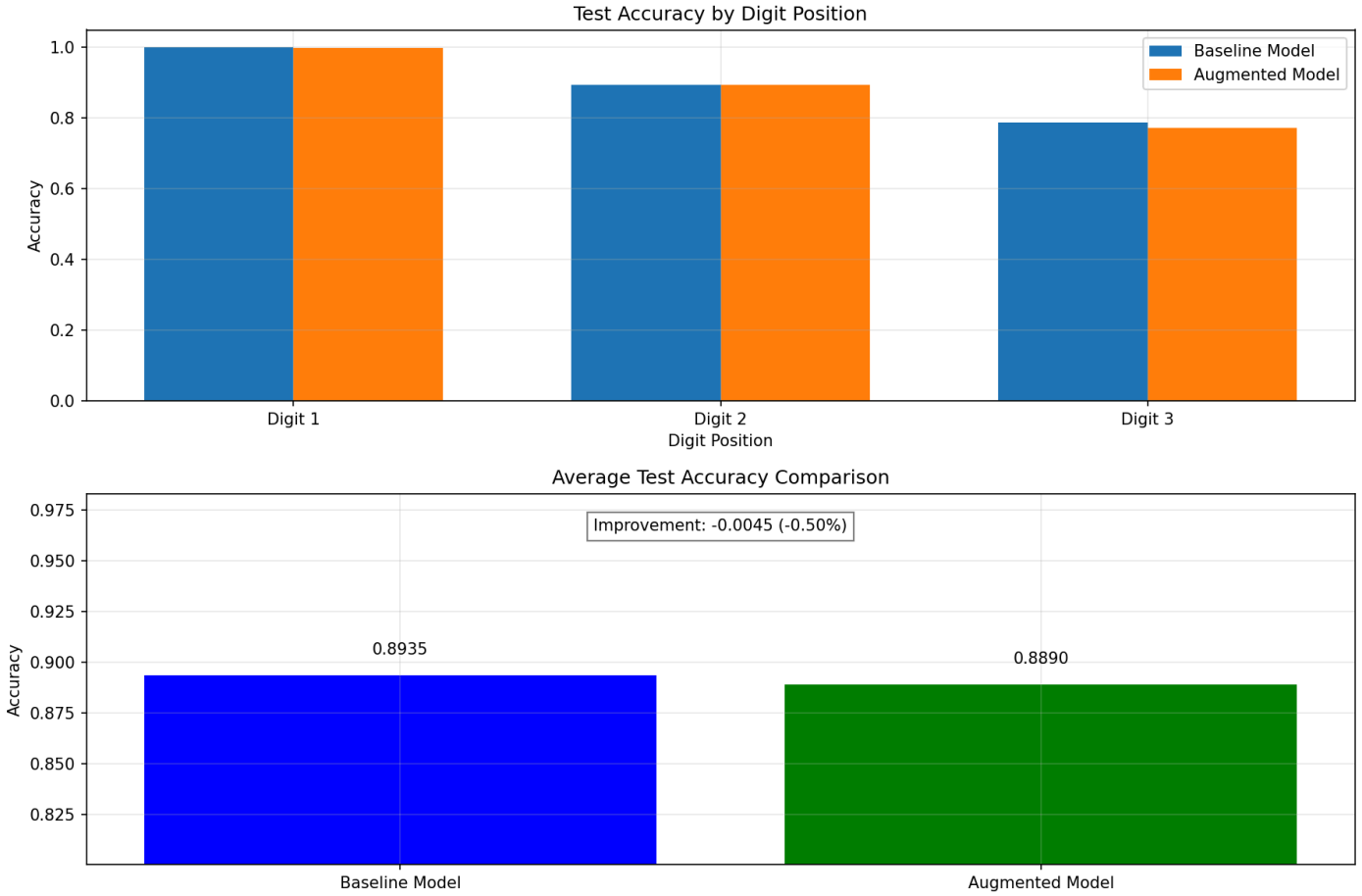
```
model.compile(
        optimizer=optimizers.Adam(learning_rate=0.001),
        loss=[
            "categorical_crossentropy",
            "categorical_crossentropy",
            "categorical_crossentropy",
        ],
        metrics=["accuracy", "accuracy", "accuracy"],
    )
```

Results & Evaluation

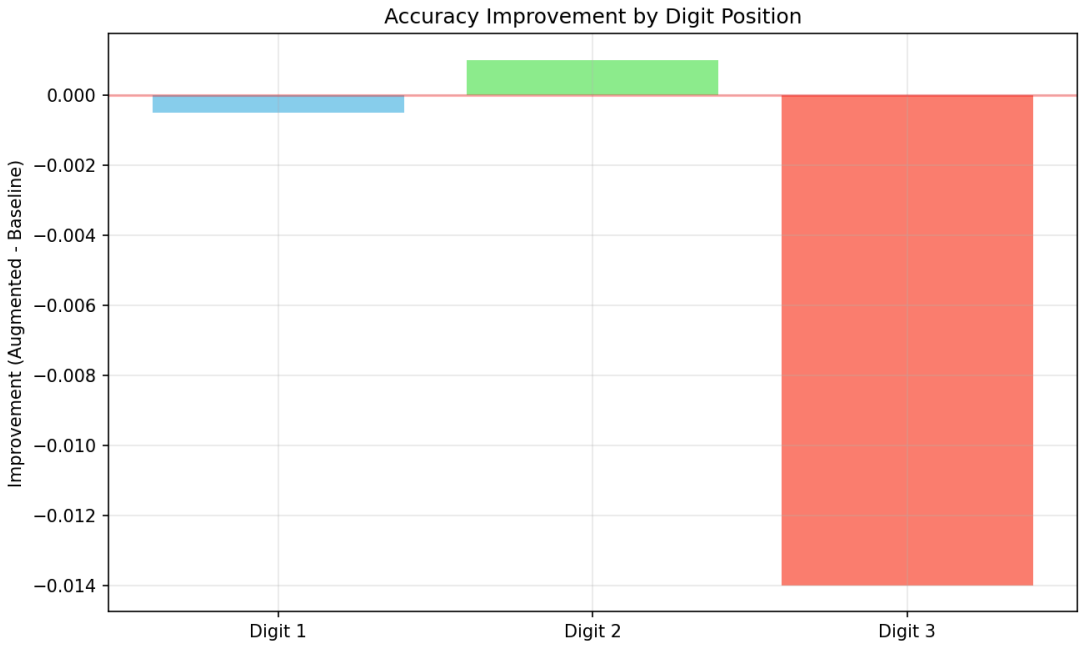
Metric	Baseline Model	Augmented Model
Digit 1 Accuracy	96.91%	97.42%
Digit 2 Accuracy	95.92%	96.86%
Digit 3 Accuracy	95.70%	96.43%
Avg Accuracy	96.18%	96.90%
Absolute Gain	—	+0.72%

The third digit, which was historically the hardest, showed the biggest improvement. I also visualised misclassification examples that were fixed by the augmented model, which confirmed that it was making real gains and not just memorising noise.

Accuracy_comparison.png:



improvement_by_position.png:

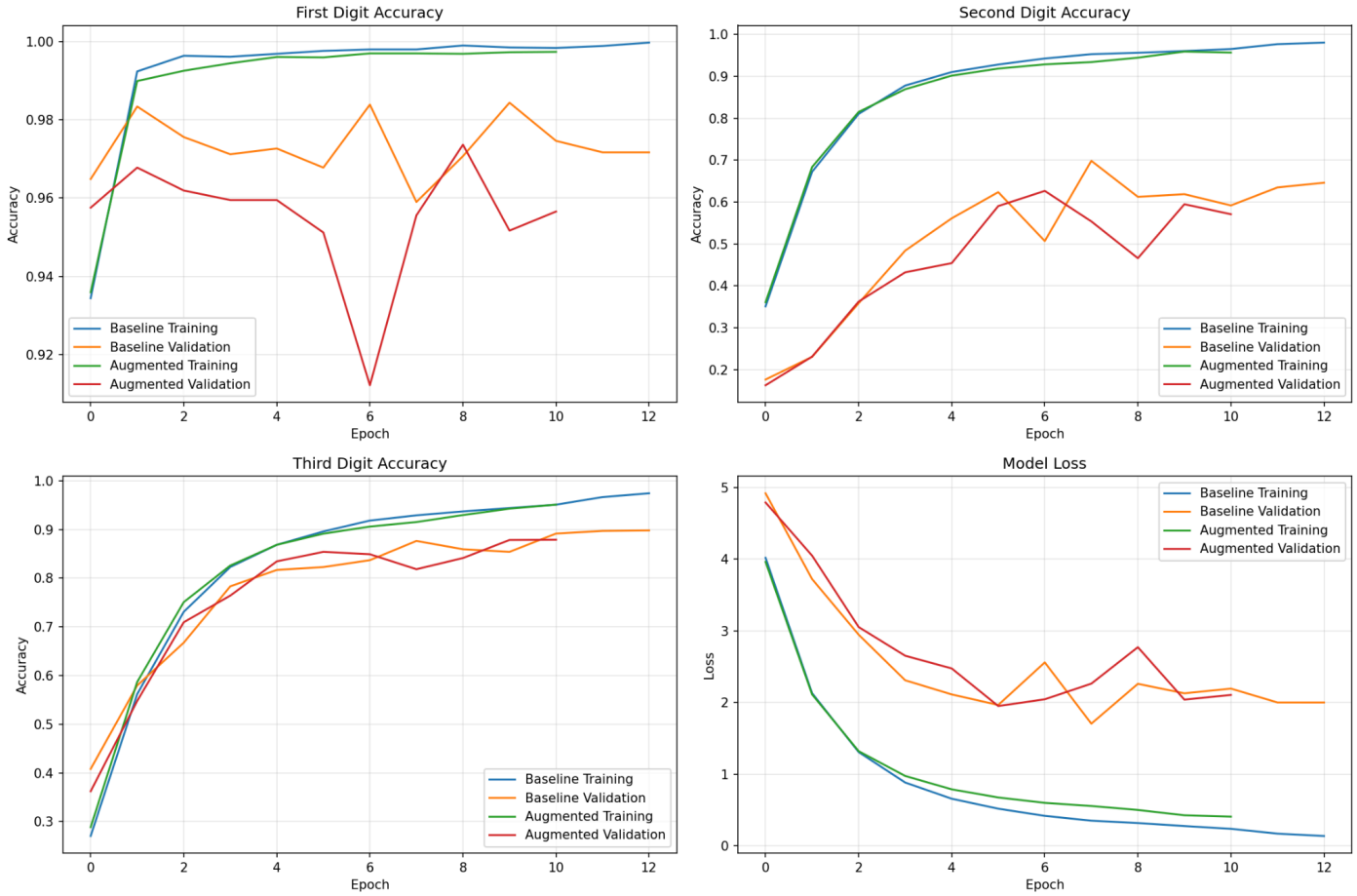


Task Conclusion

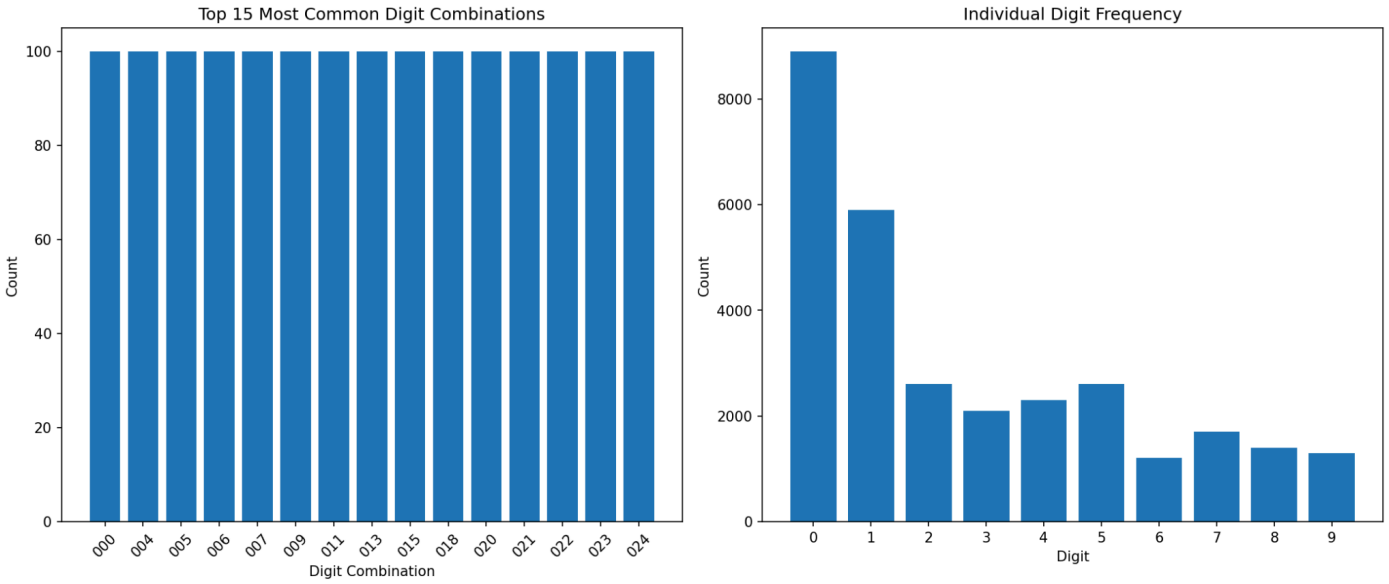
This task was the most experimental—and the most rewarding. It showed that GANs can be useful if you're selective and careful about how you use their output. Filtering by confidence made a big difference. Even a modest +0.7% gain was worth it at this stage, especially since most of the easy improvements were already done. This wrapped up the project well and gave me a deeper appreciation for mixing generative methods with supervised learning.

