perplexity

1. Pseudo Code/Algorithm

```
1. Load and preprocess MNIST dataset
   - Normalize pixel values to [0,1]
   - Reshape to (28,28,1) for CNN input
2. Define Encoder:
   - Conv2D(32) → MaxPool → Conv2D(16) → MaxPool
   - Compress 28×28×1 to 7×7×16 (bottleneck)
3. Define Decoder:
   - Conv2D(16) → UpSample → Conv2D(32) → UpSample → Conv2D(1)
   - Reconstruct 7×7×16 back to 28×28×1
4. Compile model with Adam optimizer and binary crossentropy loss
5. Train autoencoder using input images as both input and target
6. Generate reconstructions and visualize results
```

2. Main Program Chunk (Core Code Only)

```
# Data preprocessing
(x_train, _), (x_test, _) = mnist.load_data()
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
x_train = x_train.reshape((x_train.shape[0], 28, 28, 1))
x_test = x_test.reshape((x_test.shape[0], 28, 28, 1))

# Autoencoder architecture
input_img = layers.Input(shape=(28, 28, 1))

# Encoder
x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)
x = layers.MaxPooling2D((2, 2), padding='same')(x)
x = layers.Conv2D(16, (3, 3), activation='relu', padding='same')(x)
encoded = layers.MaxPooling2D((2, 2), padding='same')(x)

# Decoder
x = layers.Conv2D(16, (3, 3), activation='relu', padding='same')(encoded)
x = layers.UpSampling2D((2, 2))(x)
x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = layers.UpSampling2D((2, 2))(x)
decoded = layers.Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

# Model compilation and training
autoencoder = models.Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
autoencoder.fit(x_train, x_train, epochs=10, batch_size=128,
                validation_data=(x_test, x_test))
```

3. Architecture Specifications and Parameters

Parameters

- **Input Shape**: (28, 28, 1) - grayscale MNIST images
- **Optimizer**: Adam
- **Loss Function**: Binary crossentropy
- **Epochs**: 10
- **Batch Size**: 128
- **Learning Rate**: Default Adam (0.001)

Architecture Details

Total Layers: 9 layers (4 encoder + 5 decoder)

Encoder Architecture:

- **Layer 1**: Conv2D - 32 filters, 3×3 kernel, ReLU activation, same padding
- **Layer 2**: MaxPooling2D - 2×2 pool size, same padding
- **Layer 3**: Conv2D - 16 filters, 3×3 kernel, ReLU activation, same padding
- **Layer 4**: MaxPooling2D - 2×2 pool size, same padding (bottleneck: 7×7×16)

Decoder Architecture:

- **Layer 5**: Conv2D - 16 filters, 3×3 kernel, ReLU activation, same padding
- **Layer 6**: UpSampling2D - 2×2 upsampling
- **Layer 7**: Conv2D - 32 filters, 3×3 kernel, ReLU activation, same padding
- **Layer 8**: UpSampling2D - 2×2 upsampling
- **Layer 9**: Conv2D - 1 filter, 3×3 kernel, sigmoid activation, same padding

Architecture Diagram

```
Input (28×28×1)
       ↓
   Conv2D (32) + ReLU
       ↓
   MaxPool2D (2×2)
       ↓ (14×14×32)
   Conv2D (16) + ReLU
       ↓
   MaxPool2D (2×2)
       ↓ (7×7×16)  ← Bottleneck/Latent Space
   Conv2D (16) + ReLU
       ↓
   UpSample2D (2×2)
       ↓ (14×14×16)
   Conv2D (32) + ReLU
```

```
        ↓
    UpSample2D (2×2)
        ↓ (28×28×32)
    Conv2D (1) + Sigmoid
        ↓
    Output (28×28×1)
```

4. Code Explanation

Data Preprocessing

The code normalizes MNIST pixel values to range and reshapes images to (28×28×1) format required for convolutional operations.

Encoder Section

The encoder compresses input images through two convolutional layers (32→16 filters) with max pooling, reducing spatial dimensions from 28×28 to 7×7 while extracting hierarchical features. This creates a compressed latent representation.

Decoder Section

The decoder reconstructs images by reversing the encoder process using upsampling and convolutional layers (16→32→1 filters). The final sigmoid activation ensures output values match the normalized input range .

Training Process

The autoencoder learns unsupervised by using the same images as input and target. Binary crossentropy loss measures reconstruction quality, while Adam optimizer minimizes this loss to learn efficient compression and reconstruction of handwritten digits.

perplexity

1. Pseudo Code/Algorithm

```
1. Load and preprocess MNIST dataset
   - Load training and test data
   - Normalize pixel values to [0,1] range
   - Flatten images from 28×28 to 784-dimensional vectors
2. Define autoencoder architecture:
   - Set encoding dimension to 32 (compression ratio: 784→32)
   - Input layer: 784 neurons (flattened image)
   - Encoder: Dense layer with 32 neurons, ReLU activation
   - Decoder: Dense layer with 784 neurons, sigmoid activation
3. Create three separate models:
   - Autoencoder: Complete input→encoded→decoded pipeline
   - Encoder: Input→encoded (for feature extraction)
   - Decoder: Encoded→decoded (for reconstruction)
4. Compile autoencoder with Adam optimizer and binary crossentropy loss
5. Train for 50 epochs with batch size 256
6. Generate encoded representations and reconstructions for test data
```

2. Main Program Chunk (Core Code Only)

```
# Data preprocessing
(x_train, _), (x_test, _) = keras.datasets.mnist.load_data()
x_train = x_train.astype('float32') / 255.
x_train = x_train.reshape(x_train.shape[0], -1)
x_test = x_test.astype('float32') / 255.
x_test = x_test.reshape(x_test.shape[0], -1)

# Architecture definition
encoding_dim = 32
input_img = keras.Input(shape=(784,))

# Encoder-Decoder layers
encoded = keras.layers.Dense(encoding_dim, activation='relu')(input_img)
decoded = keras.layers.Dense(784, activation='sigmoid')(encoded)

# Model creation
autoencoder = keras.Model(input_img, decoded)
encoder = keras.Model(input_img, encoded)

decoder_input = keras.Input(shape=(encoding_dim,))
decoder_layer = autoencoder.layers[-1]
decoder = keras.Model(decoder_input, decoder_layer(decoder_input))

# Training
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
autoencoder.fit(x_train, x_train, epochs=50, batch_size=256,
                validation_data=(x_test, x_test))
```

