GTU Department of Computer Engineering
CSE 655
Homework 2 Report

Emirkan Burak Yılmaz
254201001054

Contents

Part 1: Digit Recognition Using CNNs

Overview

A convolutional neural network was constructed using TensorFlow and Keras to perform classification on the MNIST dataset. The dataset was loaded and preprocessed by normalizing pixel values to the [0,1] range and reshaping the inputs to (28,28,1). The labels were converted to one-hot encoding.

```python
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout, BatchNormalization
from tensorflow.keras.optimizers import Adam

# Load data
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Normalize input data
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Reshape to add channel dimension
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)

# One-hot encode labels
y_train_cat = to_categorical(y_train, 10)
y_test_cat = to_categorical(y_test, 10)

print("Training data shape:", x_train.shape)
print("Test data shape:", x_test.shape)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 1s 0us/step
Training data shape: (60000, 28, 28, 1)
Test data shape: (10000, 28, 28, 1)
```

Figure 1: Data Loading and Preprocessing

```python
import tensorflow as tf
from tensorflow.keras import layers, models
import matplotlib.pyplot as plt

model = models.Sequential([
    layers.Input(shape=(28, 28, 1)),

    layers.Conv2D(96, (3, 3), padding='same', activation='relu'),
    layers.BatchNormalization(),
    layers.Conv2D(96, (3, 3), padding='same', activation='relu'),
    layers.BatchNormalization(),
    layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),

    layers.Conv2D(256, (3, 3), padding='same', activation='relu'),
    layers.BatchNormalization(),
    layers.Conv2D(256, (3, 3), padding='same', activation='relu'),
    layers.BatchNormalization(),
    layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),

    layers.Conv2D(384, (3, 3), padding='same', activation='relu'),
    layers.Conv2D(384, (3, 3), padding='same', activation='relu'),
    layers.Conv2D(256, (3, 3), padding='same', activation='relu'),
    layers.MaxPooling2D(pool_size=(2, 2), strides=(1, 1)),

    layers.Flatten(),
    layers.Dense(4096, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(4096, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(10, activation='softmax')
])

model.summary()
```

Figure 2: CNN Architecture

The final architecture consisted of over 62 million parameters, slightly exceeding the complexity of AlexNet by incorporating additional convolutional layers, deeper filter stacks.

- Total params: **62,735,658** (239.32 MB)
- Trainable params: **62,734,250** (239.31 MB)
- Non-trainable params: **1,408** (5.50 KB)

Results

```
# Compile
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Train the model
history = model.fit(x_train, y_train_cat,
                    epochs=5,
                    batch_size=128,
                     validation_split=0.1)
```

```
Epoch 1/5
422/422 ━━━━━━━━━━━━━━━━━━━━ 36s 54ms/step - accuracy: 0.4621 - loss: 1.7393 - val_accuracy: 0.6837 - val_loss: 1.0961
Epoch 2/5
422/422 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - accuracy: 0.9692 - loss: 0.1137 - val_accuracy: 0.9782 - val_loss: 0.0879
Epoch 3/5
422/422 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - accuracy: 0.9783 - loss: 0.0851 - val_accuracy: 0.9883 - val_loss: 0.0475
Epoch 4/5
422/422 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - accuracy: 0.9837 - loss: 0.0672 - val_accuracy: 0.9783 - val_loss: 0.1120
Epoch 5/5
422/422 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - accuracy: 0.9834 - loss: 0.0716 - val_accuracy: 0.9862 - val_loss: 0.0503
```