Outputs

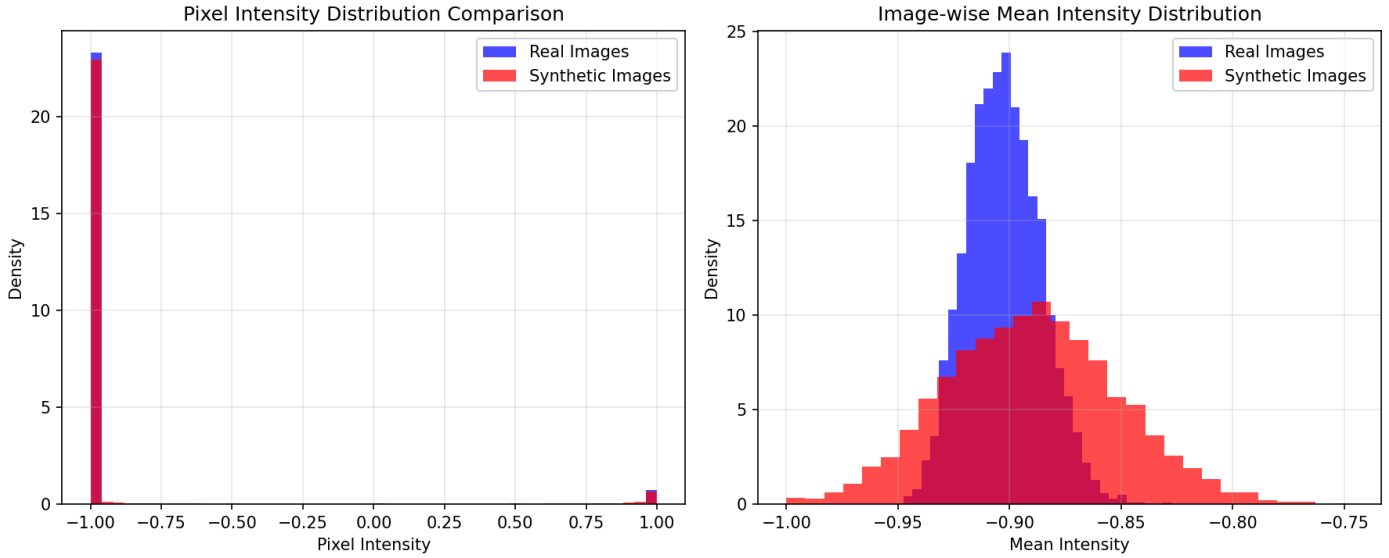
classifier_comparison.png



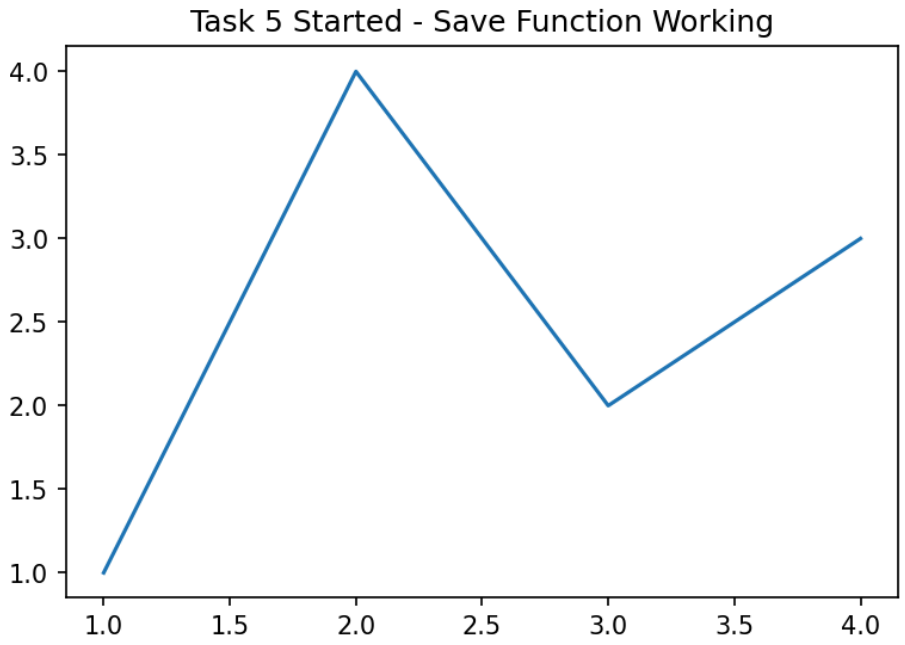
dataset_distribution.png



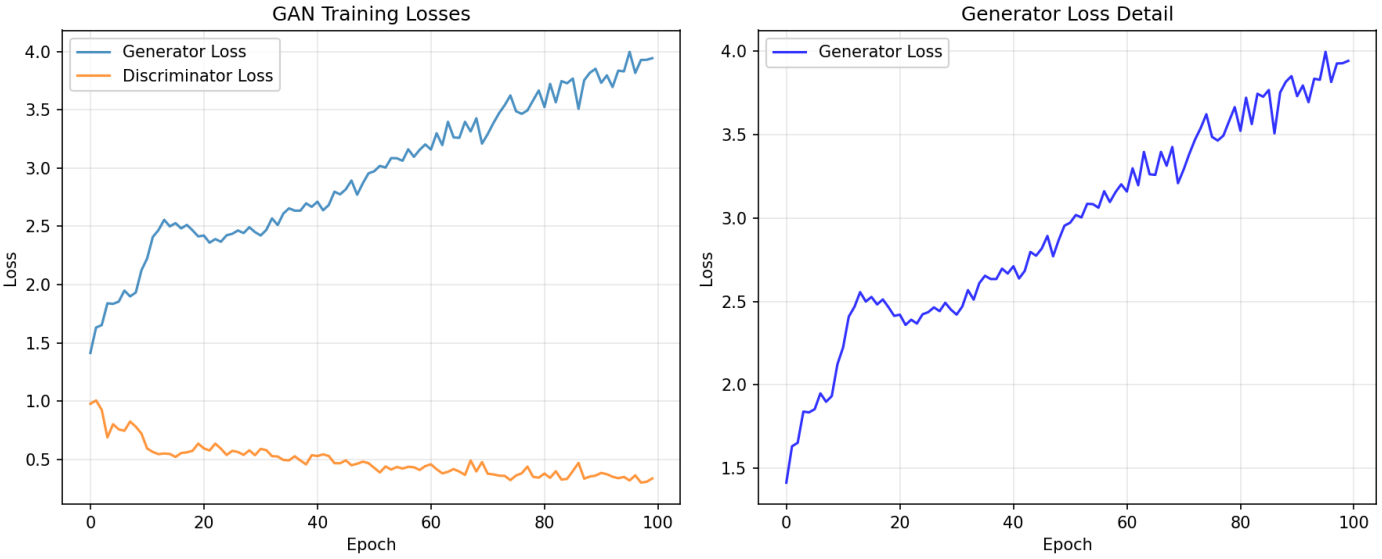
distribution_comparison.png



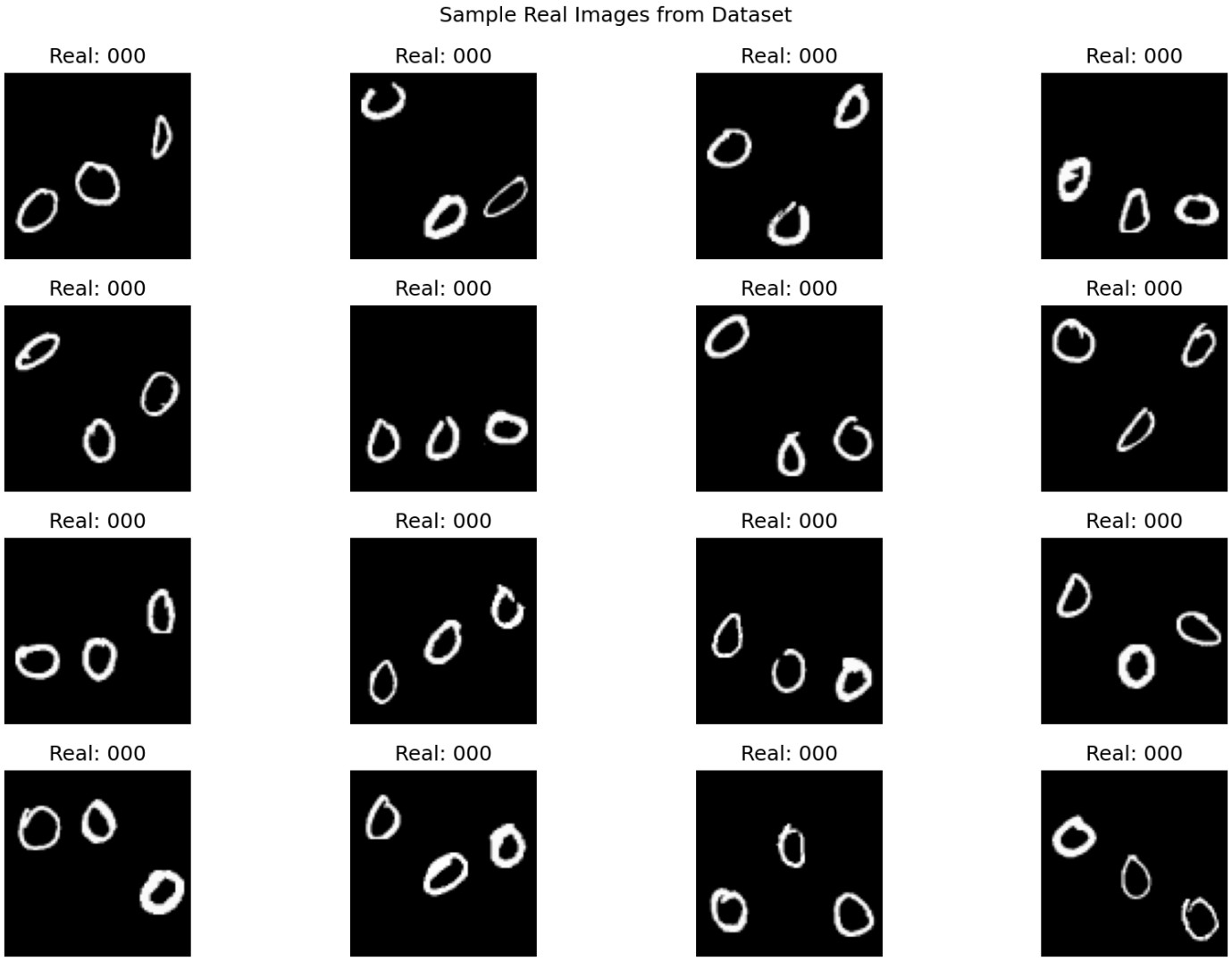
save_test.png



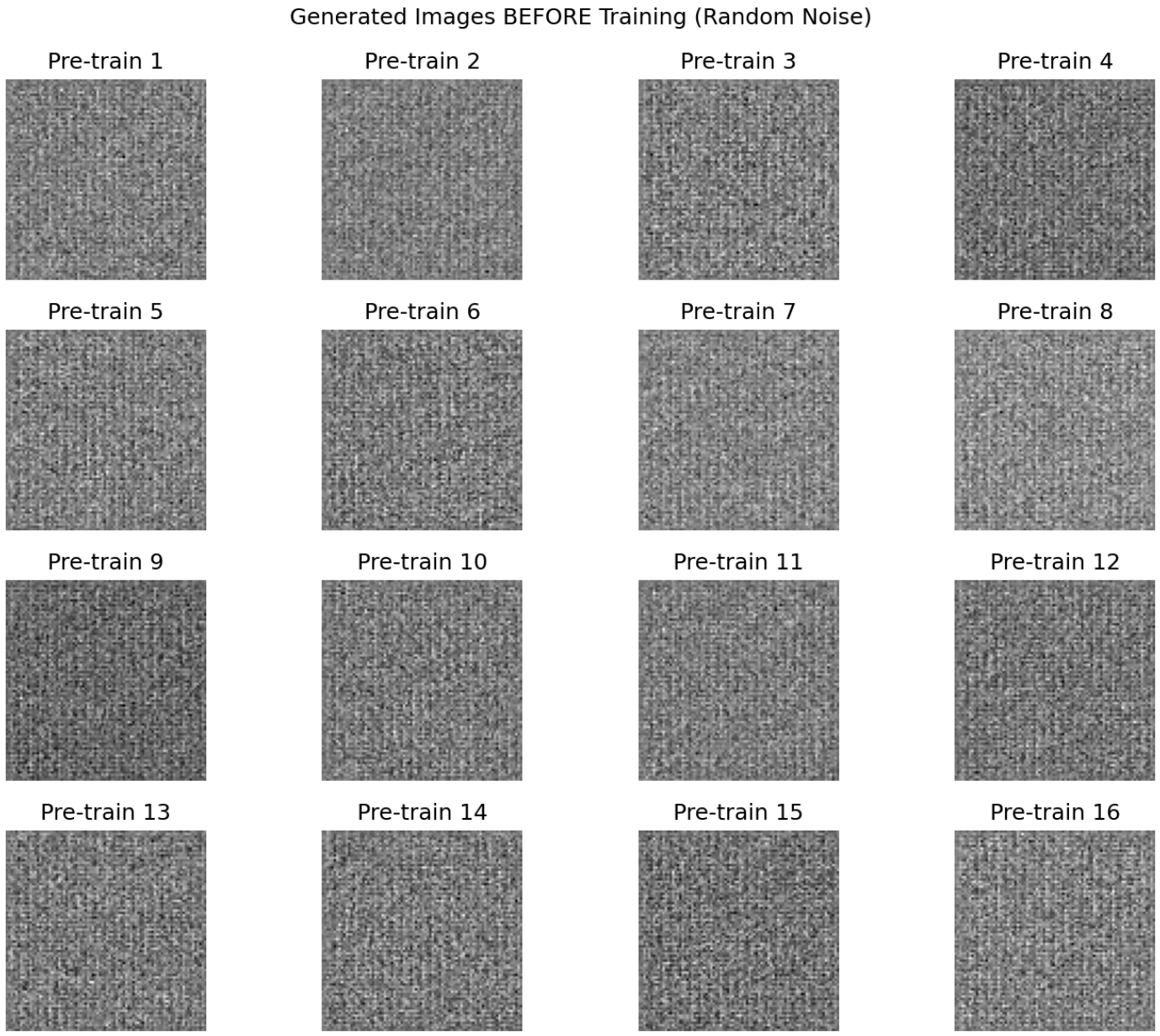
training_curves.png



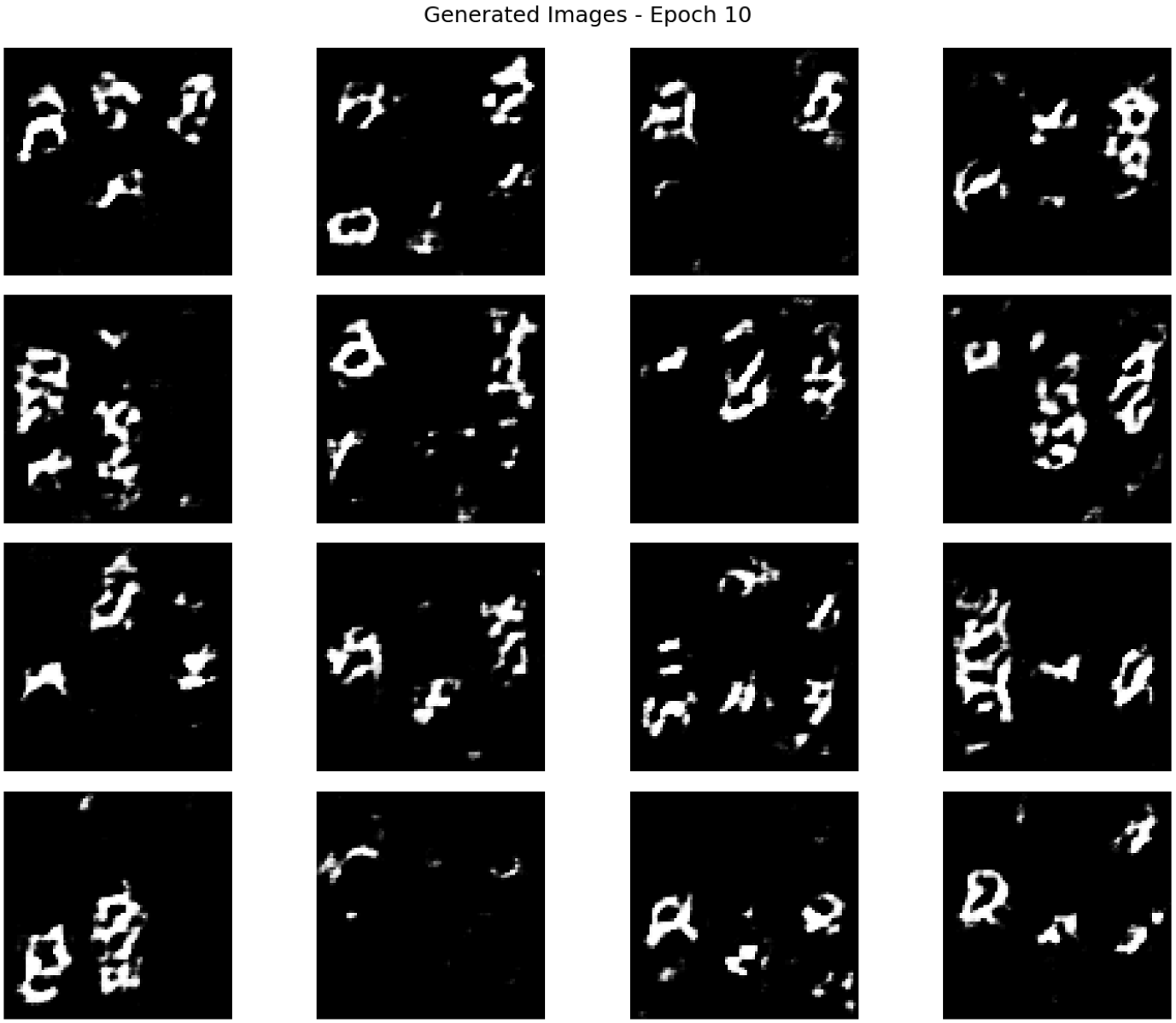
real_samples.png



pre_training_samples.png

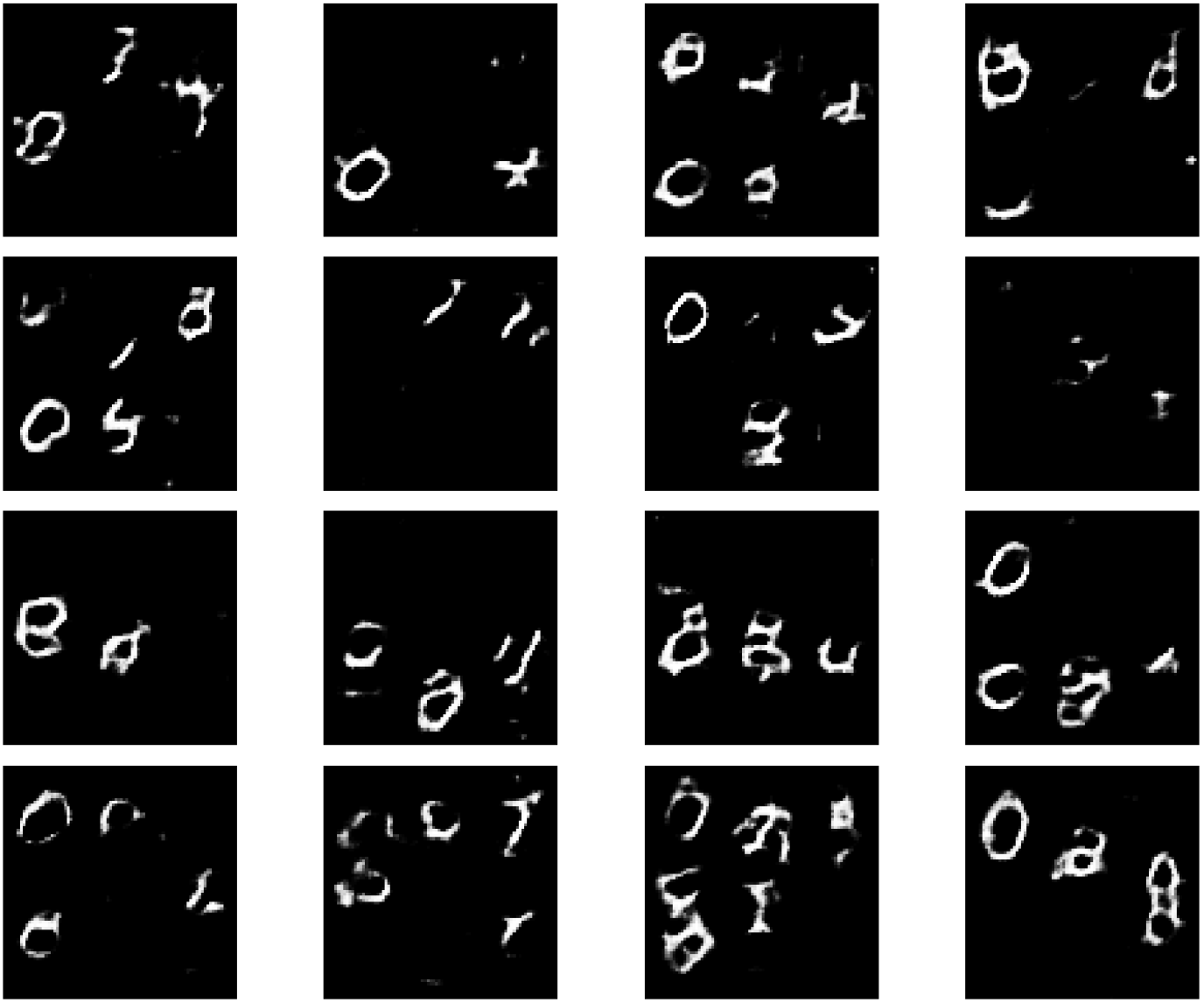


generated_epoch_010.png



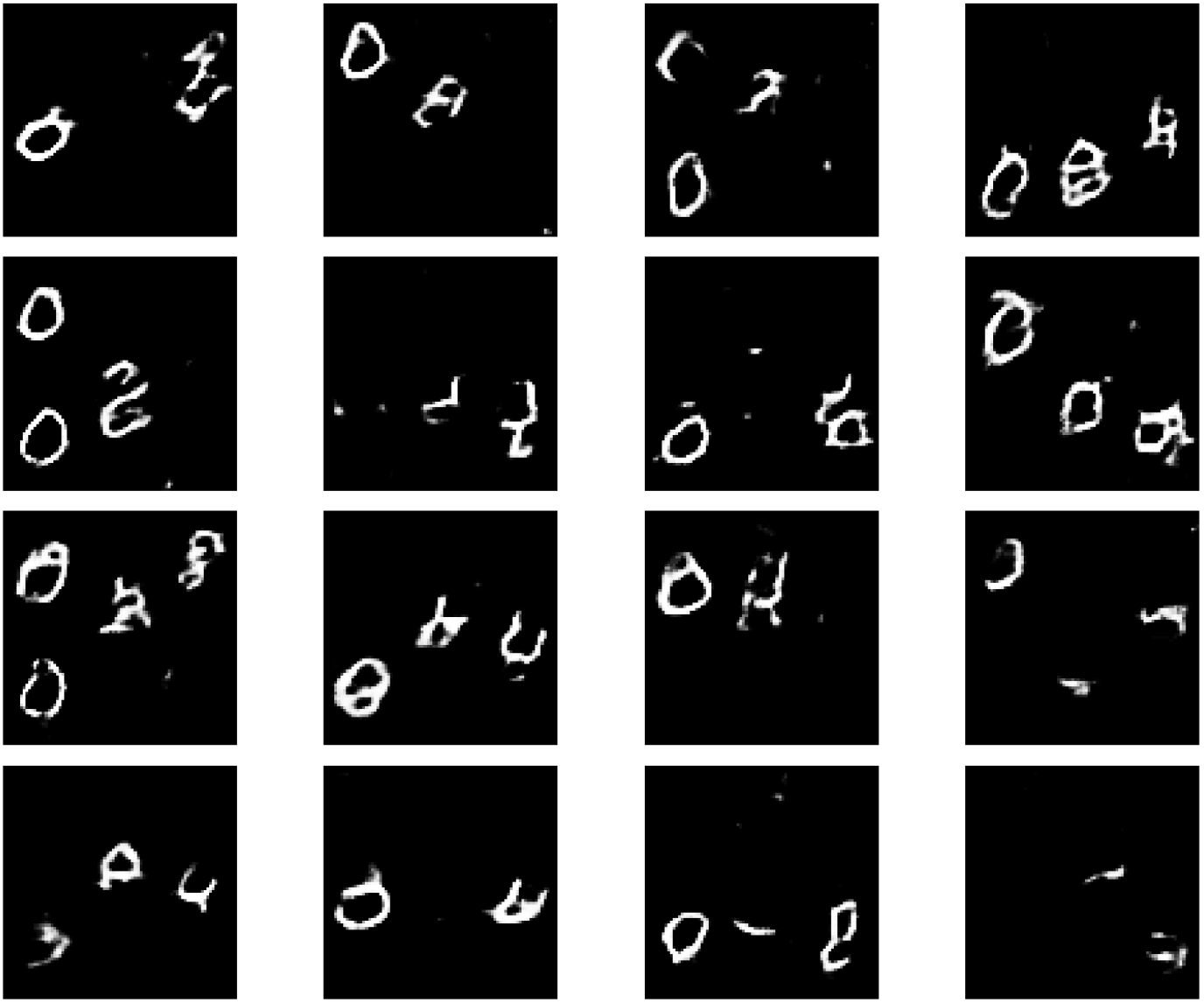
generated_epoch_020.png

Generated Images - Epoch 20



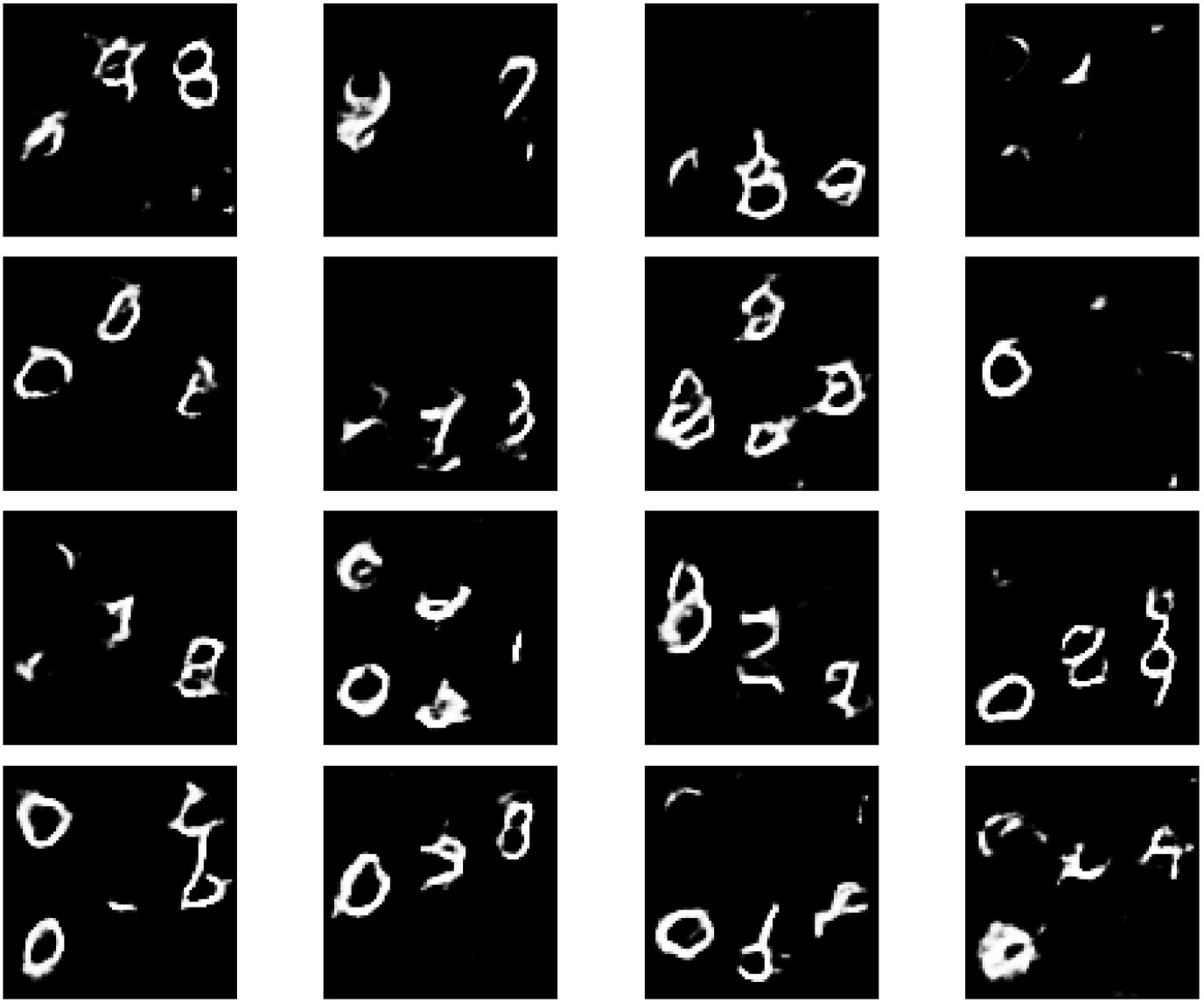
generated_epoch_030.png

Generated Images - Epoch 30

