Notes
Perceptron vs ANN vs CNN on MNIST Dataset

Ankit Kumar

February 4, 2026

Contents

1 Introduction and Theoretical Foundations

1.1 Evolution of Neural Networks

The journey from perceptrons to deep learning represents one of the most significant developments in artificial intelligence:

- **1957 - Perceptron**: Single-layer linear classifier by Frank Rosenblatt
- **1960s-1970s**: AI winter due to perceptron limitations (XOR problem)
- **1980s**: Backpropagation algorithm enables multi-layer networks
- **1990s**: Convolutional Neural Networks for image processing
- **2000s-present**: Deep learning revolution with improved hardware and algorithms

1.2 Mathematical Foundations

1.2.1 Perceptron Mathematical Model

For input vector $\mathbf{x} = [x_1, x_2, ..., x_n]$, weights $\mathbf{w} = [w_1, w_2, ..., w_n]$, and bias b:

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right) \tag{1}$$

where f is typically a step function: $f(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$

1.2.2 Multi-layer Neural Networks

For a network with L layers:

$$\mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)} \tag{2}$$

$$\mathbf{a}^{(l)} = f(\mathbf{z}^{(l)}) \tag{3}$$

where $\mathbf{a}^{(0)} = \mathbf{x}$ is the input.

1.2.3 Backpropagation Algorithm

The weight update rule using gradient descent:

$$\mathbf{W}^{(l)} := \mathbf{W}^{(l)} - \alpha \frac{\partial J}{\partial \mathbf{W}^{(l)}} \tag{4}$$

where α is the learning rate and J is the cost function.

Function	Formula	Range	Use Case
Sigmoid	$\sigma(x) = \frac{1}{1+e^{-x}}$	(0,1)	Output layer, probability
Tanh	$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	(-1,1)	Hidden layers
ReLU	$\max(0, x)$	[0,)	Hidden layers (prevents vanishing gradient)
Softmax	$\frac{e^{x_i}}{\sum_j e^{x_j}}$	(0,1), sums to 1	Multi-class output

Table 1: Common Activation Functions

1.3 Activation Functions Comparison

2 Implementation Setup and Environment

2.1 Complete Import Statements

```
# ============================================
# BASIC IMPORTS FOR DATA MANIPULATION
# ============================================
import numpy as np  # Numerical computations
import pandas as pd  # Data handling
import seaborn as sns  # Statistical visualization
import matplotlib.pyplot as plt  # Plotting

# Suppress warnings for cleaner output
import warnings
warnings.filterwarnings('ignore')

# ============================================
# SCIKIT-LEARN IMPORTS
# ============================================
from sklearn.preprocessing import LabelEncoder, StandardScaler
# LabelEncoder: Converts categorical labels to numerical
# StandardScaler: Standardizes features by removing mean and scaling

from sklearn.model_selection import train_test_split
# Splits data into training and testing sets

from sklearn.linear_model import Perceptron
# Classical perceptron implementation from scikit-learn

from sklearn.metrics import accuracy_score, confusion_matrix,
    classification_report
# accuracy_score: Calculates accuracy of predictions
# confusion_matrix: Shows correct/incorrect classifications
# classification_report: Detailed performance metrics

# ============================================
# TENSORFLOW/KERAS IMPORTS
# ============================================
from tensorflow.keras.models import Sequential
# Sequential model: Linear stack of layers

from tensorflow.keras.layers import Dense, Conv2D, Flatten,
    MaxPooling2D, Dropout
# Dense: Fully connected layer
# Conv2D: 2D convolutional layer
```

```python
# Flatten: Converts multi-dimensional input to 1D
# MaxPooling2D: Max pooling operation
# Dropout: Regularization layer that randomly sets inputs to zero

from tensorflow.keras.utils import to_categorical
# Converts class vectors to binary class matrix (one-hot encoding)
```