3. Architecture Specifications and Parameters

Parameters

- **Input Shape**: (784,) - flattened MNIST images
- **Encoding Dimension**: 32 neurons (bottleneck layer)
- **Optimizer**: Adam
- **Loss Function**: Binary crossentropy
- **Epochs**: 50
- **Batch Size**: 256
- **Compression Ratio**: 784:32 (24.5:1 compression)

Architecture Details

Total Layers: 3 layers (Input + Encoder + Decoder)

Layer Structure:

- **Input Layer**: 784 neurons (28×28 flattened)
- **Encoder Layer**: 32 neurons, ReLU activation, Dense/Fully Connected
- **Decoder Layer**: 784 neurons, Sigmoid activation, Dense/Fully Connected

Model Components:

- **Autoencoder**: Complete model (input→encoded→decoded)
- **Encoder**: Standalone encoder (input→encoded features)
- **Decoder**: Standalone decoder (encoded→reconstructed)

Architecture Diagram

```
Input Layer (784 neurons)
         ↓
    Dense Layer (ReLU)
         ↓
   Encoded/Bottleneck (32 neurons) ← Latent Space
         ↓
    Dense Layer (Sigmoid)
         ↓
   Output Layer (784 neurons)

Compression: 784 → 32 → 784
Ratio: 24.5:1 compression
```

Model Separation:

```
Autoencoder: [Input] → [Encoder] → [Decoder] → [Output]
Encoder:     [Input] → [Encoder] → [32-dim features]
Decoder:     [32-dim features] → [Decoder] → [Output]
```

4. Code Explanation

Data Preprocessing

The code flattens 28×28 MNIST images into 784-dimensional vectors and normalizes pixel values to range. This converts 2D image data into 1D vectors suitable for dense layer processing.

Architecture Design

This is a **simple/vanilla autoencoder** using only dense (fully connected) layers. The encoder compresses 784-dimensional input to 32-dimensional latent representation (96% compression), while the decoder reconstructs the original 784-dimensional output.

Three-Model Approach

The code creates three separate models:

- **Autoencoder**: For training the complete compression-reconstruction pipeline
- **Encoder**: For extracting compressed features from new data
- **Decoder**: For generating reconstructions from encoded features

Training Process

The autoencoder learns to minimize reconstruction error using binary crossentropy loss. With 50 epochs and batch size 256, it learns to capture the most important features of handwritten digits in just 32 dimensions, effectively removing noise and redundant information.

Latent Space Learning

The 32-dimensional bottleneck forces the network to learn a compact representation that preserves essential digit characteristics while discarding unnecessary details, making it useful for dimensionality reduction and feature extraction tasks.

1. Pseudo Code/Algorithm

2. Main Program Chunk (Core Code Only)

```
# Hyperparameters
batch_size = 100
original_dim = 784
latent_dim = 2
intermediate_dim = 256
nb_epoch = 5

# Encoder
x = Input(batch_shape=(batch_size, original_dim))
h = Dense(intermediate_dim, activation='relu')(x)
z_mean = Dense(latent_dim)(h)
z_log_var = Dense(latent_dim)(h)

def sampling(args):
    z_mean, z_log_var = args
    epsilon = K.random_normal(shape=(batch_size, latent_dim), mean=0.)
    return z_mean + K.exp(z_log_var / 2) * epsilon

z = Lambda(sampling, output_shape=(latent_dim,))([z_mean, z_log_var])

# Decoder
decoder_h = Dense(intermediate_dim, activation='relu')
decoder_mean = Dense(original_dim, activation='sigmoid')
h_decoded = decoder_h(z)
x_decoded_mean = decoder_mean(h_decoded)

# Loss function
def vae_loss(x, x_decoded_mean):
    xent_loss = original_dim * objectives.binary_crossentropy(x, x_decoded_mean)
    kl_loss = -0.5 * K.sum(1 + z_log_var - K.square(z_mean) - K.exp(z_log_var), axis=-1)
    return xent_loss + kl_loss

# Model compilation and training
vae = Model(x, x_decoded_mean)
vae.compile(optimizer='rmsprop', loss=vae_loss)
vae.fit(x_train, x_train, shuffle=True, nb_epoch=nb_epoch,
        batch_size=batch_size, validation_data=(x_test, x_test))
```

3. Architecture Specifications and Parameters

Parameters

- **Input Shape**: (batch_size, 784) - flattened MNIST images
- **Latent Dimension**: 2 (2D latent space for visualization)
- **Intermediate Dimension**: 256 neurons
- **Batch Size**: 100
- **Epochs**: 5
- **Optimizer**: RMSprop
- **Loss Function**: Custom VAE loss (binary crossentropy + KL divergence)

Architecture Details

Total Layers: 6 layers (3 encoder + 1 sampling + 2 decoder)

Encoder Architecture:

- **Layer 1**: Dense - 256 units, ReLU activation
- **Layer 2**: Dense - 2 units, linear activation (z_mean)
- **Layer 3**: Dense - 2 units, linear activation (z_log_var)
- **Sampling Layer**: Lambda layer implementing reparameterization trick

Decoder Architecture:

- **Layer 4**: Dense - 256 units, ReLU activation
- **Layer 5**: Dense - 784 units, sigmoid activation

Architecture Diagram

```
Input (batch_size, 784)
         ↓
    Dense (256, ReLU)
         ↓
    ┌──────────────────────┐
    ↓                 ↓
Dense (2)        Dense (2)
z_mean           z_log_var
    ↓                 ↓
    └──────────────────────┘
         ↓
   Sampling Layer (Reparameterization)
         ↓
    Latent Space z (2D)
         ↓
    Dense (256, ReLU)
         ↓
    Dense (784, Sigmoid)
```

```
         ↓
    Reconstructed Output (784)
```

Key Components:

- **Probabilistic Encoder**: Outputs mean and log variance of latent distribution
- **Reparameterization Trick**: Enables backpropagation through stochastic sampling
- **Generative Decoder**: Reconstructs data from latent samples

4. Code Explanation

Variational Autoencoder Concept

This implements a **Variational Autoencoder (VAE)**, which differs from standard autoencoders by learning a probabilistic latent representation. Instead of deterministic encoding, the encoder outputs parameters of a Gaussian distribution in latent space.

