```python
# Importing the MNIST dataset from TensorFlow/Keras
from tensorflow.keras.datasets import mnist

# Loading the MNIST dataset into training and test sets
# train_images: 60,000 of 28x28 grayscale images of handwritten digits (0-9)
# train_labels: Corresponding digit labels (0-9)
# test_images: 10,000 of images for testing
# test_labels: Corresponding test digit labels
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

# Examining the shape of training data:
# 60,000 images - each of 28x28 pixels
train_images.shape

# Number of training labels (should match number of images)
len(train_labels)

# Display first few training labels (to verify data)
train_labels

# Shape of test data: 10,000 images - each of 28x28 pixels
test_images.shape

# Number of test labels
len(test_labels)

# Display first few test labels
test_labels

# Importing Keras modules for building neural network
from tensorflow import keras
from tensorflow.keras import layers

# Creating a sequential neural network model:
# - First layer: Dense (fully connected) with 512 neurons and ReLU activation
# - Second layer: Dense output layer with 10 neurons (for 10 digits) and softmax activation
model = keras.Sequential([
    layers.Dense(512, activation="relu"),
    layers.Dense(10, activation="softmax")
])

# Compiling the model with:
# - RMSprop optimizer
# - Sparse categorical cross-entropy loss (for integer labels)
# - Accuracy as the evaluation metric
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])

# Preprocessing the image data:
# Reshaping from (60000, 28, 28) to (60000, 784) - flattening 28x28 to 784 pixels
train_images = train_images.reshape((60000, 28 * 28))
# Normalizing pixel values from 0-255 to 0-1
train_images = train_images.astype("float32") / 255

# Same preprocessing for test images
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype("float32") / 255

# Training the model for 5 epochs with batch size of 128
model.fit(train_images, train_labels, epochs=5, batch_size=128)

# Making predictions on first 10 test images
test_digits = test_images[0:10]
predictions = model.predict(test_digits)

```

```
# Displaying the prediction probabilities for first test image
predictions[0]

# Getting the predicted digit (class with highest probability)
predictions[0].argmax()

# Showing the probability score for digit 7 (just as an example)
predictions[0][7]

# Displaying the actual label of first test image
test_labels[0]

# Evaluating model performance on entire test set
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f"test_acc: {test_acc}")

# Demonstrating NumPy arrays and their dimensions
import numpy as np
x = np.array(12)  # scalar (0D tensor)
x

# Checking dimensions of the array
x.ndim

# Creating a 1D array (vector)
x = np.array([12, 3, 6, 14, 7])
x

# Checking dimensions
x.ndim

# Creating a 2D array (matrix)
x = np.array([[5, 78, 2, 34, 0],
              [6, 79, 3, 35, 1],
              [7, 80, 4, 36, 2]])
x.ndim

# Creating a 3D array
x = np.array([[[5, 78, 2, 34, 0],
               [6, 79, 3, 35, 1],
               [7, 80, 4, 36, 2]],
              [[5, 78, 2, 34, 0],
               [6, 79, 3, 35, 1],
               [7, 80, 4, 36, 2]],
              [[5, 78, 2, 34, 0],
               [6, 79, 3, 35, 1],
               [7, 80, 4, 36, 2]]])
x.ndim

# Reloading MNIST data to demonstrate tensor properties
from tensorflow.keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

# Checking tensor properties:
train_images.ndim  # Number of dimensions (3)
train_images.shape  # Shape (60000, 28, 28)
train_images.dtype  # Data type (uint8)

# Visualizing a sample digit
import matplotlib.pyplot as plt
digit = train_images[4]  # Get 5th image
plt.imshow(digit, cmap=plt.cm.binary)  # Display as grayscale
plt.show()

# Checking the label of this digit
train_labels[4]

```

```python
# Demonstrating tensor slicing operations
my_slice = train_images[10:100]  # Get images 10-99
my_slice.shape

# Equivalent slice with explicit dimensions
my_slice = train_images[10:100, :, :]
my_slice.shape

# More specific slicing
my_slice = train_images[10:100, 0:28, 0:28]
my_slice.shape

# Center-cropping digits (keeping center 14x14 pixels)
my_slice = train_images[:, 14:, 14:]  # All images, bottom-right 14x14

# Alternative center crop (keeping center 14x14 pixels)
my_slice = train_images[:, 7:-7, 7:-7]