Listing 1: Complete Python Imports

2.2 Data Loading and Exploration

```python
# Load MNIST dataset from CSV files
df = pd.read_csv('/content/sample_data/mnist_train_small.csv')
df_test = pd.read_csv('/content/sample_data/mnist_test.csv')

# Display dataset shapes
print("Training Data Shape:", df.shape)  # Expected: (19999, 785)
print("Test Data Shape:", df_test.shape)  # Expected: (10000, 785)

# Explanation of the dataset structure:
# - 785 columns: 1 label column + 784 pixel columns (28 28  image)
# - Each pixel value ranges from 0 to 255 (grayscale intensity)
# - Labels: 0-9 representing handwritten digits

# Display column names (first 10)
print("\nFirst 10 column names:")
print(df.columns[:10])
# Note: The CSV doesn't have proper column headers,
# so pandas assigns default numeric names

# Extract features and labels
# Since columns don't have proper names, we use .iloc for positional
    indexing
# Column 0: Label (digit 0-9)
# Columns 1-784: Pixel values
x_train = df.iloc[:, 1:].values  # All rows, columns 1 to end (pixels)
y_train = df.iloc[:, 0].values   # All rows, column 0 (labels)

x_test = df_test.iloc[:, 1:].values
y_test = df_test.iloc[:, 0].values

print(f"\nTraining data shape: {x_train.shape}")  # (19999, 784)
print(f"Training labels shape: {y_train.shape}")  # (19999,)
print(f"Test data shape: {x_test.shape}")        # (10000, 784)
print(f"Test labels shape: {y_test.shape}")      # (10000,)
```

Listing 2: Loading and Exploring MNIST Dataset

2.3 Data Preprocessing Steps

```python
# ============================================
# STEP 1: DATA NORMALIZATION
# ============================================
# Convert to float32 and normalize pixel values from [0,255] to [0,1]
# Why normalize?
```

```python
# 1. Prevents numerical instability during training
# 2. Helps gradient descent converge faster
# 3. Ensures consistent scale across features
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0

print("Normalized pixel value range:")
print(f"Min: {x_train.min()}, Max: {x_train.max()}")
print(f"Mean: {x_train.mean():.4f}, Std: {x_train.std():.4f}")

# ============================================
# STEP 2: RESHAPING FOR VISUALIZATION
# ============================================
# Reshape from (samples, 784) to (samples, 28, 28) for 2D representation
# This doesn't change the data, just reorganizes it for easier handling
x_train_img = x_train.reshape(-1, 28, 28)  # -1 means infer this dimension
x_test_img = x_test.reshape(-1, 28, 28)

print(f"\nReshaped training data: {x_train_img.shape}")  # (19999, 28, 28)
print(f"Reshaped test data: {x_test_img.shape}")        # (10000, 28, 28)

# Verify reshaping by looking at first image
print(f"\nFirst image shape: {x_train_img[0].shape}")
print(f"First image min pixel: {x_train_img[0].min()}")
print(f"First image max pixel: {x_train_img[0].max()}")

# ============================================
# STEP 3: LABEL ENCODING (ONE-HOT ENCODING)
# ============================================
# Convert integer labels (0-9) to one-hot encoded vectors
# Example: label 3 becomes [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
# Why one-hot encoding?
# 1. Required for categorical cross-entropy loss
# 2. Each class gets its own output neuron
# 3. Better for multi-class classification
y_train_cat = to_categorical(y_train, 10)  # 10 classes (digits 0-9)
y_test_cat = to_categorical(y_test, 10)

print(f"\nOriginal label 0: {y_train[0]}")
print(f"One-hot encoded label 0: {y_train_cat[0]}")
print(f"One-hot encoded shape: {y_train_cat.shape}")  # (19999, 10)

# Verify one-hot encoding
print(f"\nSum of one-hot vector (should be 1): {y_train_cat[0].sum()}")
print(f"Number of unique labels: {len(np.unique(y_train))}")
```

Listing 3: Data Preprocessing Pipeline

3 Perceptron Implementation

3.1 Perceptron Architecture Design

The perceptron implemented here is a single-layer neural network with:

- **Input**: 784 neurons (flattened 28×28 image)
- **Output**: 10 neurons with softmax activation (one per digit class)
- **Total Parameters**: 7,850 (784×10 weights + 10 biases)
- **Activation**: Softmax for multi-class probability distribution

```
# ============================================
# PERCEPTRON MODEL DEFINITION
# ============================================
# Using Keras Sequential API for simple linear stack of layers
Perceptron = Sequential([
    # Layer 1: Flatten layer
    # Converts 28 28  input image to 1D vector of 784 elements
    # Required because Dense layers expect 1D input
    Flatten(input_shape=(28,28)),  # Output shape: (batch_size, 784)

    # Layer 2: Dense (fully connected) output layer
    # 10 neurons for 10 digit classes (0-9)
    # softmax activation converts outputs to probability distribution
    # softmax ensures all outputs sum to 1
    Dense(10, activation='softmax')  # Output shape: (batch_size, 10)
])

# Display model architecture summary
print("="*50)
print("PERCEPTRON MODEL SUMMARY")
print("="*50)
Perceptron.summary()

# Expected output:
# Total params: 7,850
# Trainable params: 7,850
# Non-trainable params: 0
```