Figure 3: Training Process

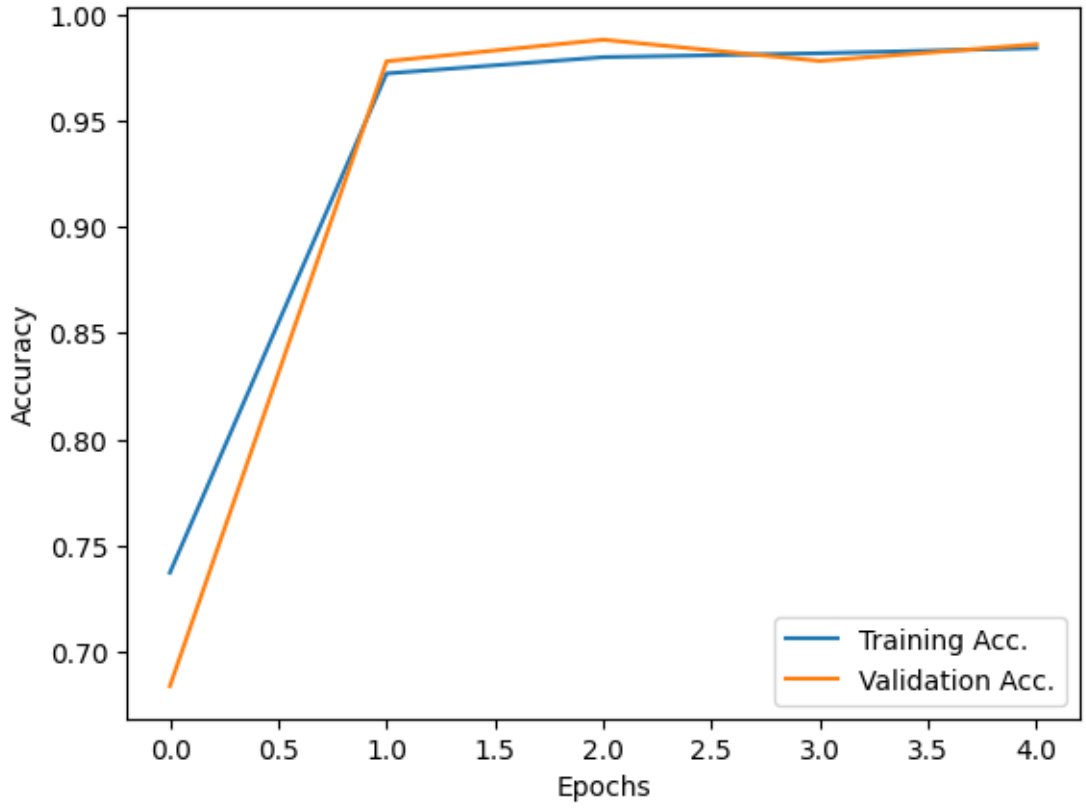


Figure 4: Training vs Validation Accuracy per Epoch

- Training Accuracy: **98.34%**
- Validation Accuracy: **98.62%**
- Test Accuracy: **98.57%**

Comments

- After several trials, training for 5 epochs was found to be sufficient.
- Increasing model complexity beyond a certain point provided diminishing returns on test accuracy.
- Batch normalization accelerated convergence and helped maintain generalization.
- Use of dropout layers reduced overfitting despite the large model capacity.
- Final model achieved near-perfect performance on the training set and strong generalization to unseen test samples.
- The trained model was saved in Keras format and used as a fixed function in Part 2 for input optimization experiments.

Part 2: Input optimization for a CNN

Overview

A gradient-based optimization algorithm was implemented to discover input images that **maximize the output activation** for a given class label. Optimization was performed using **gradient ascent**, updating the input image while keeping the trained CNN model **frozen** to maximize the activation of a desired output class.

- Initialization Options:
    - Pure random grayscale images sampled uniformly from [0,1]
    - Noisy real images sampled from the dataset
- Loss Function:
    - Primary loss: **categorical cross entropy** between the model output and the target vector
    - Regularization: **L2 penalty** on the input image to discourage extreme pixel values.
- Optimization Details:
    - Adam optimizer was used with a learning rate of 0.05
    - Pixel values were **clipped** to [0,1] at each step
    - Optimization was run for 500-1000 steps with snapshots taken every 100 steps
    - Each snapshot was saved along with the current loss value for visualization
- Target Vectors:
    - One-hot vectors were used for single-digit targets
    - Soft labels (e.g., [0,0,0.5,0.5,0,0,0,0,0,0]) were used to represent interpolated digit classes.

```python
def create_target_output(digits, weights=None):
    target_output = np.zeros((1, 10), dtype=np.float32)

    if isinstance(digits, int):
        target_output[0, digits] = 1.0
    else:
        if weights is None:
            weights = [1.0 / len(digits)] * len(digits)
        for d, w in zip(digits, weights):
            target_output[0, d] = w

    return target_output

def create_random_gray_image():
    return np.random.rand(28, 28, 1).reshape(1, 28, 28, 1).astype(np.float32)

def sample_random_image_from_dataset(x_train, y_train, target_digit, seed=0, add_noise=False, noise_scale=0.3):
    np.random.seed(seed)
    indices = np.where(y_train == target_digit)[0]
    random_index = np.random.choice(indices)
    image = x_train[random_index].reshape(1, 28, 28, 1).astype(np.float32)

    if add_noise:
        noise = noise_scale * np.random.randn(*image.shape)
        image += noise
        image = np.clip(image, 0.0, 1.0)

    return image
```