# Batch processing examples
batch = train_images[:128]  # First batch of 128 images
batch = train_images[128:256]  # Second batch of 128 images

# General batch selection
n = 3
batch = train_images[128 * n:128 * (n + 1)]  # nth batch

# Implementing ReLU activation function manually
def naive_relu(x):
    assert len(x.shape) == 2  # Only works for 2D arrays
    x = x.copy()  # Do not modify original
    for i in range(x.shape[0]):  # Rows
        for j in range(x.shape[1]):  # Columns
            x[i, j] = max(x[i, j], 0)  # ReLU: max(0, x)
    return x

# Implementing matrix addition manually
def naive_add(x, y):
    assert len(x.shape) == 2  # Only works for 2D arrays
    assert x.shape == y.shape  # Must be same shape
    x = x.copy()  # Do not modify original
    for i in range(x.shape[0]):  # Rows
        for j in range(x.shape[1]):  # Columns
            x[i, j] += y[i, j]  # Element-wise addition
    return x

# Performance comparison between NumPy and manual operations
import time
x = np.random.random((20, 100))  # Random 20x100 matrices
y = np.random.random((20, 100))

# Time NumPy operations (1000 iterations)
t0 = time.time()
for _ in range(1000):
    z = x + y  # Matrix addition
    z = np.maximum(z, 0.)  # ReLU
print("Took: {0:.2f} s".format(time.time() - t0))

# Time manual operations (1000 iterations)
t0 = time.time()
for _ in range(1000):
    z = naive_add(x, y)  # Manual addition
    z = naive_relu(z)  # Manual ReLU
print("Took: {0:.2f} s".format(time.time() - t0))

# Demonstrating broadcasting with NumPy
import numpy as np
X = np.random.random((32, 10))  # 32 samples, 10 features
```

```python
y = np.random.random((10,))  # Single vector of 10 values

# Demonstrate broadcasting by expanding dimensions
y = np.expand_dims(y, axis=0)  # Now shape (1, 10)
Y = np.concatenate([y] * 32, axis=0)  # Now shape (32, 10) via tiling

# Manual implementation of matrix + vector addition
def naive_add_matrix_and_vector(x, y):
    assert len(x.shape) == 2  # Must be matrix
    assert len(y.shape) == 1  # Must be vector
    assert x.shape[1] == y.shape[0]  # Compatible dimensions
    x = x.copy()  # Do not modify original
    for i in range(x.shape[0]):  # Rows
        for j in range(x.shape[1]):  # Columns
            x[i, j] += y[j]  # Add vector element to each row
    return x

# More broadcasting examples
import numpy as np
x = np.random.random((64, 3, 32, 10))  # 4D tensor
y = np.random.random((32, 10))         # 2D tensor
z = np.maximum(x, y)  # Broadcasting across first two dimensions

# Vector dot product example
x = np.random.random((32,))
y = np.random.random((32,))
z = np.dot(x, y)  # Dot product

# Manual implementation of vector dot product
def naive_vector_dot(x, y):
    assert len(x.shape) == 1  # Must be vectors
    assert len(y.shape) == 1
    assert x.shape[0] == y.shape[0]  # Same length
    z = 0.  # Initialize result
    for i in range(x.shape[0]):
        z += x[i] * y[i]  # Sum of element-wise products
    return z

# Manual matrix-vector multiplication
def naive_matrix_vector_dot(x, y):
    assert len(x.shape) == 2  # Must be matrix and vector
    assert len(y.shape) == 1
    assert x.shape[1] == y.shape[0]  # Compatible dimensions
    z = np.zeros(x.shape[0])  # Initialize output vector
    for i in range(x.shape[0]):  # For each row
        for j in range(x.shape[1]):  # For each column
            z[i] += x[i, j] * y[j]  # Dot product with vector
    return z

# Alternative implementation using previous function
def naive_matrix_vector_dot_new(x, y):
    z = np.zeros(x.shape[0])
    for i in range(x.shape[0]):
        z[i] = naive_vector_dot(x[i, :], y)  # Use vector dot product
    return z

# Manual matrix multiplication
def naive_matrix_dot(x, y):
    assert len(x.shape) == 2  # Must be matrices
    assert len(y.shape) == 2
    assert x.shape[1] == y.shape[0]  # Compatible dimensions
    z = np.zeros((x.shape[0], y.shape[1]))  # Result matrix
    for i in range(x.shape[0]):  # Rows of x
        for j in range(y.shape[1]):  # Columns of y
            row_x = x[i, :]  # Get row from x
            column_y = y[:, j]  # Get column from y
            z[i, j] = naive_vector_dot(row_x, column_y)  # Dot product
```

```
    return z

# Reshaping demonstration
train_images = train_images.reshape((60000, 28 * 28))  # Flatten images

# More reshaping examples
x = np.array([[0., 1.],
              [2., 3.],
              [4., 5.]])