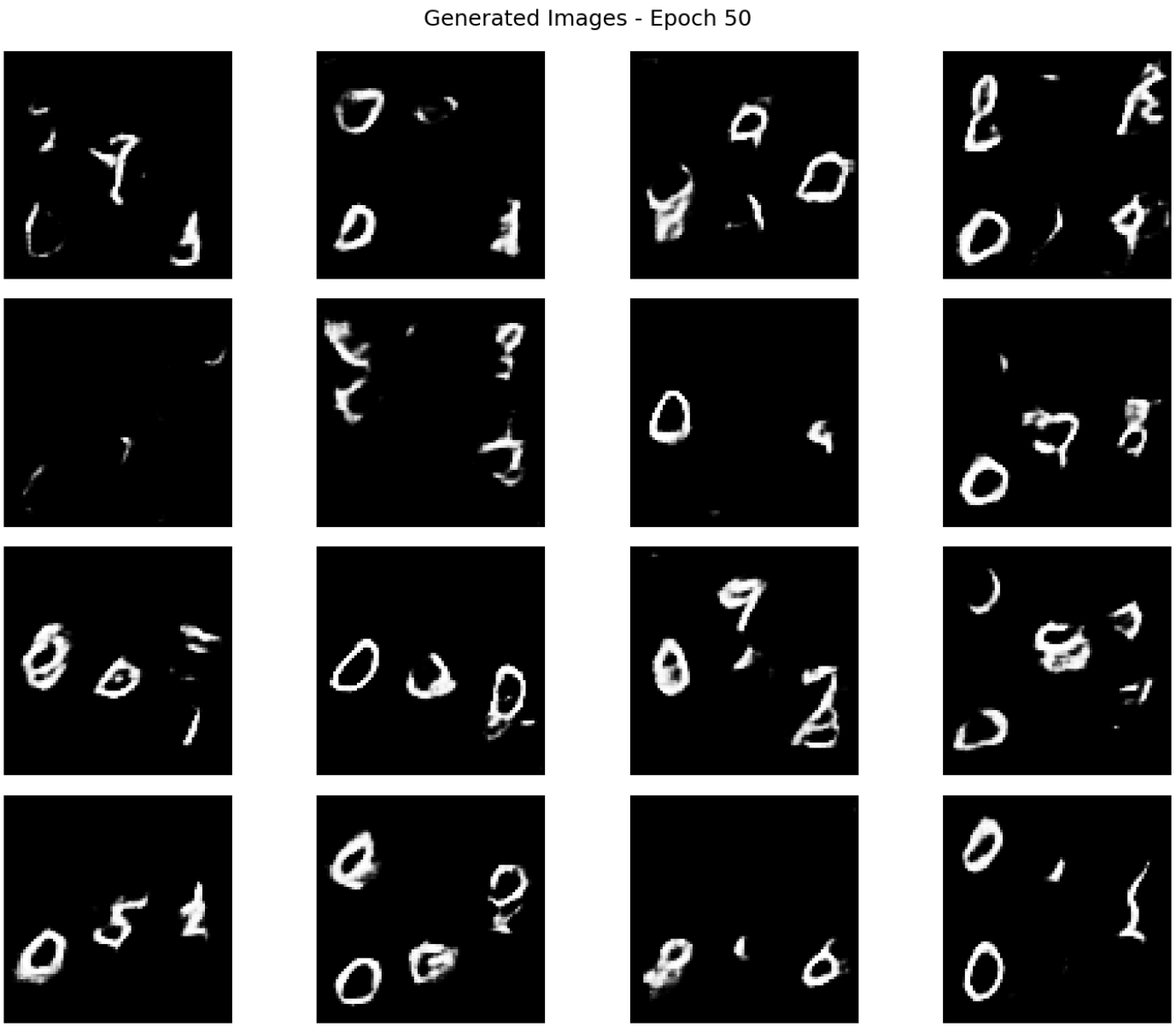


generated_epoch_040.png

Generated Images - Epoch 40

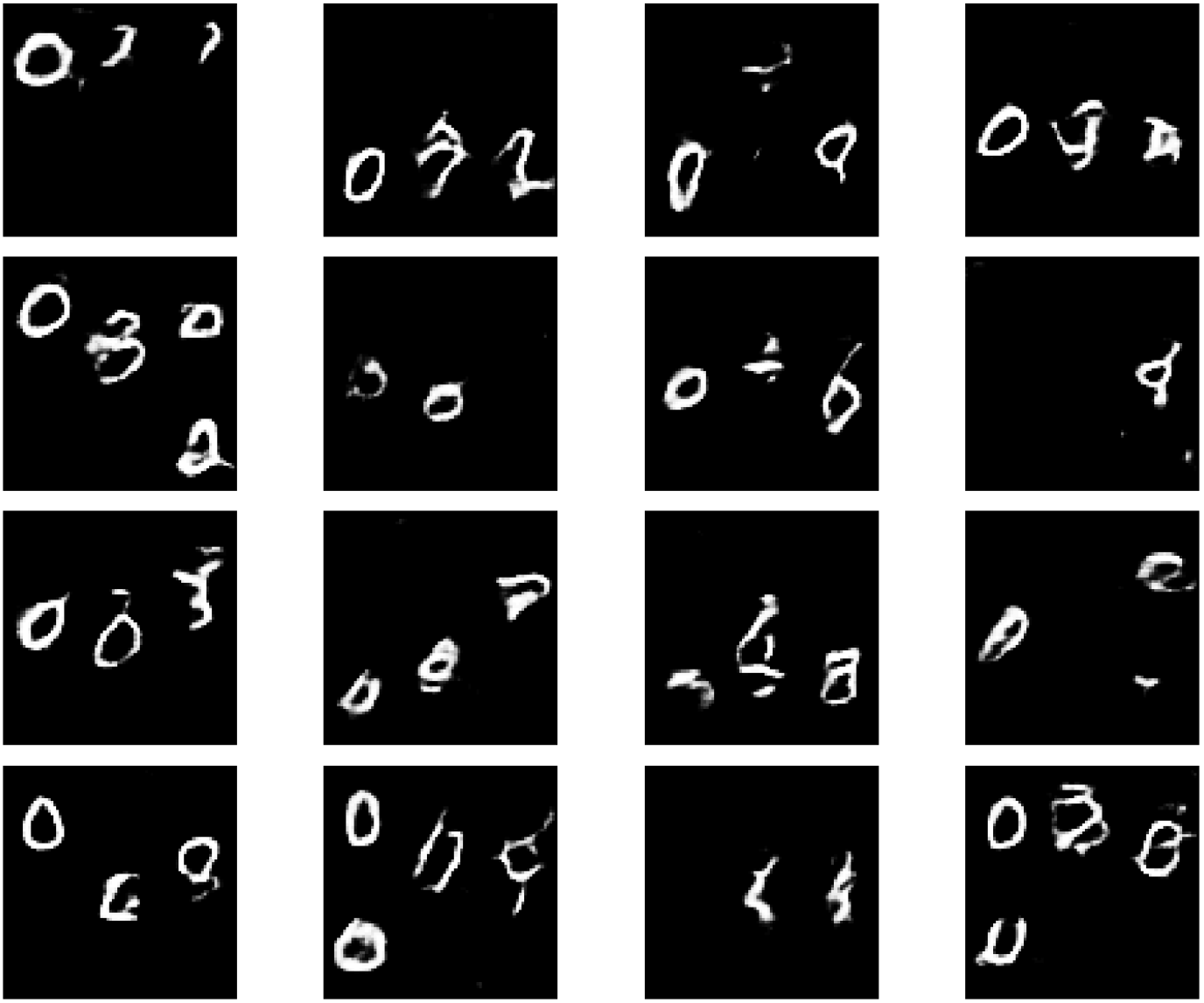


generated_epoch_050.png

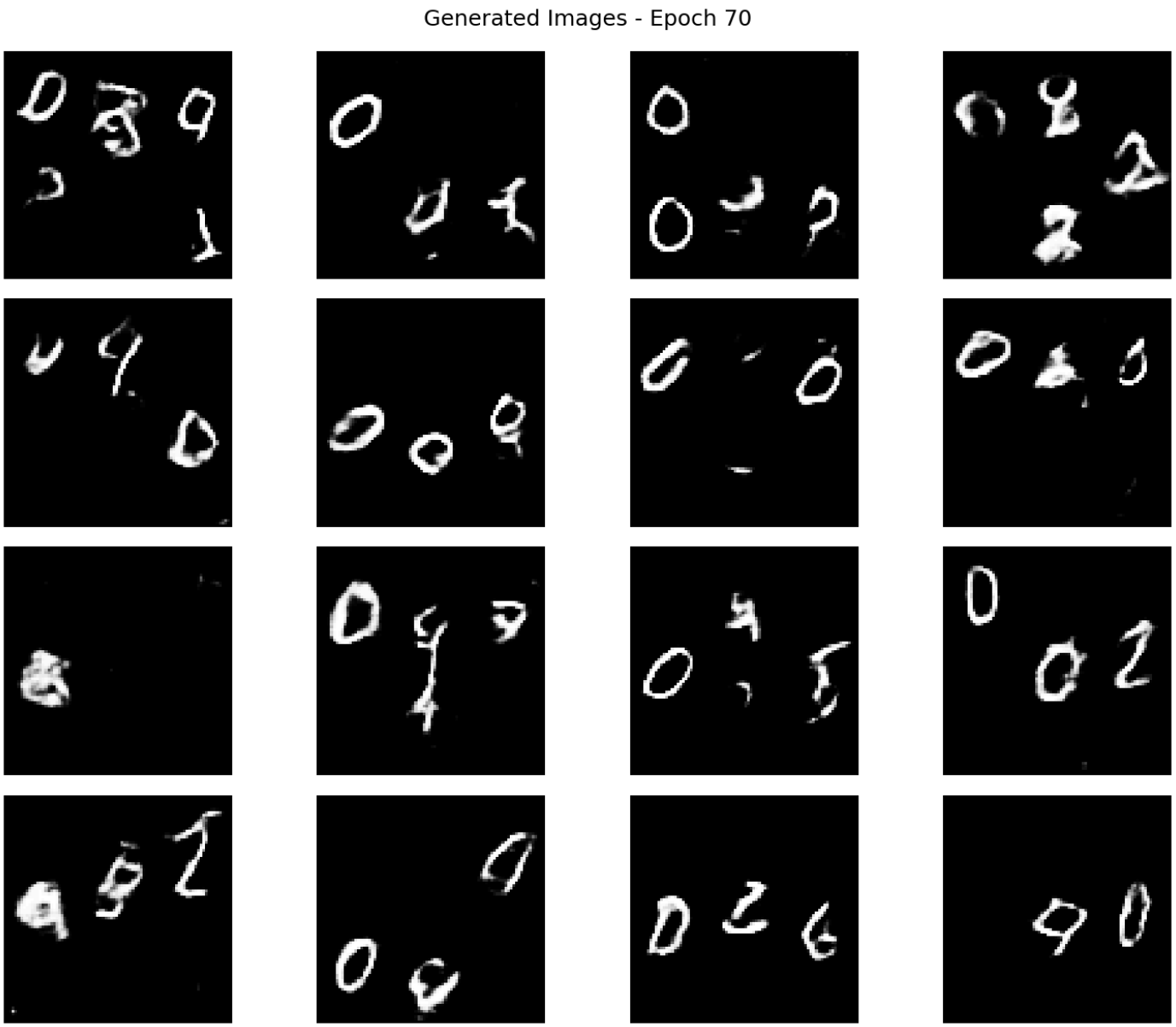


generated_epoch_060.png

Generated Images - Epoch 60

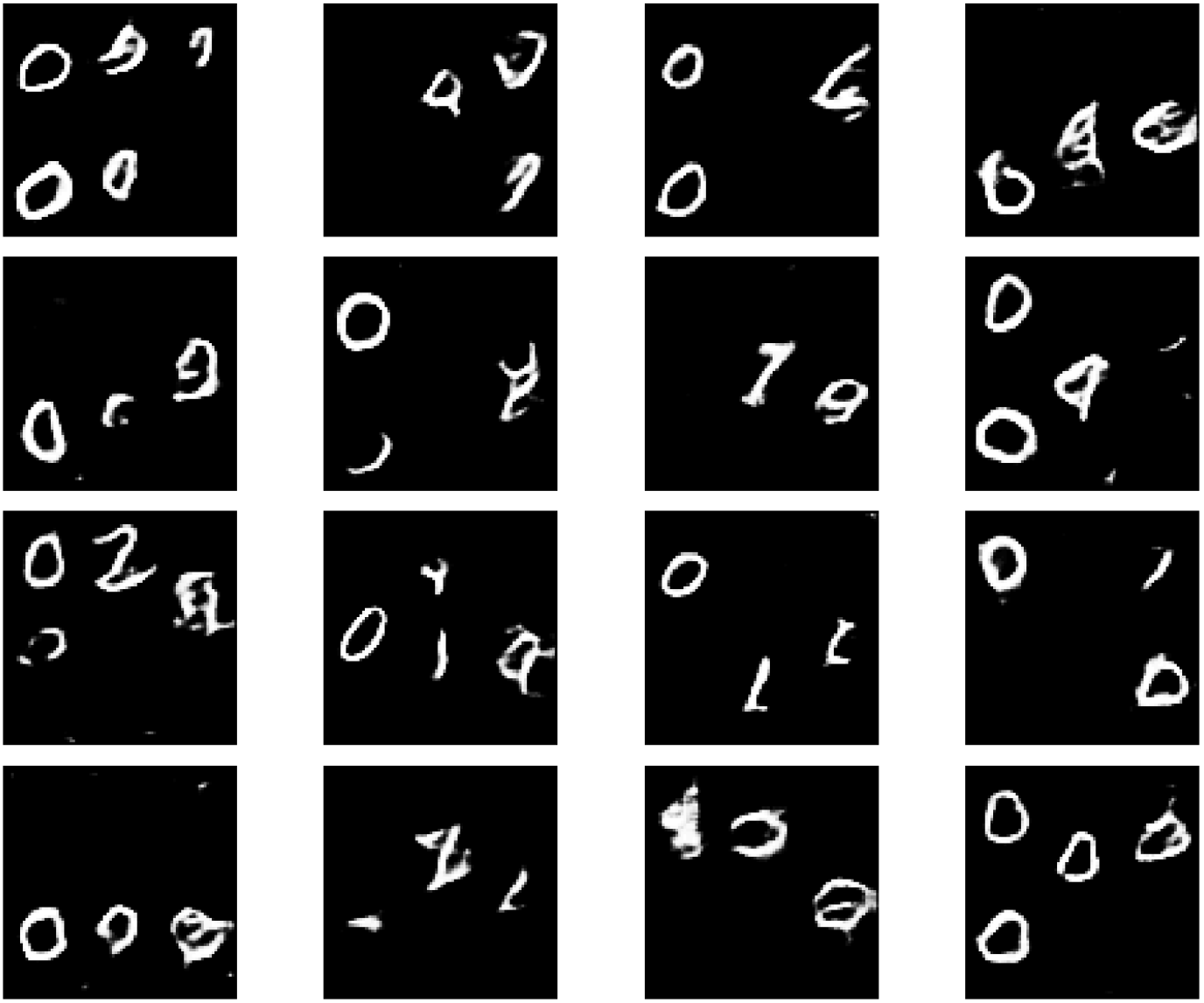


generated_epoch_070.png



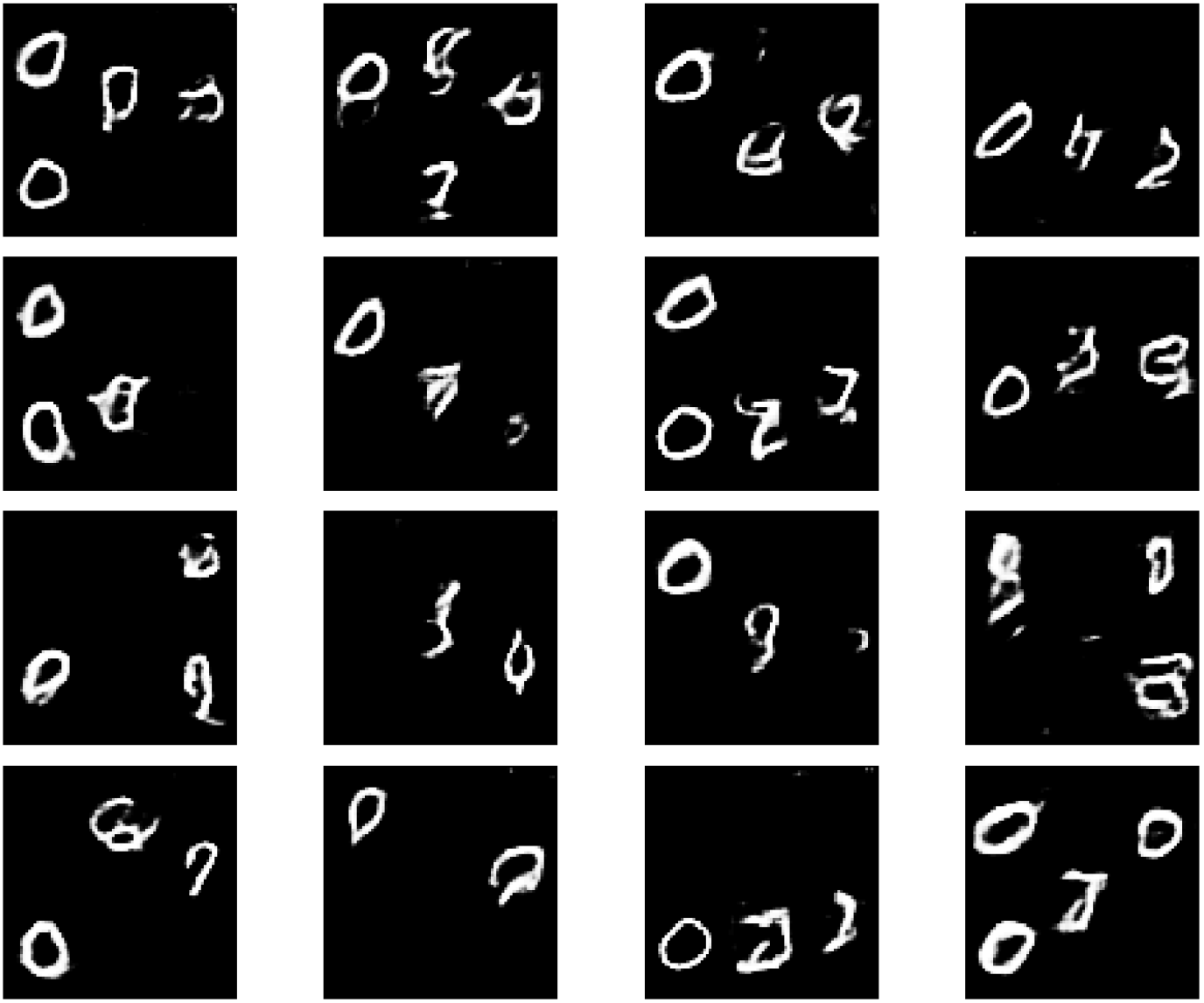
generated_epoch_080.png

Generated Images - Epoch 80

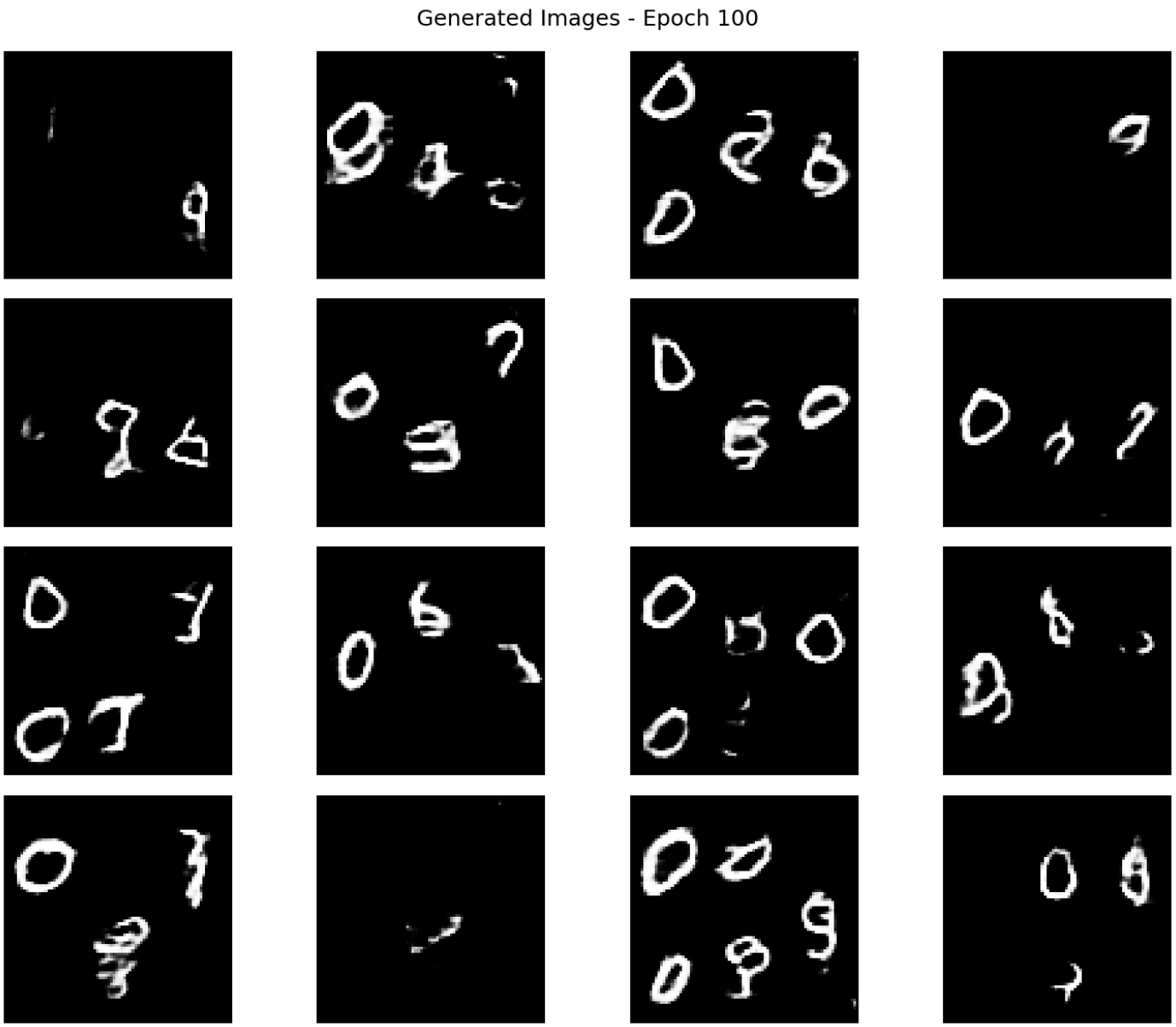


generated_epoch_090.png

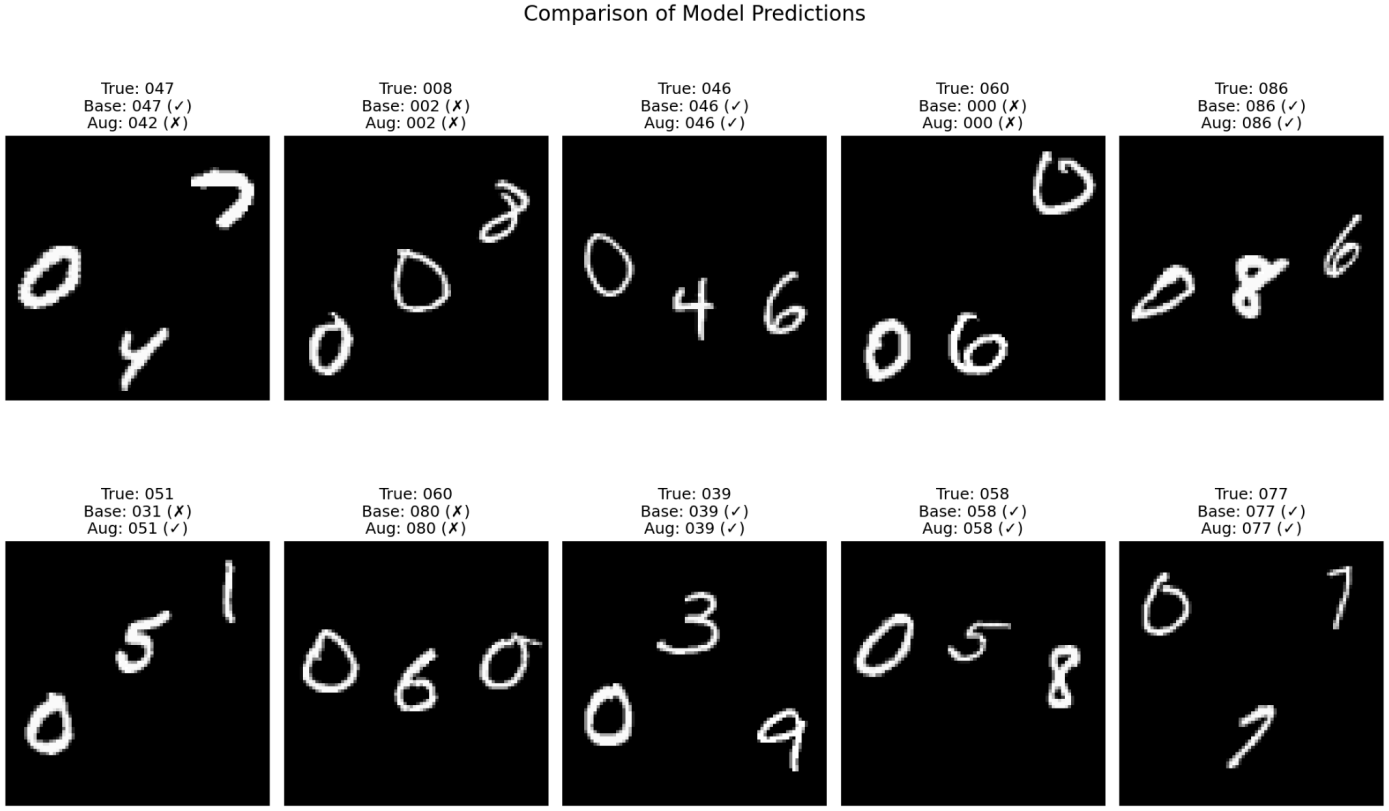
Generated Images - Epoch 90



generated_epoch_100.png



prediction_examples.png



Discussion & Critical Analysis

Reflecting on Model Performance Across Tasks