Encoder with Probabilistic Output

The encoder produces two outputs: `z_mean` and `z_log_var`, representing the mean and log variance of a Gaussian distribution for each latent dimension. This probabilistic approach enables the model to generate new samples by sampling from the learned latent distribution.

Reparameterization Trick

The sampling function implements the reparameterization trick: $z = \mu + \sigma \odot \epsilon$, where ϵ is random noise. This allows gradients to flow through the stochastic sampling operation, making the model trainable via backpropagation.

VAE Loss Function

The loss combines two terms:

- **Reconstruction Loss**: Binary crossentropy measuring how well the decoder reconstructs the input
- **KL Divergence**: Regularizes the latent space to follow a standard Gaussian distribution, enabling smooth interpolation and generation

Generative Capabilities

Unlike standard autoencoders, VAEs can generate new data by sampling from the prior distribution (standard Gaussian) and passing samples through the decoder. The 2D latent space allows visualization of the learned representation and generation of new digits by sampling different points in latent space.

⁂

1. Pseudo Code/Algorithm

2. Main Program Chunk (Core Code Only)

```
# Data preprocessing
training_set = convert(training_set)
test_set = convert(test_set)
training_set = torch.FloatTensor(training_set)
test_set = torch.FloatTensor(test_set)

# Convert ratings to binary preference
training_set[training_set == 0] = -1
training_set[training_set == 1] = 0
training_set[training_set == 2] = 0
training_set[training_set >= 3] = 1

test_set[test_set == 0] = -1
test_set[test_set == 1] = 0
test_set[test_set == 2] = 0
test_set[test_set >= 3] = 1

# RBM class definition
class RBM():
    def __init__(self, nv, nh):
        self.W = torch.randn(nh, nv)
        self.a = torch.randn(1, nh)
        self.b = torch.randn(1, nv)
    def sample_h(self, x):
        wx = torch.mm(x, self.W.t())
        activation = wx + self.a.expand_as(wx)
        p_h_given_v = torch.sigmoid(activation)
        return p_h_given_v, torch.bernoulli(p_h_given_v)
    def sample_v(self, y):
        wy = torch.mm(y, self.W)
        activation = wy + self.b.expand_as(wy)
        p_v_given_h = torch.sigmoid(activation)
        return p_v_given_h, torch.bernoulli(p_v_given_h)
    def train(self, v0, vk, ph0, phk):
        self.W += torch.mm(v0.t(), ph0) - torch.mm(vk.t(), phk)
        self.b += torch.sum((v0 - vk), 0)
        self.a += torch.sum((ph0 - phk), 0)

# Initialize and train RBM
nv = len(training_set[^0])
nh = 100
rbm = RBM(nv, nh)

for epoch in range(1, nb_epoch + 1):
    train_loss = 0
    s = 0.
    for id_user in range(0, nb_users - batch_size, batch_size):
        vk = training_set[id_user:id_user+batch_size]
        v0 = training_set[id_user:id_user+batch_size]
        ph0,_ = rbm.sample_h(v0)
        for k in range(10):
            _,hk = rbm.sample_h(vk)
            _,vk = rbm.sample_v(hk)
            vk[v0<0] = v0[v0<0]
        phk,_ = rbm.sample_h(vk)
        rbm.train(v0, vk, ph0, phk)
        train_loss += torch.mean(torch.abs(v0[v0>=0] - vk[v0>=0]))
        s += 1.
```

3. Architecture Specifications and Parameters

Parameters

- **Number of Visible Units (nv)**: Number of movies in dataset (varies by dataset)
- **Number of Hidden Units (nh)**: 100 latent features
- **Batch Size**: 100 users per batch
- **Number of Epochs**: 10
- **Gibbs Sampling Steps**: 10 (k=10)
- **Learning Rate**: Implicit (no explicit learning rate in weight updates)

Architecture Details

Total Components: 3 main components (Visible layer + Hidden layer + Connections)

RBM Structure:

- **Visible Layer**: nv units (one per movie)
- **Hidden Layer**: 100 units (latent features)
- **Weight Matrix W**: (nh × nv) connecting visible and hidden units
- **Hidden Bias (a)**: (1 × nh) vector
- **Visible Bias (b)**: (1 × nv) vector

Data Representation:

- **Binary Preferences**: 1 (liked), 0 (not liked), -1 (unrated)
- **User-Item Matrix**: Each row represents a user's preferences across all movies

Architecture Diagram

```
Hidden Layer (100 units)
h₁  h₂  h₃  ...  h₁₀₀
 ○   ○   ○   ...   ○
 |\  |\  |\       /|
 | \ | \ | \     / |
 |  \|  \|  \   /  |
 |   \   \   \ /   |
 |    \   \   /    |
 |     \   \/     |
 ○      ○   ○      ○
v₁     v₂  v₃    vₙᵥ
Visible Layer (nv units - movies)

Biases: a (hidden), b (visible)
Weights: W (nh × nv matrix)
```

Training Process Flow:

```
Input: User ratings (v₀)
    ↓
Sample hidden units: p(h|v₀)
    ↓
Gibbs sampling (k=10 steps):
h → v → h → v → ... → vₖ
    ↓
Sample final hidden: p(h|vₖ)
    ↓
Update weights: ΔW = v₀ᵀh₀ - vₖᵀhₖ
```

4. Code Explanation

Restricted Boltzmann Machine (RBM) Concept

This implements an **RBM-based collaborative filtering recommendation system**. RBMs are generative stochastic neural networks that learn probability distributions over binary data. Unlike traditional autoencoders, RBMs use undirected connections and probabilistic sampling.

Data Preprocessing and Binarization

The code converts MovieLens ratings into a binary preference system: ratings ≥ 3 become 1 (liked), ratings <3 become 0 (not liked), and unrated items are marked as -1. This binarization simplifies the learning problem and makes the RBM more effective for recommendation tasks.