Figure 5: Utility Functions for Generating Initial Images and Target Output Vectors

```python
def optimize_with_snapshots(model, target_output, img, steps=500, lr=0.05, snapshot_interval=100):
    optimized_img = tf.Variable(img, trainable=True)
    optimizer = tf.keras.optimizers.Adam(learning_rate=lr)
    loss_fn = tf.keras.losses.CategoricalCrossentropy()

    snapshots = [(0, img.copy().squeeze(), None)]  # Initial image, no loss yet

    for i in range(1, steps + 1):
        with tf.GradientTape() as tape:
            output = model(optimized_img)
            loss = loss_fn(target_output, output)
            reg_loss = tf.reduce_sum(tf.square(optimized_img))
            total_loss = loss + 1e-4 * reg_loss

        grads = tape.gradient(total_loss, optimized_img)
        optimizer.apply_gradients([(grads, optimized_img)])
        optimized_img.assign(tf.clip_by_value(optimized_img, 0.0, 1.0))

        if i % snapshot_interval == 0 or i == steps:
            snapshots.append((i, optimized_img.numpy().squeeze(), total_loss.numpy()))

    return snapshots
```

Figure 6: Input Image Optimization Algorithm

```python
def optimize_input_image(model,
                         digits,
                         sample_from_dataset=False,
                         x=None,
                         y=None,
                         steps=500,
                         lr=0.05,
                         noise_scale=0.3,
                         seed=0,
                         snapshot_interval=100,
                         plot=True):

    # Create target output vector
    target_output = create_target_output(digits)

    # Handle image initialization
    if not sample_from_dataset:
        initial_img = create_random_gray_image()
    else:
        assert x is not None and y is not None, "x and y must be provided when sampling from dataset."
        initial_img = sample_random_image_from_dataset(
            x_train=x,
            y_train=y,
            target_digit=digits,
            seed=seed,
            add_noise=True,
            noise_scale=noise_scale)

    # Optimize
    snapshots = optimize_with_snapshots(
        model=model,
        target_output=target_output,
        img=initial_img,
        steps=steps,
        lr=lr,
        snapshot_interval=snapshot_interval
    )

    if plot:
        plot_snapshots(snapshots, main_title=f"Optimization Toward Digits {digits}")

    return snapshots
```

Figure 7: High-Level Wrapper for Input Initialization and Optimization Workflow

```python
# Prepare target output and initial image
target_digit = 2
target_output = create_target_output(digits=target_digit)

# Get real image with added noise
initial_img = sample_random_image_from_dataset(x_train, y_train, target_digit=target_digit, add_noise=True)

# Run optimization
snapshots = optimize_with_snapshots(model, target_output, initial_img, steps=500, snapshot_interval=100)

# Plot snapshots
plot_snapshots(snapshots, main_title=f"Optimization Toward Digit {target_digit}")
```

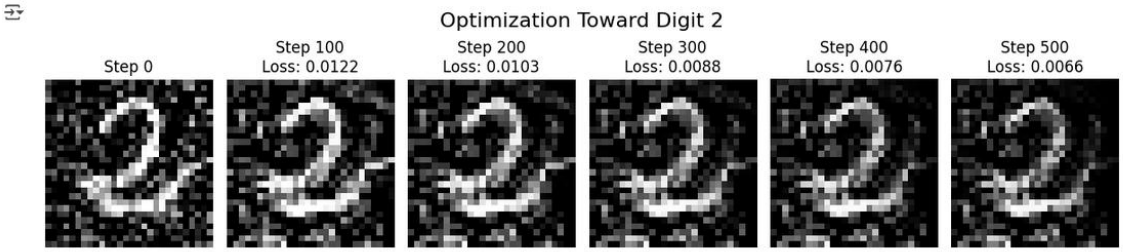


Figure 8: Sample Optimization Result for Digit 2

Results

1) Single Digit Optimization from Noise

- Target vector: [0,0,1,0,0,0,0,0,0,0] (Digit 2)
- Initial image: random grayscale noise
- Final optimized image revealed partial digit-like structures such as **curves** and **lines**, but did not resemble a clearly identifiable digit 2 as seen in original samples.