Listing 4: Perceptron Model Definition

3.2 Model Compilation

```
# ============================================
# MODEL COMPILATION
# ============================================
# Compile configures the model for training
Perceptron.compile(
    optimizer='sgd',  # Stochastic Gradient Descent
    # SGD updates weights after each training example
    # Simple but effective for linear models

```

```
    loss='categorical_crossentropy',  # Loss function for multi-class
    # Measures difference between predicted and true distributions
    # Formula: -    y_true * log(y_pred)

    metrics=['accuracy']  # Evaluation metric to monitor
    # Accuracy = (correct predictions) / (total predictions)
)

# Alternative optimizer configurations:
# optimizer='sgd' - basic stochastic gradient descent
# optimizer=tf.keras.optimizers.SGD(learning_rate=0.01, momentum=0.9)
# optimizer='adam' - adaptive moment estimation (better for complex models)
```

Listing 5: Perceptron Model Compilation

3.3 Model Training

```
# ============================================
# MODEL TRAINING
# ============================================
# Fit method trains the model on training data
history_precp = Perceptron.fit(
    x_train_img,      # Training features (images)
    y_train_cat,      # Training labels (one-hot encoded)

    epochs=10,        # Number of complete passes through training data
    # Each epoch = one forward pass + one backward pass of ALL training samples
    # More epochs = more learning, but risk of overfitting

    batch_size=32,    # Number of samples per gradient update
    # Smaller batches: more frequent updates, noisier gradient
    # Larger batches: smoother gradient, more memory required

    validation_data=(x_test_img, y_test_cat),  # Validation/test data
    # Used to evaluate loss and metrics after each epoch
    # Not used for training, only for monitoring

    verbose=1         # Verbosity mode (1 = progress bar)
)

# Training process explanation:
# 1. Forward pass: Compute predictions
# 2. Compute loss: Compare predictions with true labels
# 3. Backward pass: Compute gradients using backpropagation
# 4. Update weights: Adjust weights using optimizer
# 5. Repeat for all batches in epoch
# 6. Repeat for all epochs
```

Listing 6: Perceptron Training Process

3.4 Training Output Analysis

The perceptron training shows typical learning behavior:

```
Epoch 1/10:  accuracy: 0.5885  loss: 1.5343  val_accuracy: 0.8497
Epoch 2/10:  accuracy: 0.8489  loss: 0.6749  val_accuracy: 0.8720
Epoch 3/10:  accuracy: 0.8715  loss: 0.5393  val_accuracy: 0.8818
Epoch 4/10:  accuracy: 0.8784  loss: 0.4835  val_accuracy: 0.8903
Epoch 5/10:  accuracy: 0.8868  loss: 0.4465  val_accuracy: 0.8932
Epoch 6/10:  accuracy: 0.8880  loss: 0.4243  val_accuracy: 0.8964
Epoch 7/10:  accuracy: 0.8884  loss: 0.4136  val_accuracy: 0.8986
Epoch 8/10:  accuracy: 0.8948  loss: 0.3957  val_accuracy: 0.9005
Epoch 9/10:  accuracy: 0.8949  loss: 0.3912  val_accuracy: 0.9025
Epoch 10/10: accuracy: 0.8976  loss: 0.3759  val_accuracy: 0.9032
```

3.4.1 Key Observations

- **Rapid initial learning**: Accuracy jumps from 58.85% to 84.89% in first epoch
- **Gradual improvement**: Slower learning in later epochs
- **Good generalization**: Validation accuracy (90.32%) close to training accuracy (89.76%)
- **Stable convergence**: Loss steadily decreases without oscillations