Sampling Functions

The `sample_h` and `sample_v` functions implement **Gibbs sampling**. Given visible units, `sample_h` computes the probability of hidden units being active using sigmoid activation, then samples binary states. Similarly, `sample_v` reconstructs visible units from hidden states. This probabilistic approach allows the model to capture uncertainty in user preferences.

Contrastive Divergence Training

The training uses **Contrastive Divergence (CD-k)** with k=10 Gibbs sampling steps. For each batch, the algorithm:

1. Initializes visible units with user ratings (v_0)
2. Performs alternating sampling between hidden and visible units
3. Updates weights based on the difference between initial and reconstructed samples
4. Preserves unrated items by setting `vk[v0<0] = v0[v0<0]`

Recommendation Generation

After training, the RBM can generate recommendations by sampling from the learned probability distribution. The hidden units capture latent factors (genres, preferences) that explain user-movie interactions, enabling the model to predict preferences for unseen movies and make personalized recommendations.

⁂

perplexity

1. Pseudo Code/Algorithm

```
1. Load and preprocess digit dataset:
   - Load 8x8 digit images from sklearn
   - Normalize pixel values to [0,1] range
   - Split into training and test sets (80:20 ratio)

2. Define stacked RBM architecture:
   - RBM Layer 1: 64 → 64 hidden units (feature extraction)
   - RBM Layer 2: 64 → 32 hidden units (dimensionality reduction)
   - Classifier: 32 → 10 classes (logistic regression)

3. Configure RBM parameters:
   - Learning rate: 0.06
   - Training iterations: 20 epochs per RBM
   - Use Bernoulli RBM for binary/continuous data

4. Create pipeline combining:
   - First RBM for initial feature learning
   - Second RBM for compressed representation
   - Logistic regression for final classification

5. Train the complete pipeline:
   - RBMs learn unsupervised feature representations
   - Logistic regression learns supervised classification

6. Evaluate performance:
   - Generate predictions on test set
   - Calculate classification metrics
   - Compare actual vs predicted labels
```

2. Main Program Chunk (Core Code Only)

```python
# Data preprocessing
digits = load_digits()
X, y = digits.data, digits.target
X = X / 16.0
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Define stacked RBM architecture
rbm1 = BernoulliRBM(n_components=64, learning_rate=0.06, n_iter=20, random_state=0)
rbm2 = BernoulliRBM(n_components=32, learning_rate=0.06, n_iter=20, random_state=0)
logistic = LogisticRegression(max_iter=1500)

# Create pipeline
stacked_rbm = Pipeline(steps=[
    ('rbm1', rbm1),
    ('rbm2', rbm2),
    ('logistic', logistic)
])

# Train and evaluate
stacked_rbm.fit(X_train, y_train)
y_pred = stacked_rbm.predict(X_test)
```

3. Architecture Specifications and Parameters

Parameters

- **Input Dimension**: 64 features (8×8 flattened digit images)
- **RBM1 Hidden Units**: 64 components
- **RBM2 Hidden Units**: 32 components
- **Output Classes**: 10 digits (0-9)
- **Learning Rate**: 0.06 for both RBMs
- **Training Iterations**: 20 epochs per RBM
- **Test Split**: 20% (360 samples)
- **Logistic Regression**: Max iterations = 1500

Architecture Details

Total Layers: 4 layers (Input + 2 RBM layers + Classifier)

Layer Structure:

- **Input Layer**: 64 features (normalized pixel values)
- **RBM Layer 1**: 64 hidden units (feature extraction)
- **RBM Layer 2**: 32 hidden units (dimensionality reduction)
- **Output Layer**: 10 classes (logistic regression classifier)

Model Performance: 82% accuracy with balanced precision and recall across digit classes

Architecture Diagram

```
Input Layer (64 features)
8×8 flattened digit image
         ↓
    RBM Layer 1
   (64 hidden units)
   Feature Learning
         ↓
    RBM Layer 2
   (32 hidden units)
 Dimensionality Reduction
         ↓
  Logistic Regression
```

```
    (10 output classes)
    Classification
         ↓
   Digit Prediction (0-9)

Data Flow: 64 → 64 → 32 → 10
```

Pipeline Structure:

```
Unsupervised Learning:
RBM1: [64 input] → [64 features] (learns edge/texture features)
RBM2: [64 features] → [32 features] (learns higher-level patterns)

Supervised Learning:
Logistic: [32 features] → [10 classes] (digit classification)
```

4. Code Explanation

Stacked RBM Architecture

This implements a **Deep Belief Network (DBN)** using stacked Restricted Boltzmann Machines for digit classification. The architecture combines unsupervised feature learning with supervised classification, creating a hybrid approach that leverages the strengths of both paradigms.

Two-Stage RBM Feature Learning

The first RBM (64→64) learns low-level features like edges and textures from raw pixel data. The second RBM (64→32) learns higher-level, more abstract representations while reducing dimensionality. This hierarchical feature learning mimics how deep networks extract increasingly complex patterns.

Pipeline Integration

The sklearn Pipeline seamlessly combines the RBMs with logistic regression, allowing the entire system to be trained and used like a single model. The RBMs perform unsupervised pre-training to learn good feature representations, while the logistic regression performs supervised fine-tuning for classification.

Performance Analysis

The model achieves 82% accuracy on digit classification. The classification report shows strong performance for digits like 0, 4, and 6 (precision >0.95), while digits 1, 8, and 9 prove more challenging due to their visual similarity and potential confusion between classes.

Hybrid Learning Approach

This architecture demonstrates the power of combining unsupervised and supervised learning. The RBMs learn meaningful feature representations without labels, capturing the underlying structure of digit images, while the final classifier leverages these learned features for accurate digit recognition.