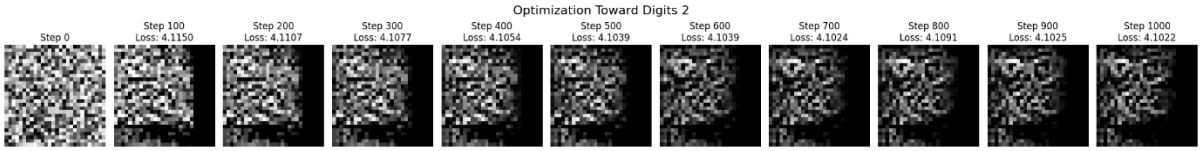


Figure 9: Optimization for Digit 2 Starting from Random Image

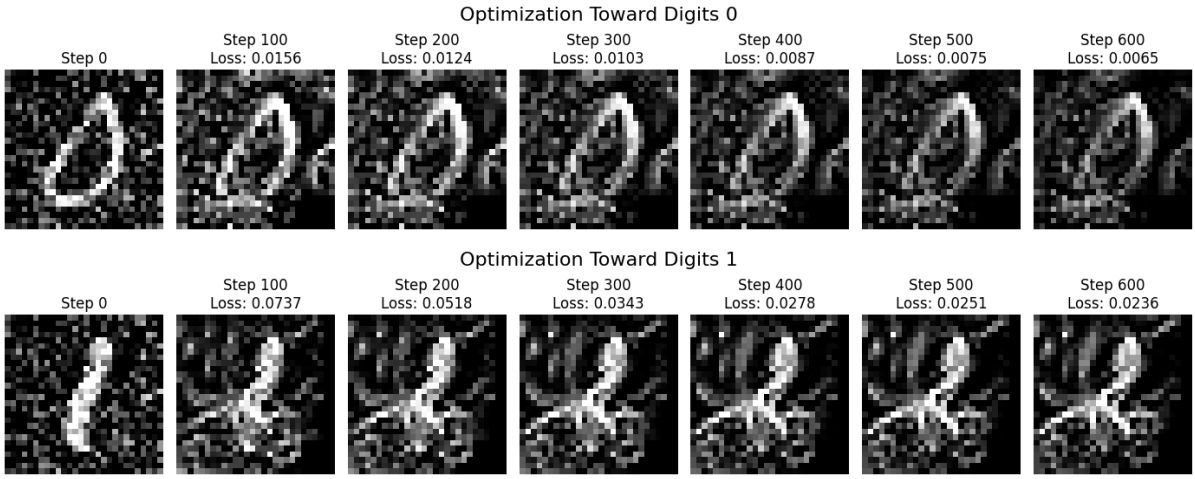
2) Optimization from Noisy Real Image

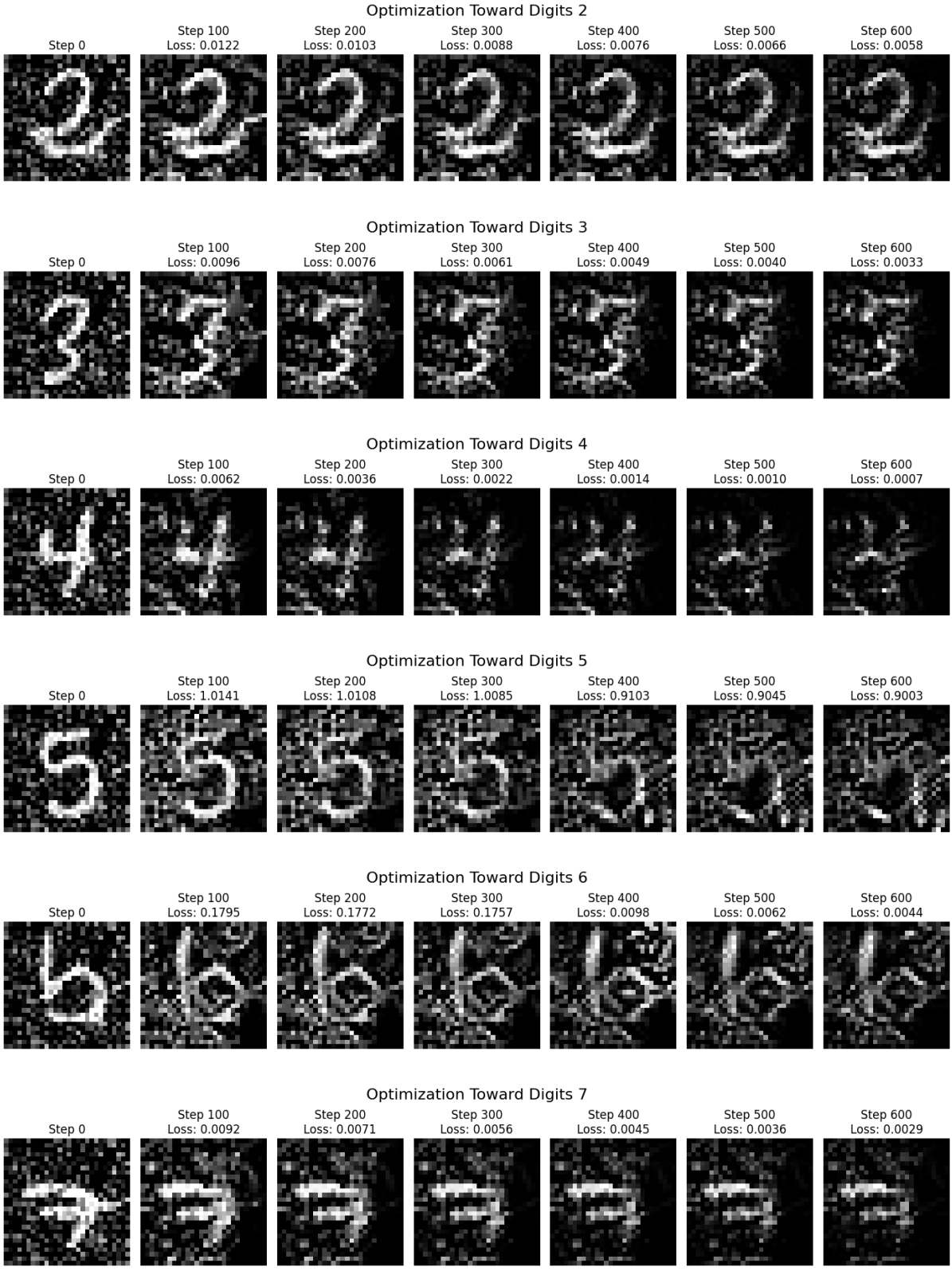
- Initial image: a real digit sample corrupted with Gaussian noise
- Optimized image retained the original digit's structure while emphasizing features relevant to the target class

Figure 10: Optimization for Digit 2 Starting from Noisy Real Image

3) Optimization for All Digits (0–9) from Noisy Real Images

- For each digit, input optimization was repeated using noisy real images.
- Digits 1, 2, 6, and 8 produced more human-recognizable results when optimized from noisy real images.
- Particularly for digit 8, the optimization from a random image yielded an interesting result.




Optimization Toward Digits 2
Step 0
Step 100
Loss: 0.0122
Step 200
Loss: 0.0103
Step 300
Loss: 0.0088