3.5 Performance Evaluation

```python
# ============================================
# PERFORMANCE EVALUATION
# ============================================
# Evaluate model on test data
test_loss, test_accuracy = Perceptron.evaluate(x_test_img, y_test_cat,
    verbose=0)
print("\n" + "="*50)
print("PERCEPTRON FINAL EVALUATION")
print("="*50)
print(f"Test Loss: {test_loss:.4f}")
print(f"Test Accuracy: {test_accuracy:.4f} ({test_accuracy*100:.2f}%)")

# Alternative evaluation methods:
# 1. Get predictions
predictions = Perceptron.predict(x_test_img)
predicted_classes = np.argmax(predictions, axis=1)
true_classes = np.argmax(y_test_cat, axis=1)

# 2. Calculate accuracy manually
correct_predictions = np.sum(predicted_classes == true_classes)
total_predictions = len(true_classes)
manual_accuracy = correct_predictions / total_predictions
print(f"\nManual Accuracy Calculation:")
print(f"Correct: {correct_predictions}/{total_predictions}")
print(f"Accuracy: {manual_accuracy:.4f}")

# 3. Confusion Matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(true_classes, predicted_classes)
print(f"\nConfusion Matrix Shape: {cm.shape}")
```

```python
print("Confusion Matrix (first 5x5):")
print(cm[:5, :5])
```

Listing 7: Perceptron Evaluation

4 Artificial Neural Network (ANN) Implementation

4.1 ANN Architecture Design

The ANN extends the perceptron with hidden layers:

- **Input**: 784 neurons (flattened image)
- **Hidden Layer 1**: 128 neurons with ReLU activation
- **Hidden Layer 2**: 64 neurons with ReLU activation
- **Output**: 10 neurons with softmax activation
- **Total Parameters**: 109,386

```python
# ============================================
# ARTIFICIAL NEURAL NETWORK (ANN) DEFINITION
# ============================================
ann = Sequential([
    # Layer 1: Flatten input
    Flatten(input_shape=(28,28)),  # (batch_size, 784)

    # Layer 2: First hidden layer
    Dense(128, activation='relu'),  # (batch_size, 128)
    # 128 neurons with ReLU activation
    # ReLU: f(x) = max(0, x)
    # Advantages: Computationally efficient, mitigates vanishing gradient

    # Layer 3: Second hidden layer
    Dense(64, activation='relu'),   # (batch_size, 64)
    # 64 neurons with ReLU activation
    # Creates deeper feature representations

    # Layer 4: Output layer
    Dense(10, activation='softmax') # (batch_size, 10)
    # 10 neurons for digit classification
])

# Calculate total parameters:
# Layer 1 (Flatten): 0 parameters (only reshapes)
# Layer 2 (Dense 128): (784 128 ) weights + 128 biases = 100,480
# Layer 3 (Dense 64): (128 64 ) weights + 64 biases = 8,256
# Layer 4 (Dense 10): (64 10 ) weights + 10 biases = 650
# Total: 100,480 + 8,256 + 650 = 109,386

print("="*50)
print("ARTIFICIAL NEURAL NETWORK SUMMARY")
print("="*50)
```

```
ann.summary()
```

Listing 8: ANN Model Definition

4.2 ANN Compilation with Adam Optimizer

```
# ============================================
# ANN COMPILATION
# ============================================
ann.compile(
    optimizer='adam',  # Adaptive Moment Estimation
    # Adam combines benefits of two extensions of SGD:
    # 1. Adaptive Gradient Algorithm (AdaGrad): Maintains per-parameter
        learning rates
    # 2. Root Mean Square Propagation (RMSProp): Maintains moving
       average of squared gradients
    # Advantages: Faster convergence, less sensitive to learning rate

    loss='categorical_crossentropy',
    metrics=['accuracy']
)

# Adam optimizer parameters (default values):
# learning_rate=0.001
# beta_1=0.9 (exponential decay rate for first moment estimates)
# beta_2=0.999 (exponential decay rate for second moment estimates)
# epsilon=1e-07 (numerical stability term)

# Custom Adam optimizer:
# from tensorflow.keras.optimizers import Adam
# optimizer=Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999)
```

Listing 9: ANN Compilation with Adam Optimizer

4.3 ANN Training

```
# ============================================
# ANN TRAINING
# ============================================
history_ann = ann.fit(
    x_train_img,
    y_train_cat,
    epochs=10,
    batch_size=32,
    validation_data=(x_test_img, y_test_cat),
    verbose=1
)

# Training behavior expectations:
# 1. Faster convergence than perceptron due to Adam optimizer
# 2. Higher capacity to learn complex patterns (more parameters)
# 3. Risk of overfitting due to increased model complexity
# 4. Better feature learning through hidden layers
```