⁂

perplexity

1. Pseudo Code/Algorithm

Load MNIST dataset and apply tensor transformation
Create DataLoader for training and testing with batch sizes 64 and 1000 respectively
Define a simple feedforward neural network with 3 fully connected layers:

- Input layer: 784 neurons (flattened 28×28 image)
- Hidden layer 1: 256 neurons with ReLU activation
- Hidden layer 2: 128 neurons with ReLU activation
- Output layer: 10 neurons (digit classes)
  Initialize model, loss function (CrossEntropyLoss), and Adam optimizer
  Train the model for 5 epochs:
- For each batch, compute output, calculate loss, backpropagate, and update weights
  Evaluate the model on test data and calculate accuracy
  Predict and display 5 sample test images with predicted and actual labels

2. Main Program Chunk (Core Code Only)

```
# Load MNIST dataset and create DataLoader
transform = transforms.ToTensor()
train_set = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
test_set = datasets.MNIST(root='./data', train=False, download=True, transform=transform)
train_loader = DataLoader(train_set, batch_size=64, shuffle=True)
test_loader = DataLoader(test_set, batch_size=1000)

# Define simple neural network
class SimpleDBN(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(28*28, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 10)

    def forward(self, x):
        x = x.view(-1, 28*28)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        return self.fc3(x)

# Initialize model, loss, optimizer
model = SimpleDBN()
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

# Training loop
```

```
for epoch in range(5):
    for images, labels in train_loader:
        outputs = model(images)
        loss = criterion(outputs, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

# Testing and evaluation
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for images, labels in test_loader:
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
```

3. Architecture Specifications and Parameters

Parameters

- **Input Shape**: 28×28 flattened to 784 neurons
- **Hidden Layer 1**: 256 neurons, ReLU activation
- **Hidden Layer 2**: 128 neurons, ReLU activation
- **Output Layer**: 10 neurons (digit classes)
- **Loss Function**: CrossEntropyLoss
- **Optimizer**: Adam with learning rate 0.001
- **Batch Size**: 64 for training, 1000 for testing
- **Epochs**: 5

Architecture Details

Total Layers: 3 fully connected layers

Layer Structure:

- **Input Layer**: 784 neurons
- **Hidden Layer 1**: 256 neurons with ReLU
- **Hidden Layer 2**: 128 neurons with ReLU
- **Output Layer**: 10 neurons producing logits

Architecture Diagram

```
Input Layer (784 neurons)
28×28 flattened image
         ↓
   Fully Connected 1
   (256 neurons + ReLU)
         ↓
   Fully Connected 2
   (128 neurons + ReLU)
         ↓
   Fully Connected 3
   (10 neurons - logits)
         ↓
   Output Classes (0-9)

Data Flow: 784 → 256 → 128 → 10
```

Network Structure:

```
Input (784) -> FC(256) + ReLU -> FC(128) + ReLU -> FC(10) -> Output Classes
```

4. Code Explanation

Data Loading and Preprocessing

MNIST dataset is loaded using torchvision with tensor transformation. DataLoader batches data with batch size 64 for training and 1000 for testing.

Model Architecture

Simple feedforward neural network with 3 fully connected layers. Input images are flattened from 28×28 to 784-dimensional vectors. Two hidden layers with ReLU activation introduce non-linearity. Output layer produces logits for 10 digit classes.

Training Process

Uses CrossEntropyLoss suitable for multi-class classification. Adam optimizer with learning rate 0.001 updates model weights. For each epoch, batches are processed with forward pass, loss calculation, backpropagation, and optimizer step.

Evaluation

Model is evaluated on test data without gradient computation. Accuracy is calculated by comparing predicted labels with true labels.

Prediction Visualization

Predicts labels for 5 sample test images. Displays images with predicted and actual labels using matplotlib.

⁂

perplexity

1. Pseudo Code/Algorithm

```
1. Initialize Deep Boltzmann Machine parameters:
   - Set visible units v0 = [1, 0, 1, 0] (4 units)
   - Initialize weight matrices W1 (4×3) and W2 (3×2) with small random values
   - Initialize bias vectors b1 (3 units) and b2 (2 units) to zeros
   - Set learning rate lr = 0.01

2. Define helper functions:
   - sigmoid(x): Compute sigmoid activation function
   - sample(prob): Sample binary units using binomial distribution
   - sample_layer(input, weights, bias): Compute activation, probability, and sample

3. Perform one DBM training step:
   a. UPWARD PASS (Inference):
      - Sample h1 from visible units v0 using W1 and b1
      - Sample h2 from hidden1 units h1 using W2 and b2

   b. DOWNWARD PASS (Reconstruction):
      - Reconstruct h1 from h2 using W2^T
      - Reconstruct v1 from reconstructed h1 using W1^T

   c. WEIGHT UPDATES (Contrastive Divergence):
      - Positive phase: Compute correlations between original data
      - Negative phase: Compute correlations between reconstructed data
      - Update weights: W += lr × (positive_correlations - negative_correlations)
      - Update biases: b += lr × (original_activations - reconstructed_activations)

4. Return updated parameters W1, b1, W2, b2
```

2. Main Program Chunk (Core Code Only)

```python
# Helper functions
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sample(prob):
    return np.random.binomial(1, prob)

def sample_layer(input_data, weights, bias):
    activation = np.dot(input_data, weights) + bias
    prob = sigmoid(activation)
    return sample(prob), prob

# DBM training step
def dbm_step(v0, W1, b1, W2, b2, lr=0.01):
    # Upward pass
    h1, h1_prob = sample_layer(v0, W1, b1)
    h2, h2_prob = sample_layer(h1, W2, b2)

    # Downward pass (reconstruction)
    h1_down, _ = sample_layer(h2, W2.T, np.zeros_like(b1))
    v1, _ = sample_layer(h1_down, W1.T, np.zeros_like(v0))

    # Weight updates
    pos_W1 = np.outer(v0, h1)
    pos_W2 = np.outer(h1, h2)
    neg_W1 = np.outer(v1, h1_down)
    neg_W2 = np.outer(h1_down, h2)

    W1 += lr * (pos_W1 - neg_W1)
    W2 += lr * (pos_W2 - neg_W2)
    b1 += lr * (h1 - h1_down)
    b2 += lr * (h2 - h2_prob)

    return W1, b1, W2, b2

# Initialize and train
np.random.seed(42)
v0 = np.array([1, 0, 1, 0])
W1 = np.random.randn(4, 3) * 0.1
b1 = np.zeros(3)
W2 = np.random.randn(3, 2) * 0.1
b2 = np.zeros(2)

W1, b1, W2, b2 = dbm_step(v0, W1, b1, W2, b2)
```