Step 400
Loss: 0.0076
Step 500
Loss: 0.0066
Step 600
Loss: 0.0058
Optimization Toward Digits 3
Step 0
Step 100
Loss: 0.0096
Step 200
Loss: 0.0076
Step 300
Loss: 0.0061
Step 400
Loss: 0.0049
Step 500
Loss: 0.0040
Step 600
Loss: 0.0033
Optimization Toward Digits 4
Step 0
Step 100
Loss: 0.0062
Step 200
Loss: 0.0036
Step 300
Loss: 0.0022
Step 400
Loss: 0.0014
Step 500
Loss: 0.0010
Step 600
Loss: 0.0007
Optimization Toward Digits 5
Step 0
Step 100
Loss: 1.0141
Step 200
Loss: 1.0108
Step 300
Loss: 1.0085
Step 400
Loss: 0.9103
Step 500
Loss: 0.9045
Step 600
Loss: 0.9003
Optimization Toward Digits 6
Step 0
Step 100
Loss: 0.1795
Step 200
Loss: 0.1772
Step 300
Loss: 0.1757
Step 400
Loss: 0.0098
Step 500
Loss: 0.0062
Step 600
Loss: 0.0044
Optimization Toward Digits 7
Step 0
Step 100
Loss: 0.0092
Step 200
Loss: 0.0071
Step 300
Loss: 0.0056
Step 400
Loss: 0.0045
Step 500
Loss: 0.0036
Step 600
Loss: 0.0029