Listing 10: ANN Training Process

4.4 ANN Training Output Analysis

```
Epoch 1/10:  accuracy: 0.8126  loss: 0.6513  val_accuracy: 0.9403
Epoch 2/10:  accuracy: 0.9525  loss: 0.1713  val_accuracy: 0.9509
Epoch 3/10:  accuracy: 0.9674  loss: 0.1141  val_accuracy: 0.9586
Epoch 4/10:  accuracy: 0.9753  loss: 0.0806  val_accuracy: 0.9607
Epoch 5/10:  accuracy: 0.9796  loss: 0.0632  val_accuracy: 0.9637
Epoch 6/10:  accuracy: 0.9867  loss: 0.0446  val_accuracy: 0.9689
Epoch 7/10:  accuracy: 0.9900  loss: 0.0334  val_accuracy: 0.9672
Epoch 8/10:  accuracy: 0.9941  loss: 0.0219  val_accuracy: 0.9667
Epoch 9/10:  accuracy: 0.9952  loss: 0.0196  val_accuracy: 0.9644
Epoch 10/10: accuracy: 0.9944  loss: 0.0164  val_accuracy: 0.9598
```

4.4.1 Key Observations

- **Faster convergence**: Reaches 94.03% validation accuracy in just 1 epoch
- **Higher training accuracy**: 99.44% vs perceptron's 89.76%
- **Better generalization**: 95.98% validation accuracy vs 90.32%
- **Overfitting signs**: Training accuracy (99.44%) ¿ Validation accuracy (95.98%)
- **Lower loss**: 0.1620 vs perceptron's 0.3613

4.5 ANN Evaluation

```python
# ============================================
# ANN EVALUATION
# ============================================
acc_ann = ann.evaluate(x_test_img, y_test_cat, verbose=0)[1]
print("\n" + "="*50)
print("ARTIFICIAL NEURAL NETWORK FINAL EVALUATION")
print("="*50)
print(f"Test Accuracy: {acc_ann:.4f} ({acc_ann*100:.2f}%)")

# Performance comparison with perceptron
print(f"\nPerformance Improvement over Perceptron:")
print(f"Accuracy increase: {(acc_ann - acc_precp)*100:.2f}%")
print(f"Relative improvement: {((acc_ann/acc_precp)-1)*100:.2f}%")
```

Listing 11: ANN Performance Evaluation

5 Convolutional Neural Network (CNN) Implementation

5.1 Data Preparation for CNN

CNNs require 4D input tensors: (samples, height, width, channels)

```python
# ============================================
# DATA PREPARATION FOR CNN
# ============================================
# CNN expects 4D input: (batch_size, height, width, channels)
# MNIST images are grayscale, so channels=1
X_train_cnn = x_train.reshape(-1, 28, 28, 1)  # Shape: (19999, 28, 28, 1)
X_test_cnn = x_test.reshape(-1, 28, 28, 1)    # Shape: (10000, 28, 28, 1)

print("CNN Input Shapes:")
print(f"Training: {X_train_cnn.shape}")
print(f"Test: {X_test_cnn.shape}")

# Explanation of dimensions:
# Dimension 0: Number of samples (19999 for training)
# Dimension 1: Height (28 pixels)
# Dimension 2: Width (28 pixels)
# Dimension 3: Channels (1 for grayscale, 3 for RGB)

# For color images, you would reshape to (samples, height, width, 3)
# Example: X_train_cnn = x_train.reshape(-1, 28, 28, 3)
```