3. Architecture Specifications and Parameters

Parameters

- **Visible Units**: 4 binary units (input layer)
- **Hidden Layer 1**: 3 binary units
- **Hidden Layer 2**: 2 binary units (top layer)
- **Learning Rate**: 0.01
- **Weight Initialization**: Gaussian random values × 0.1
- **Bias Initialization**: Zeros
- **Activation Function**: Sigmoid
- **Sampling**: Binomial distribution for binary units

Architecture Details

Total Layers: 3 layers (1 visible + 2 hidden)

Layer Structure:

- **Visible Layer**: 4 binary units

- **Hidden Layer 1**: 3 binary units
- **Hidden Layer 2**: 2 binary units

Connection Matrices:

- **W1**: 4×3 matrix (visible ↔ hidden1)
- **W2**: 3×2 matrix (hidden1 ↔ hidden2)
- **b1**: 3-element bias vector for hidden1
- **b2**: 2-element bias vector for hidden2

Architecture Diagram

```
Hidden Layer 2 (2 units)
    h2₁    h2₂
     ○      ○
     |\    /|
     | \  / |
     |  \/  |
     |  /\  |
     | /  \ |
     |/    \|
     ○      ○      ○
   h1₁    h1₂    h1₃
Hidden Layer 1 (3 units)
     |      |      |
     |\     |     /|
     | \    |    / |
     |  \   |   /  |
     |   \  |  /   |
     |    \ | /    |
     |     \|/     |
     ○      ○      ○      ○
    v₁     v₂     v₃     v₄
Visible Layer (4 units)

Weights: W1 (4×3), W2 (3×2)
Biases: b1 (3), b2 (2)
```

4. Code Explanation

Deep Boltzmann Machine (DBM) Concept

This implements a simplified **Deep Boltzmann Machine**, which is a type of deep generative model with multiple layers of hidden units. Unlike RBMs that have only one hidden layer, DBMs have multiple hidden layers that can learn hierarchical representations of data.

Bidirectional Information Flow

The DBM performs both **upward pass** (inference) and **downward pass** (reconstruction). The upward pass samples hidden units from visible units, while the downward pass reconstructs the original data. This bidirectional flow allows the model to learn both generative and discriminative features.

Contrastive Divergence Learning

The training uses a **Contrastive Divergence-like** algorithm that compares the positive phase (correlations in original data) with the negative phase (correlations in reconstructed data). The weight updates follow the principle: increase weights for positive correlations and decrease weights for negative correlations.

Stochastic Binary Units

All units in the DBM are **binary stochastic units** that sample their states based on sigmoid probabilities. This stochastic sampling introduces noise that helps the model escape local minima and learn robust representations of the input data.

Hierarchical Feature Learning

The two-layer structure allows the DBM to learn hierarchical features: the first hidden layer captures low-level patterns, while the second hidden layer learns higher-level abstractions. This hierarchical learning is crucial for modeling complex data distributions and generating realistic samples.

⁂

1. Pseudo Code/Algorithm

Load the MNIST dataset and normalize pixel values to range
Reshape data to include single channel dimension (28×28×1) for CNN input
Define convolutional autoencoder architecture:

- Encoder: Conv2D(32) → MaxPool → Conv2D(16) → MaxPool
- Decoder: Conv2D(16) → UpSample → Conv2D(32) → UpSample → Conv2D(1)
  Compile model with Adam optimizer and binary crossentropy loss
  Train model for 20 epochs with batch size 256, validating on test data
  Generate predictions (reconstructions) on test set
  Randomly select test image for visualization
  Display original and reconstructed images side by side
  Calculate Mean Squared Error between original and reconstructed image

2. Main Program Chunk (Core Code Only)

```
# Data preprocessing
(x_train, _), (x_test, _) = mnist.load_data()
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
x_train = x_train.reshape(len(x_train), 28, 28, 1)
x_test = x_test.reshape(len(x_test), 28, 28, 1)

# Model definition
model = Sequential([
    # Encoder
    Conv2D(32, 3, activation='relu', padding='same', input_shape=(28, 28, 1)),
    MaxPooling2D(2, padding='same'),
    Conv2D(16, 3, activation='relu', padding='same'),
    MaxPooling2D(2, padding='same'),
    # Decoder
    Conv2D(16, 3, activation='relu', padding='same'),
    UpSampling2D(2),
    Conv2D(32, 3, activation='relu', padding='same'),
    UpSampling2D(2),
    Conv2D(1, 3, activation='sigmoid', padding='same')
])

# Training
model.compile(optimizer='adam', loss='binary_crossentropy')
model.fit(x_train, x_train, epochs=20, batch_size=256, validation_data=(x_test, x_test))

# Prediction and evaluation
pred = model.predict(x_test)
```

3. Architecture Specifications and Parameters

Parameters

- **Input Shape**: (28, 28, 1) - grayscale MNIST images
- **Optimizer**: Adam
- **Loss Function**: Binary crossentropy
- **Epochs**: 20
- **Batch Size**: 256
- **Learning Rate**: Default Adam (0.001)

Architecture Details

Total Layers: 9 layers (4 encoder + 5 decoder)

Encoder Architecture:

- **Layer 1**: Conv2D - 32 filters, 3×3 kernel, ReLU activation, same padding
- **Layer 2**: MaxPooling2D - 2×2 pool size, same padding
- **Layer 3**: Conv2D - 16 filters, 3×3 kernel, ReLU activation, same padding
- **Layer 4**: MaxPooling2D - 2×2 pool size, same padding (bottleneck: 7×7×16)

Decoder Architecture:

- **Layer 5**: Conv2D - 16 filters, 3×3 kernel, ReLU activation, same padding
- **Layer 6**: UpSampling2D - 2×2 upsampling
- **Layer 7**: Conv2D - 32 filters, 3×3 kernel, ReLU activation, same padding
- **Layer 8**: UpSampling2D - 2×2 upsampling
- **Layer 9**: Conv2D - 1 filter, 3×3 kernel, sigmoid activation, same padding

Architecture Diagram

```
Input (28×28×1)
       ↓
   Conv2D (32) + ReLU
       ↓
   MaxPool2D (2×2)
       ↓ (14×14×32)
   Conv2D (16) + ReLU
       ↓
   MaxPool2D (2×2)
       ↓ (7×7×16)  ← Bottleneck/Latent Space
   Conv2D (16) + ReLU
       ↓
   UpSample2D (2×2)
       ↓ (14×14×16)
   Conv2D (32) + ReLU
```

```
    ↓
UpSample2D (2×2)
    ↓ (28×28×32)
Conv2D (1) + Sigmoid
    ↓
Output (28×28×1)
```

4. Code Explanation

Data Preprocessing

The code loads MNIST handwritten digit images and normalizes pixel values to range for stable training. Images are reshaped to include a channel dimension (28×28×1), making them compatible with convolutional layers that expect 3D input tensors.