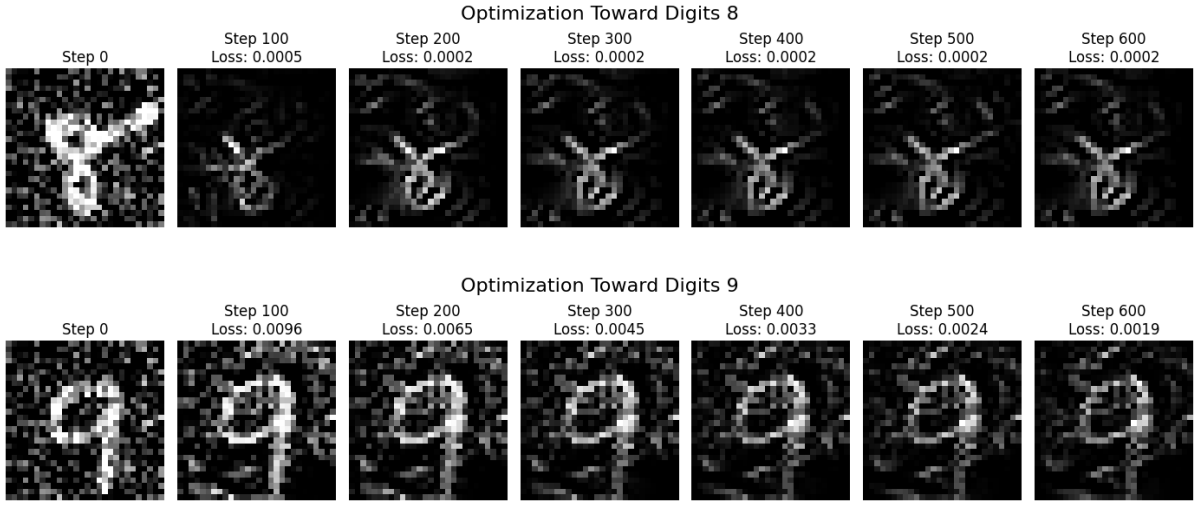
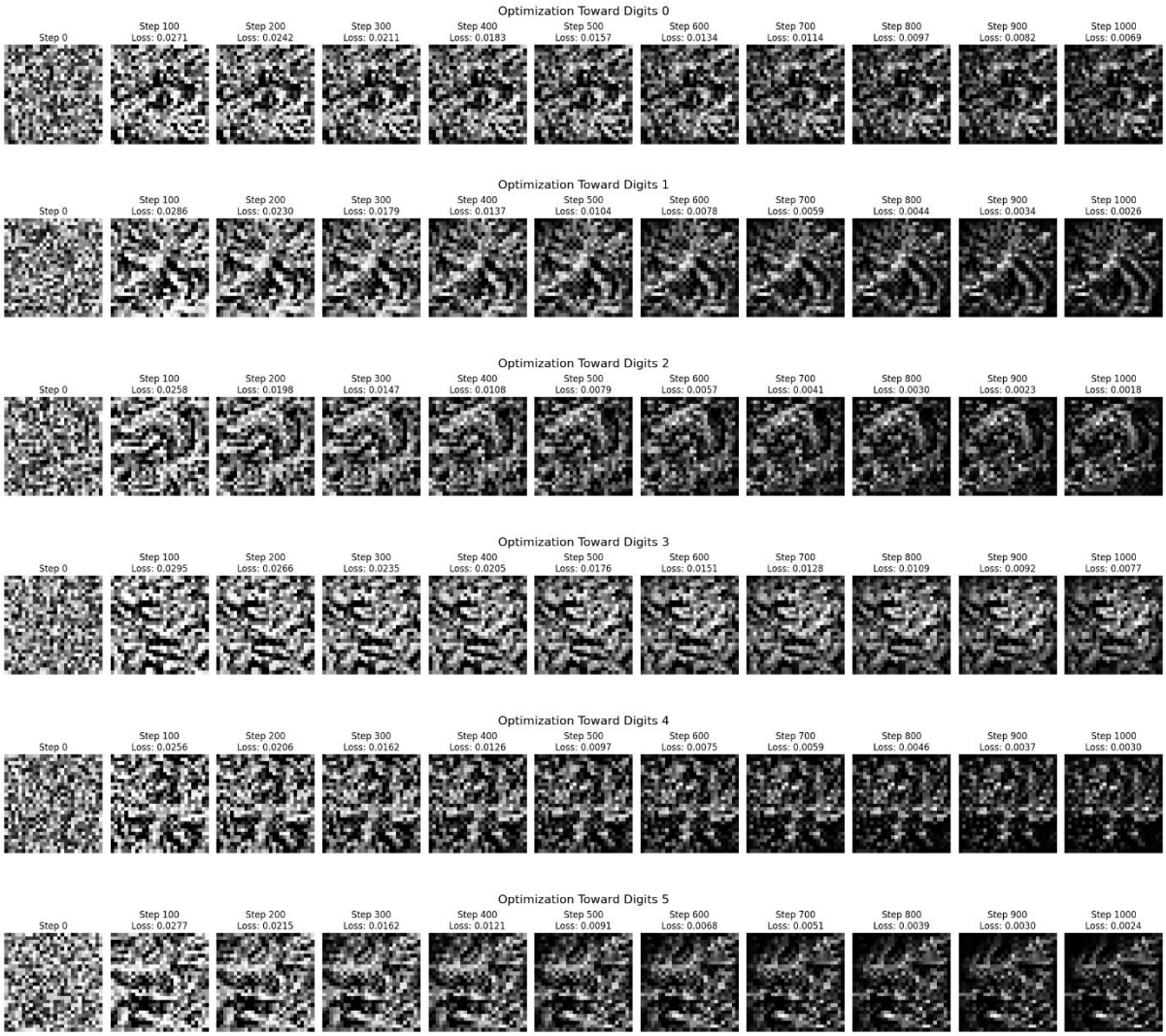