Listing 12: Data Reshaping for CNN

5.2 CNN Architecture Design

```python
# ============================================
# CONVOLUTIONAL NEURAL NETWORK DEFINITION
# ============================================
cnn = Sequential([
    # ===== CONVOLUTIONAL BLOCK 1 =====
    # Layer 1: First convolutional layer
    Conv2D(32, kernel_size=(3,3), activation='relu', input_shape=(28,28,1)),
    # 32 filters, each 3 3
    # Input: (batch_size, 28, 28, 1)
    # Output: (batch_size, 26, 26, 32) [28-3+1=26, no padding]
    # Parameters: (3   3132   ) + 32 = 320

    # Layer 2: First pooling layer
    MaxPooling2D(pool_size=(2,2)),
    # Max pooling with 2 2 window, stride=2
    # Input: (batch_size, 26, 26, 32)
    # Output: (batch_size, 13, 13, 32) [26/2=13]
    # Reduces spatial dimensions by half

    # ===== CONVOLUTIONAL BLOCK 2 =====
    # Layer 3: Second convolutional layer
    Conv2D(64, kernel_size=(3,3), activation='relu'),
    # 64 filters, each 3 3
    # Input: (batch_size, 13, 13, 32)
    # Output: (batch_size, 11, 11, 64) [13-3+1=11]
    # Parameters: (3   33264   ) + 64 = 18,496

    # Layer 4: Second pooling layer
```

```
    MaxPooling2D(pool_size=(2,2)),
    # Input: (batch_size, 11, 11, 64)
    # Output: (batch_size, 5, 5, 64) [11/2=5.5     5]
    # Note: 11/2 = 5.5, integer division gives 5

    # ===== FLATTEN AND DENSE LAYERS =====
    # Layer 5: Flatten layer
    Flatten(),
    # Converts 3D feature maps to 1D vector
    # Input: (batch_size, 5, 5, 64)
    # Output: (batch_size, 1600) [5  564  =1600]

    # Layer 6: Fully connected layer
    Dense(128, activation='relu'),
    # 128 neurons with ReLU activation
    # Parameters: (1600 128 ) + 128 = 204,928

    # Layer 7: Dropout for regularization
    Dropout(0.5),
    # Randomly sets 50% of inputs to zero during training
    # Prevents overfitting by preventing co-adaptation of neurons
    # No parameters

    # Layer 8: Output layer
    Dense(10, activation='softmax')
    # 10 neurons for digit classification
    # Parameters: (128 10 ) + 10 = 1,290
])

# Total parameters calculation:
# Conv2D 1: 320
# Conv2D 2: 18,496
# Dense 1: 204,928
# Dense 2: 1,290
# Total: 225,034

print("="*50)
print("CONVOLUTIONAL NEURAL NETWORK SUMMARY")
print("="*50)
cnn.summary()
```

Listing 13: CNN Model Definition

5.3 Convolution Operation Details

5.3.1 Convolution Mathematics

For a single filter F of size $k \times k$ applied to input I:

$$G[i,j] = \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} F[m,n] \cdot I[i+m, j+n] + b$$

5.3.2 Output Size Calculation

For input size $W \times H$, filter size F, padding P, stride S:

$$W_{\text{out}} = \frac{W - F + 2P}{S} + 1$$

In our CNN: $W = 28$, $F = 3$, $P = 0$, $S = 1 \rightarrow W_{\text{out}} = 26$

5.3.3 Pooling Operation

Max pooling selects the maximum value in each window:

$$P[i, j] = \max\{G[Si : Si + F, Sj : Sj + F]\}$$

where S is stride and F is pool size.

5.4 CNN Compilation and Training

```
# ============================================
# CNN COMPILATION
# ============================================
cnn.compile(
    optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

# ============================================
# CNN TRAINING
# ============================================
history_cnn = cnn.fit(
    X_train_cnn,      # 4D tensor for CNN
    y_train_cat,
    epochs=10,
    batch_size=32,
    validation_data=(X_test_cnn, y_test_cat),
    verbose=1
)

# CNN training characteristics:
# 1. Slower per epoch due to convolutional operations
# 2. Better feature extraction for spatial data
# 3. Parameter sharing reduces overfitting risk
# 4. Dropout provides additional regularization
```

Listing 14: CNN Compilation and Training

5.5 CNN Training Output Analysis

```
Epoch 1/10:  accuracy: 0.7595  loss: 0.7384  val_accuracy: 0.9718
Epoch 2/10:  accuracy: 0.9573  loss: 0.1400  val_accuracy: 0.9824
Epoch 3/10:  accuracy: 0.9711  loss: 0.0908  val_accuracy: 0.9832
Epoch 4/10:  accuracy: 0.9764  loss: 0.0711  val_accuracy: 0.9857
```

```
Epoch 5/10:  accuracy: 0.9798  loss: 0.0636  val_accuracy: 0.9853
Epoch 6/10:  accuracy: 0.9807  loss: 0.0554  val_accuracy: 0.9866
Epoch 7/10:  accuracy: 0.9853  loss: 0.0455  val_accuracy: 0.9866
Epoch 8/10:  accuracy: 0.9879  loss: 0.0376  val_accuracy: 0.9890
Epoch 9/10:  accuracy: 0.9906  loss: 0.0312  val_accuracy: 0.9876
Epoch 10/10: accuracy: 0.9903  loss: 0.0309  val_accuracy: 0.9877
```