Encoder Architecture

The encoder progressively compresses input images through two convolutional layers with ReLU activation followed by max pooling operations. The first Conv2D layer extracts 32 feature maps capturing edges and textures, while the second reduces this to 16 feature maps, creating a compressed representation in the bottleneck layer (7×7×16 = 784 parameters vs original 784 pixels).

Decoder Architecture

The decoder mirrors the encoder structure but uses upsampling instead of pooling to reconstruct original image dimensions. It starts with the compressed representation and gradually increases spatial resolution while reducing feature depth. The final layer uses sigmoid activation to ensure output values are between 0 and 1, matching the normalized input range.

Training and Evaluation

The model is trained using the same images as both input and target (unsupervised learning). Binary crossentropy loss measures reconstruction quality, while Adam optimizer adjusts weights to minimize this loss. The visualization component randomly selects a test image and displays both original and reconstructed versions side by side, with MSE calculation providing quantitative assessment of reconstruction fidelity.

Compression and Reconstruction

The autoencoder learns to compress 28×28 images into a 7×7×16 latent representation (96% compression) while preserving essential digit characteristics. This compressed representation captures the most important features needed to reconstruct recognizable digit images, effectively removing noise and redundant information.

⁂

perplexity

1. Pseudo Code/Algorithm

Load and normalize MNIST dataset to range and reshape to (28, 28, 1) for CNN input
Define encoder with two stages:

- Conv2D with 16 filters, 3×3 kernel, ReLU activation, same padding
- MaxPooling2D with 2×2 pool size, same padding
- Conv2D with 8 filters, 3×3 kernel, ReLU activation, same padding
- MaxPooling2D with 2×2 pool size, same padding (creates bottleneck)
  Define decoder with symmetric structure:
- Conv2D with 8 filters, 3×3 kernel, ReLU activation, same padding
- UpSampling2D with 2×2 size
- Conv2D with 16 filters, 3×3 kernel, ReLU activation, same padding
- UpSampling2D with 2×2 size
- Conv2D with 1 filter, 3×3 kernel, sigmoid activation, same padding
  Build autoencoder model from input to decoded output
  Compile model with Adam optimizer and binary crossentropy loss
  Train model for 5 epochs with batch size 128, shuffle training data
  Predict reconstructed images on test set
  Visualize original and reconstructed images side by side

2. Main Program Chunk (Core Code Only)

```
# Load and preprocess MNIST dataset
(x_train, _), (x_test, _) = mnist.load_data()
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = np.reshape(x_train, (len(x_train), 28, 28, 1))
x_test = np.reshape(x_test, (len(x_test), 28, 28, 1))

# Define encoder
input_img = Input(shape=(28, 28, 1))
x = Conv2D(16, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
encoded = MaxPooling2D((2, 2), padding='same')(x)

# Define decoder
x = Conv2D(8, (3, 3), activation='relu', padding='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(16, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)
```

```
# Build and compile autoencoder
autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

# Train model
autoencoder.fit(x_train, x_train, epochs=5, batch_size=128, shuffle=True, validation_data

# Predict on test set
decoded_imgs = autoencoder.predict(x_test)
```

3. Architecture Specifications and Parameters

Parameters

- **Input Shape**: (28, 28, 1) grayscale images
- **Optimizer**: Adam
- **Loss Function**: Binary crossentropy
- **Epochs**: 5
- **Batch Size**: 128
- **Learning Rate**: Default Adam (0.001)

Architecture Details

Total Layers: 9 layers (4 encoder + 5 decoder)

Encoder Architecture:

- **Layer 1**: Conv2D - 16 filters, 3×3 kernel, ReLU activation, same padding
- **Layer 2**: MaxPooling2D - 2×2 pool size, same padding
- **Layer 3**: Conv2D - 8 filters, 3×3 kernel, ReLU activation, same padding
- **Layer 4**: MaxPooling2D - 2×2 pool size, same padding (bottleneck)

Decoder Architecture:

- **Layer 5**: Conv2D - 8 filters, 3×3 kernel, ReLU activation, same padding
- **Layer 6**: UpSampling2D - 2×2
- **Layer 7**: Conv2D - 16 filters, 3×3 kernel, ReLU activation, same padding
- **Layer 8**: UpSampling2D - 2×2
- **Layer 9**: Conv2D - 1 filter, 3×3 kernel, sigmoid activation, same padding

Architecture Diagram

```
Input (28×28×1)
       ↓
   Conv2D (16) + ReLU
       ↓
   MaxPooling2D (2×2)
       ↓ (14×14×16)
   Conv2D (8) + ReLU
       ↓
   MaxPooling2D (2×2)
       ↓ (7×7×8) ← Bottleneck
   Conv2D (8) + ReLU
       ↓
   UpSampling2D (2×2)
       ↓ (14×14×8)
   Conv2D (16) + ReLU
       ↓
   UpSampling2D (2×2)
       ↓ (28×28×16)
   Conv2D (1) + Sigmoid
       ↓
   Output (28×28×1)
```

4. Code Explanation

Data Preprocessing

MNIST dataset is loaded and pixel values are normalized to range for stable training. Images are reshaped to (28, 28, 1) to add a channel dimension required for convolutional layers.

Encoder Architecture

The encoder compresses the input image through two convolutional layers with ReLU activation. MaxPooling layers reduce spatial dimensions by half each time, resulting in a bottleneck representation of size 7×7×8. This creates a compressed latent representation that captures essential image features.