Figure 11: Optimization for All Digits Using Noisy Real Images

4) Optimization for All Digits (0–9) from Random Images

- For each digit, input optimization was repeated using pure noisy images.
- Particularly for digit 8, the optimization from a random image yielded an interesting result.



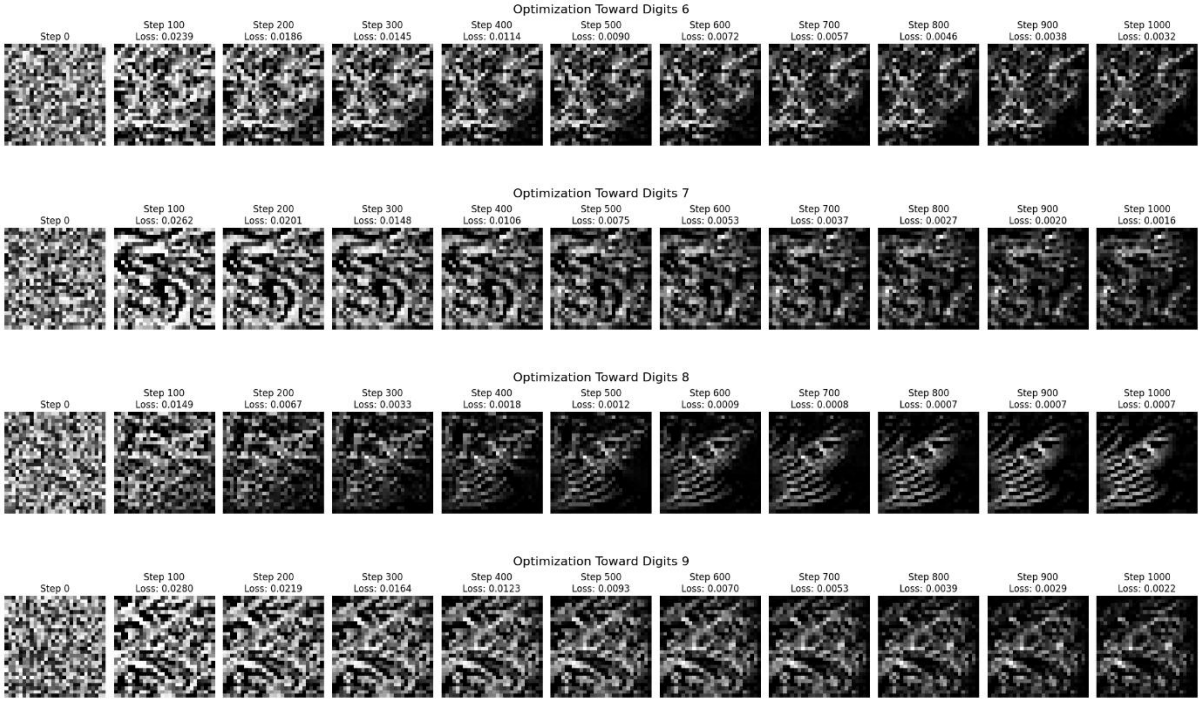


Figure 12: Optimization for All Digits Using Random Images

5) Mixed Digit Optimization

- Target vector: [0,0,0.5,0.5,0,0,0,0,0,0] indicating equal preference for digits 2 and 3
- The optimized images exhibited hybrid characteristics, such as the upper curve of 2 blended with the lower half of 3.