5.5.1 Key Observations

- **Highest accuracy**: 98.77% validation accuracy (best among all models)
- **Excellent generalization**: 98.77% validation vs 99.03% training
- **Fast convergence**: 97.18% accuracy in first epoch
- **Lowest loss**: 0.0309 training loss vs ANN's 0.0164
- **Stable learning**: Consistent improvement without oscillations

5.6 CNN Evaluation

```
# ============================================
# CNN EVALUATION
# ============================================
acc_cnn = cnn.evaluate(X_test_cnn, y_test_cat, verbose=0)[1]
print("\n" + "="*50)
print("CONVOLUTIONAL NEURAL NETWORK FINAL EVALUATION")
print("="*50)
print(f"Test Accuracy: {acc_cnn:.4f} ({acc_cnn*100:.2f}%)")

# Performance comparison with all models
print(f"\nCOMPREHENSIVE PERFORMANCE COMPARISON:")
print(f"Perceptron Accuracy:  {acc_precp*100:.2f}%")
print(f"ANN Accuracy:         {acc_ann*100:.2f}%")
print(f"CNN Accuracy:         {acc_cnn*100:.2f}%")
print(f"\nImprovement over Perceptron: {(acc_cnn - acc_precp)*100:.2f}%
    ")
print(f"Improvement over ANN:        {(acc_cnn - acc_ann)*100:.2f}%")
```

Listing 15: CNN Performance Evaluation

6 Visualization and Analysis Functions

```
# ============================================
# VISUALIZATION FUNCTION
# ============================================
def plot_training(history, title):
    """
    Plots training and validation accuracy/loss.

    Parameters:
    -----------
    history : keras History object
```

```
        Training history returned by model.fit()
    title : str
        Title for the plots
    """
    plt.figure(figsize=(12, 4))

    # Plot Accuracy
    plt.subplot(1, 2, 1)
    plt.plot(history.history['accuracy'], label="Train", marker='o')
    plt.plot(history.history['val_accuracy'], label="Validation",
        marker='s')
    plt.title(f"{title} - Accuracy")
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.grid(True, alpha=0.3)

    # Plot Loss
    plt.subplot(1, 2, 2)
    plt.plot(history.history['loss'], label="Train", marker='o')
    plt.plot(history.history['val_loss'], label="Validation", marker='s
        ')
    plt.title(f"{title} - Loss")
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.show()

# Usage examples:
plot_training(history_precp, "Perceptron")
plot_training(history_ann, "Artificial Neural Network")
plot_training(history_cnn, "Convolutional Neural Network")

# ============================================
# CONFUSION MATRIX VISUALIZATION
# ============================================
def plot_confusion_matrix(model, x_test, y_test, model_name):
    """
    Plots confusion matrix for model predictions.
    """
    # Get predictions
    predictions = model.predict(x_test)
    predicted_classes = np.argmax(predictions, axis=1)
    true_classes = np.argmax(y_test, axis=1)

    # Calculate confusion matrix
    cm = confusion_matrix(true_classes, predicted_classes)

    # Plot
    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=range(10), yticklabels=range(10))
    plt.title(f'Confusion Matrix - {model_name}')
    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
```

```python
    plt.show()

    # Calculate per-class accuracy
    class_accuracy = cm.diagonal() / cm.sum(axis=1)
    print(f"\nPer-class accuracy for {model_name}:")
    for i, acc in enumerate(class_accuracy):
        print(f"Digit {i}: {acc:.2%}")

# Usage:
# plot_confusion_matrix(Perceptron, x_test_img, y_test_cat, "Perceptron")
# plot_confusion_matrix(ann, x_test_img, y_test_cat, "ANN")
# plot_confusion_matrix(cnn, X_test_cnn, y_test_cat, "CNN")
```

Listing 16: Training Visualization Function

7 Comprehensive Analysis and Comparison

7.1 Parameter Count Analysis

Model	Parameters	Calculation	Efficiency
Perceptron	7,850	$(784 \times 10) + 10$	0.0115 acc/para
ANN	109,386	$(784 \times 128 + 128) + (128 \times 64 + 64) + (64 \times 10 + 10)$	0.000877 acc/par
CNN	225,034	$320 + 18,496 + 204,928 + 1,290$	0.000439 acc/par