Decoder Architecture

The decoder reconstructs the image by reversing the encoder operations. UpSampling layers increase spatial dimensions back to original size. The final Conv2D layer uses sigmoid activation to output pixel values between 0 and 1, ensuring the reconstructed images match the normalized input range.

Training Process

The model is compiled with Adam optimizer and binary crossentropy loss. Training occurs for 5 epochs with batch size 128, shuffling training data. The autoencoder learns to minimize reconstruction error by using the same images as both input and target.

Visualization and Evaluation

The trained model predicts reconstructed images from the test set. Original and reconstructed images are displayed side by side for visual comparison. This convolutional autoencoder learns to compress and reconstruct MNIST images, capturing essential features while reducing dimensionality from 784 pixels to 392 parameters in the bottleneck layer.

⁂

perplexity

1. Pseudo Code/Algorithm

Load and normalize MNIST dataset with mean=0.5 and std=0.5
Define Generator (G) network:

- Input: random noise vector (100-dim)
- Fully connected layer with 256 units + ReLU
- Fully connected layer with 784 units + Tanh activation
  Define Discriminator (D) network:
- Input: flattened image (784-dim)
- Fully connected layer with 256 units + LeakyReLU(0.2)
- Fully connected layer with 1 unit + Sigmoid activation
  Define optimizers for G and D (Adam, lr=0.0002)
  Define binary cross entropy loss
  For each epoch (5 epochs):
  a. For each batch of real images:
    - Flatten real images
    - Generate fake images from random noise using G
    - Compute D loss on real images (label=1) and fake images (label=0)
    - Backpropagate and update D
    - Compute G loss based on D's prediction on fake images (label=1)
    - Backpropagate and update G
      After training, generate samples from G using random noise
      Visualize generated images

2. Main Program Chunk (Core Code Only)

```
# Data loading and preprocessing
data = DataLoader(datasets.MNIST('.', train=True, download=True,
    transform=transforms.Compose([transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))])),
    batch_size=64, shuffle=True)

# Define Generator and Discriminator
G = nn.Sequential(nn.Linear(100, 256), nn.ReLU(), nn.Linear(256, 784), nn.Tanh())
D = nn.Sequential(nn.Linear(784, 256), nn.LeakyReLU(0.2), nn.Linear(256, 1), nn.Sigmoid()

# Optimizers and loss function
opt_G = optim.Adam(G.parameters(), lr=0.0002)
opt_D = optim.Adam(D.parameters(), lr=0.0002)
loss = nn.BCELoss()
```

```
# Training loop
for epoch in range(5):
    for real, _ in data:
        real = real.view(-1, 784)
        z = torch.randn(real.size(0), 100)
        fake = G(z)

        # Train Discriminator
        D_real = D(real)
        D_fake = D(fake.detach())
        loss_D = loss(D_real, torch.ones_like(D_real)) + loss(D_fake, torch.zeros_like(D_
        opt_D.zero_grad()
        loss_D.backward()
        opt_D.step()

        # Train Generator
        D_fake = D(fake)
        loss_G = loss(D_fake, torch.ones_like(D_fake))
        opt_G.zero_grad()
        loss_G.backward()
        opt_G.step()

# Generate and visualize images
z = torch.randn(num_samples, 100)
generated_images = G(z).view(-1, 28, 28).detach().numpy()
```

3. Architecture Specifications and Parameters

Parameters

- **Input noise dimension**: 100
- **Generator hidden layer**: 256 units, ReLU activation
- **Generator output layer**: 784 units, Tanh activation
- **Discriminator input**: 784 units (flattened image)
- **Discriminator hidden layer**: 256 units, LeakyReLU(0.2) activation
- **Discriminator output layer**: 1 unit, Sigmoid activation
- **Optimizer**: Adam with learning rate 0.0002 for both G and D
- **Loss function**: Binary Cross Entropy
- **Batch size**: 64
- **Epochs**: 5

Architecture Details

Total Networks: 2 networks (Generator + Discriminator)

Generator Architecture:

- **Layer 1**: Fully connected - 100 → 256 units, ReLU activation
- **Layer 2**: Fully connected - 256 → 784 units, Tanh activation

Discriminator Architecture:

- **Layer 1**: Fully connected - 784 → 256 units, LeakyReLU(0.2) activation
- **Layer 2**: Fully connected - 256 → 1 unit, Sigmoid activation

Architecture Diagram

```
GENERATOR NETWORK:
Noise (100) → FC(256) + ReLU → FC(784) + Tanh → Generated Image (784)

DISCRIMINATOR NETWORK:
Image (784) → FC(256) + LeakyReLU(0.2) → FC(1) + Sigmoid → Real/Fake Probability
```

Adversarial Training Flow:

```
Random Noise (100) → Generator → Fake Images (784)
                                         ↓
Real Images (784) ←→ Discriminator ←→ Real/Fake Classification
                                         ↓
                              Loss Feedback to both networks
```

4. Code Explanation

Generative Adversarial Network (GAN) Concept

This implements a simple **Generative Adversarial Network (GAN)** to generate MNIST digit images. GANs consist of two competing neural networks: a Generator that creates fake data and a Discriminator that distinguishes between real and fake data through adversarial training.

Data Loading and Preprocessing

MNIST dataset is loaded with normalization to mean=0.5 and std=0.5. DataLoader batches data with batch size 64 and shuffling. This normalization maps pixel values to the range [-1, 1], which matches the Tanh output range of the Generator.

Generator Network

The Generator takes a 100-dimensional random noise vector as input. It passes through a fully connected layer with 256 units and ReLU activation, then outputs a 784-dimensional vector (28×28 image flattened) with Tanh activation to produce pixel values in [-1, 1].

Discriminator Network

The Discriminator takes a 784-dimensional flattened image as input. It passes through a fully connected layer with 256 units and LeakyReLU activation (negative slope 0.2), then outputs a single probability value via Sigmoid indicating real or fake.

Adversarial Training Process

The training uses Binary Cross Entropy loss for both networks. For each batch, real images are flattened and fake images are generated from random noise. The Discriminator is trained to classify real images as 1 and fake images as 0, while the Generator is trained to fool the Discriminator by making it classify fake images as real (label=1). This adversarial process continues until the Generator learns to create realistic handwritten digit images.

⁂