x.shape

x = x.reshape((6, 1))  # Reshape to 6x1
x

# Transposition example
x = np.zeros((300, 20))  # Create 300x20 matrix
x = np.transpose(x)  # Transpose to 20x300
x.shape

# TensorFlow automatic differentiation example
import tensorflow as tf
x = tf.Variable(0.)  # Create variable with value 0
with tf.GradientTape() as tape:
    y = 2 * x + 3  # Define computation
grad_of_y_wrt_x = tape.gradient(y, x)  # Compute gradient dy/dx

# Matrix case
x = tf.Variable(tf.random.uniform((2, 2)))  # 2x2 variable
with tf.GradientTape() as tape:
    y = 2 * x + 3  # Simple computation
grad_of_y_wrt_x = tape.gradient(y, x)  # Gradient is also 2x2

# More complex differentiation example
W = tf.Variable(tf.random.uniform((2, 2)))  # Weights
b = tf.Variable(tf.zeros((2,)))  # Bias
x = tf.random.uniform((2, 2))  # Input
with tf.GradientTape() as tape:
    y = tf.matmul(x, W) + b  # Linear transformation
grad_of_y_wrt_W_and_b = tape.gradient(y, [W, b])  # Gradients for W and b

# Reloading and preparing MNIST data again
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.reshape((60000, 28 * 28))  # Flatten
train_images = train_images.astype("float32") / 255  # Normalize
test_images = test_images.reshape((10000, 28 * 28))  # Same for test
test_images = test_images.astype("float32") / 255

# Rebuilding the model
model = keras.Sequential([
    layers.Dense(512, activation="relu"),  # Hidden layer
    layers.Dense(10, activation="softmax")  # Output layer
])

# Compiling the model
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])

# Training the model
model.fit(train_images, train_labels, epochs=5, batch_size=128)

# Custom layer implementation
import tensorflow as tf
class NaiveDense:
    def __init__(self, input_size, output_size, activation):
        self.activation = activation
        # Initialize weights with small random values
```

```
            w_shape = (input_size, output_size)
            w_initial_value = tf.random.uniform(w_shape, minval=0, maxval=1e-1)
            self.W = tf.Variable(w_initial_value)
            # Initialize biases with zeros
            b_shape = (output_size,)
            b_initial_value = tf.zeros(b_shape)
            self.b = tf.Variable(b_initial_value)

        # Forward pass
        def __call__(self, inputs):
            return self.activation(tf.matmul(inputs, self.W) + self.b)

        # Property to access weights
        @property
        def weights(self):
            return [self.W, self.b]

    # Custom sequential model implementation
    class NaiveSequential:
        def __init__(self, layers):
            self.layers = layers  # List of layers

        # Forward pass through all layers
        def __call__(self, inputs):
            x = inputs
            for layer in self.layers:
                x = layer(x)
            return x

        # Property to access all weights
        @property
        def weights(self):
            weights = []
            for layer in self.layers:
                weights += layer.weights
            return weights

    # Creating an instance of our custom model
    model = NaiveSequential([
        NaiveDense(input_size=28 * 28, output_size=512, activation=tf.nn.relu),
        NaiveDense(input_size=512, output_size=10, activation=tf.nn.softmax)
    ])
    assert len(model.weights) == 4  # 2 weight matrices + 2 bias vectors

    # Batch generator for custom training loop
    import math
    class BatchGenerator:
        def __init__(self, images, labels, batch_size=128):
            assert len(images) == len(labels)  # Sanity check
            self.index = 0  # Current position
            self.images = images
            self.labels = labels
            self.batch_size = batch_size
            self.num_batches = math.ceil(len(images) / batch_size)  # Total batches

        # Get next batch
        def next(self):
            images = self.images[self.index : self.index + self.batch_size]
            labels = self.labels[self.index : self.index + self.batch_size]
            self.index += self.batch_size  # Advance position
            return images, labels

    # Single training step implementation
    def one_training_step(model, images_batch, labels_batch):
        # Forward pass
        with tf.GradientTape() as tape:
            predictions = model(images_batch)  # Get predictions
```

```
            # Compute loss
            per_sample_losses = tf.keras.losses.sparse_categorical_crossentropy(
                labels_batch, predictions)
            average_loss = tf.reduce_mean(per_sample_losses)  # Average loss

        # Backward pass
        gradients = tape.gradient(average_loss, model.weights)  # Compute gradients
        update_weights(gradients, model.weights)  # Update weights
        return average_loss  # Return loss for monitoring

# Learning rate for weight updates
learning_rate = 1e-3

# Simple weight update function
def update_weights(gradients, weights):
    for g, w in zip(gradients, weights):
        # w = w - learning_rate * gradient
        w.assign_sub(g * learning_rate)

# Alternative using Keras optimizer
from tensorflow.keras import optimizers
optimizer = optimizers.SGD(learning_rate=1e-3)

# Optimizer-based weight update
def update_weights_new(gradients, weights):
    optimizer.apply_gradients(zip(gradients, weights))

# Full training loop
def fit(model, images, labels, epochs, batch_size=128):
    for epoch_counter in range(epochs):
        print(f"Epoch {epoch_counter}")
        batch_generator = BatchGenerator(images, labels, batch_size)
        for batch_counter in range(batch_generator.num_batches):
            images_batch, labels_batch = batch_generator.next()
            loss = one_training_step(model, images_batch, labels_batch)
            # Print progress periodically
            if batch_counter % 100 == 0:
                print(f"loss at batch {batch_counter}: {loss:.2f}")

# Reload and prepare MNIST data
from tensorflow.keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype("float32") / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype("float32") / 255

# Train our custom model
fit(model, train_images, train_labels, epochs=10, batch_size=128)

# Evaluating custom model
predictions = model(test_images)  # Get predictions
predictions = predictions.numpy()  # Convert to NumPy
predicted_labels = np.argmax(predictions, axis=1)  # Get class predictions
matches = predicted_labels == test_labels  # Compare with true labels
print(f"accuracy: {matches.mean():.2f}")  # Calculate accuracy
```