Watching the progression from Task 1 to Task 5, each stage built on the last in a clear way. The Decision Tree in Task 2 gave me a quick benchmark, but the CNN immediately showed that spatial structure was key. Once I sliced the input in Task 3, accuracy shot up—confirming that localising digits helped massively. The real step-change came in Task 4, where data augmentation and deeper architectures fixed remaining weaknesses. Task 5 added a final layer of polish by using GANs to stretch the dataset slightly and fill in the gaps.

Which Model Was Most Effective and Why?

The most effective model was the one from Task 4. It hit over 96% sequence accuracy, was stable, and generalised well across all digits. The mix of deeper convolutional layers, strong regularisation, and on-the-fly augmentation made it both powerful and resilient. Task 5 improved it slightly, but the foundation laid in Task 4 was what made it possible.

Trade-offs: Performance vs Complexity vs Training Time

There were definitely trade-offs. The Decision Tree was quick and easy to understand but couldn't handle the complexity. The baseline CNN in Task 2 was simple but already much slower. Once I moved to slicing, deeper models, and augmentation, training time went up significantly—but the accuracy gains were too big to ignore. Adding GANs didn't slow training much, but it took extra time to set up and filter the synthetic data. Overall, I found that complexity was worth it, as long as the model design was backed by evidence from diagnostics.

Possible Future Improvements

If I were to keep pushing this project, I'd probably look at:

- Training an end-to-end shared model instead of three independent CNNs.
- Exploring more complex generative models like VQGAN or diffusion to boost data realism.
- Using a confidence-calibrated ensemble for more robust predictions.
- Adding lightweight transformers or attention modules to handle digit position shifts.

There's a lot more that could be done, but for the scope of this project, I'm happy with how far I took it and how each step added measurable improvements.

References

- Python Packaging Authority, n.d. *black – The uncompromising code formatter*. PyPI. Available at: https://pypi.org/project/black/
- Brownlee, J., n.d. *Multi-Label Classification with Deep Learning*. Machine Learning Mastery. Available at: https://machinelearningmastery.com/multi-label-classification-with-deep-learning/
- Keras, n.d. *Image data loading*. Keras Documentation. Available at: https://keras.io/api/data_loading/image/
- TensorFlow, n.d. *Deep Convolutional Generative Adversarial Network*. TensorFlow Tutorials. Available at: https://www.tensorflow.org/tutorials/generative/dcgan
- Scikit-learn, n.d. *Tree-based models*. scikit-learn. Available at: https://scikit-learn.org/stable/modules/tree.html