Table 2: Parameter Efficiency Analysis

7.2 Performance Metrics Summary

Model	Train Acc	Val Acc	Train Loss	Val Loss
Perceptron	89.76%	90.32%	0.3759	0.3613
ANN	99.44%	95.98%	0.0164	0.1620
CNN	99.03%	98.77%	0.0309	0.0395

Table 3: Complete Performance Comparison

7.3 Training Time Analysis

Model	Time/Epoch	Total Time	Accuracy Gain/Time
Perceptron	2-3 seconds	20-30 seconds	3.01%/second
ANN	3-5 seconds	30-50 seconds	0.192%/second
CNN	18-22 seconds	180-220 seconds	0.0384%/second

Table 4: Training Efficiency Analysis

7.4 Error Analysis by Digit

- **Perceptron Errors**: Frequently confuses:
  - 3 8 (similar rounded shapes)
  - 4 9 (similar top structure)
  - 5 6 (similar curves)
- **ANN Improvements**: Better at distinguishing:
  - 3 vs 8 (learns subtle curvature differences)
  - 4 vs 9 (recognizes open/closed loops)
  - Still struggles with poorly written digits
- **CNN Strengths**: Excellent at:
  - Recognizing digits regardless of position
  - Handling variations in writing style
  - Distinguishing subtle features
  - Generalizing to unseen variations

8 Key Insights and Learnings

8.1 Architectural Insights

1. **Model Complexity vs Performance**:
   - Perceptron: Simple but limited (90.32%)
   - ANN: Good balance (95.98%)
   - CNN: Complex but optimal (98.77%)
2. **Feature Learning Hierarchy**:
   - Perceptron: No feature learning, linear combination
   - ANN: Learns features but ignores spatial relationships
   - CNN: Hierarchical feature learning with spatial awareness
3. **Regularization Needs**:
   - Perceptron: Minimal overfitting due to simplicity
   - ANN: Shows overfitting (needs dropout/L2 regularization)
   - CNN: Dropout effectively prevents overfitting

8.2 Practical Recommendations

1. **When to use each model**:
   - **Perceptron**: Simple binary classification, linearly separable data, limited resources
   - **ANN**: Tabular data, moderate complexity, need for interpretability
   - **CNN**: Image/video data, spatial patterns, maximum accuracy requirement
2. **Training recommendations**:
   - Start with simple models, then increase complexity
   - Use Adam optimizer for faster convergence
   - Implement early stopping to prevent overfitting
   - Use data augmentation for image data
3. **Performance optimization**:
   - Batch size: 32-128 for good balance
   - Learning rate: 0.001 for Adam, adjust based on progress
   - Epochs: Monitor validation loss for early stopping
   - Regularization: Use dropout (0.2-0.5) for complex models

8.3 Limitations and Future Improvements

1. **Current Limitations**:
   - MNIST is relatively simple
   - Models not tested on noisy/varied data
   - No hyperparameter tuning performed
   - Limited to grayscale images
2. **Future Enhancements**:
   - Add batch normalization layers
   - Implement data augmentation
   - Try different architectures (ResNet, VGG)
   - Add attention mechanisms
   - Implement transfer learning
3. **Advanced Techniques to Explore**:
   - Learning rate scheduling
   - Gradient clipping
   - Ensemble methods
   - Bayesian optimization for hyperparameters
   - Explainable AI techniques

9 Conclusion

9.1 Key Takeaways

1. **Architecture matters**: CNN significantly outperforms simpler models for image data
2. **Feature learning**: Hierarchical feature extraction in CNNs provides substantial benefits
3. **Computational trade-off**: More complex models require more resources but deliver better results
4. **Regularization is crucial**: Dropout effectively prevents overfitting in deep networks
5. **Optimizer choice**: Adam provides faster convergence than basic SGD

9.2 Final Performance Ranking

1. **1st**: CNN (98.77%) - Best for image classification
2. **2nd**: ANN (95.98%) - Good general-purpose model
3. **3rd**: Perceptron (90.32%) - Simple baseline model

9.3 Educational Value

This implementation provides:

- Hands-on experience with three fundamental architectures
- Understanding of neural network evolution
- Practical insights into model selection criteria
- Foundation for more advanced deep learning studies
- Template for comparative analysis methodology

9.4 Code Availability

The complete Jupyter notebook with all implementations, visualizations, and analysis is available for further study and experimentation. This serves as a comprehensive reference for understanding neural network architectures and their practical applications.