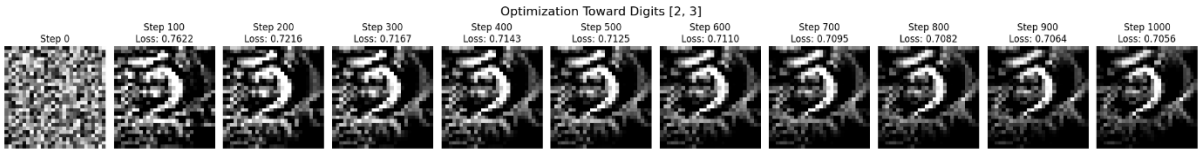


Figure 13: Optimization of Random Image Toward a Mix of Digits 2 and 3

Figure 14: Optimization of Random Image Toward a Mix of Digits 1 and 7

6) Digit Conversion

- Conversion of a real image of digit x to digit y.
- The optimization process successfully produced visually recognizable transformations, with the final images closely resembling the target classes. For example, the transformation from digit 5 to digit 9 produced a clear and human-recognizable image of digit 9. Similar transformations for other digits also resulted in coherent outputs.

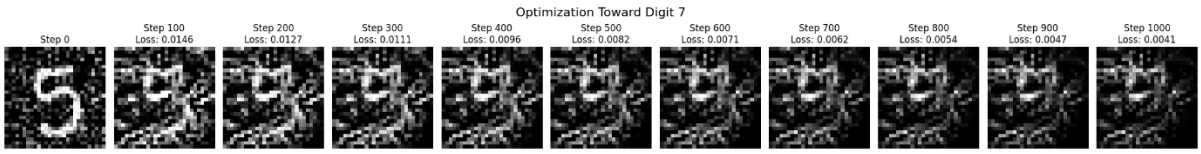


Figure 15: Optimization of Real Image of Digit 5 Toward Digit 7

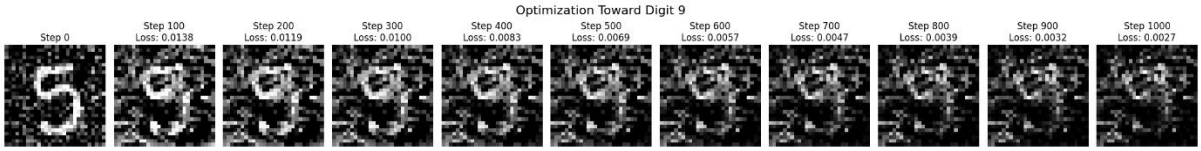


Figure 16: Optimization of Real Image of Digit 5 Toward Digit 9

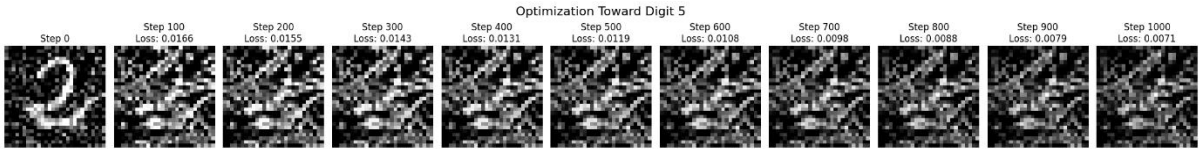


Figure 17: Optimization of Real Image of Digit 2 Toward Digit 5

Comments

- Initialization strategy had a significant impact on convergence and visual clarity; initializing with **noisy real images** generally produced more structured and interpretable outputs compared to pure noise.
- Regularization through **L2 penalty** and **pixel clipping** effectively constrained pixel values, reducing artifacts and enhancing output interpretability.
- Intermediate snapshots revealed the trajectory of optimization, showing how input features gradually evolved toward activating the desired class.
- Optimized inputs often **lacked human-recognizable structure**, especially when initialized from noise or targeting mixed classes, highlighting a disconnect between model confidence and visual plausibility.
- Many generated images resembled **adversarial** examples—unrecognizable to humans but confidently classified by the model. These inputs tended to **distribute class-specific textures across** the image rather than **forming coherent digit shapes**. This might be controlled with more strong regularizations.
- This adversarial-like behaviour underscores a key limitation of CNNs: model confidence can be exploited in ways that defy human intuition, exposing vulnerabilities in their internal representation.
- Mixed-class optimization illustrated the network's ability to represent and interpolate between digit concepts, further supporting the notion of a continuous latent space in deep models.