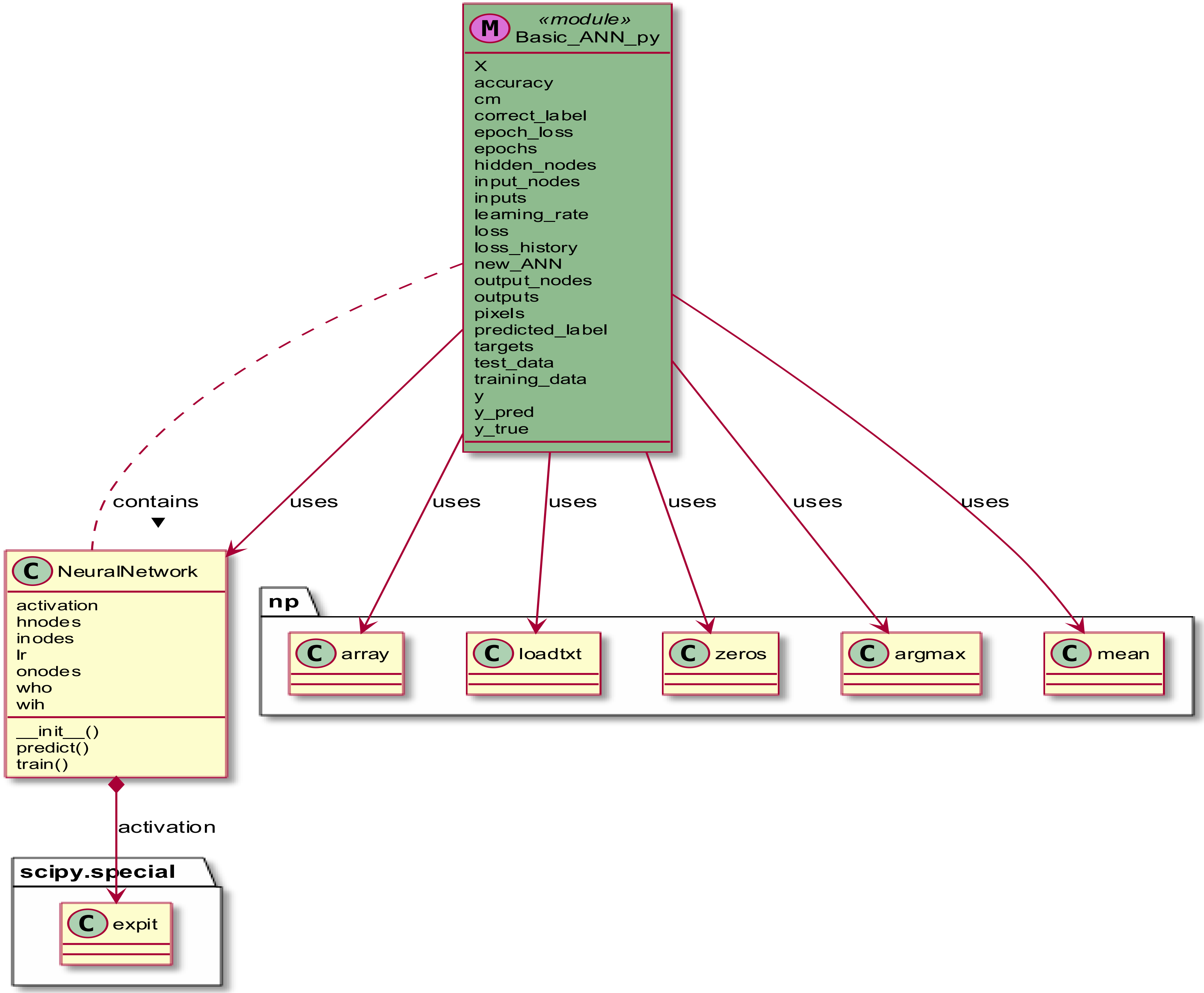

«module»
Basic_ANN_py
X
accuracy
cm
correct_label
epoch_loss
epochs
hidden_nodes
input_nodes
inputs
learning_rate
loss
loss_history
new_ANN
output_nodes
outputs
pixels
predicted_label
targets
test_data
training_data
y
y_pred
y_true
contains
uses
uses
uses
uses
uses
uses
NeuralNetwork
activation
hnodes
inodes
lr
onodes
who
wih
__init__()
predict()
train()
np
array
loadtxt
zeros
argmax
mean
activation
scipy.special
expit

```python
import numpy as np
import scipy.special
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import seaborn as sns

class NeuralNetwork:
    def __init__(self, input_nodes, hidden_nodes, output_nodes, learning_rate):
        self.inodes = input_nodes
        self.hnodes = hidden_nodes
        self.onodes = output_nodes
        self.lr = learning_rate

        # Initialize weights with normal distribution
        self.wih = np.random.normal(
            0.0, pow(self.inodes, -0.5), (self.hnodes, self.inodes)
        )
        self.who = np.random.normal(
            0.0, pow(self.hnodes, -0.5), (self.onodes, self.hnodes)
        )

        # Activation function (sigmoid)
        self.activation = lambda x: scipy.special.expit(x)

    def train(self, inputs_list, targets_list):
        inputs = np.array(inputs_list, ndmin=2).T
        targets = np.array(targets_list, ndmin=2).T

        # Forward pass
        hidden_inputs = np.dot(self.wih, inputs)
        hidden_outputs = self.activation(hidden_inputs)
        final_inputs = np.dot(self.who, hidden_outputs)
        final_outputs = self.activation(final_inputs)

        # Backpropagation
        output_errors = targets - final_outputs
        hidden_errors = np.dot(self.who.T, output_errors)

        # Update weights
        self.who += self.lr * np.dot(
            (output_errors * final_outputs * (1.0 - final_outputs)), hidden_outputs.T
        )

        self.wih += self.lr * np.dot(
            (hidden_errors * hidden_outputs * (1.0 - hidden_outputs)), inputs.T
        )

        return np.mean(output_errors**2)  # Return MSE loss

    def predict(self, inputs_list):
        inputs = np.array(inputs_list, ndmin=2).T
        hidden_outputs = self.activation(np.dot(self.wih, inputs))
        return self.activation(np.dot(self.who, hidden_outputs))

# Network parameters
input_nodes = 784  # 28x28 pixels
hidden_nodes = 200
output_nodes = 10  # 0-9 digits
learning_rate = 0.1
epochs = 5

# Initialize network
new_ANN = NeuralNetwork(input_nodes, hidden_nodes, output_nodes, learning_rate)

# Verification Step 1: Minimal test dataset
if __name__ == "__main__":
    # Tiny test dataset (2 samples)
    X = np.array([[0.1] * 784, [0.9] * 784])  # Fake image data
    y = np.array(
        [
            [0.99, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01],
            [0.01, 0.99, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01],
        ]
    )

    # Create and test network
    for i in range(3):  # Few training steps
        loss = new_ANN.train(X[0], y[0])
        print(f"Step {i+1}, Loss:", loss)

    # Verify prediction works
    print("Sample prediction:", new_ANN.predict(X[0]))

    # Verification Step 2: Load MNIST data
    print("\nLoading MNIST data...")
    training_data = np.loadtxt("mnist_train_100.csv", delimiter=",")
    print("First training sample label:", training_data[0, 0])
    print("Data shape:", training_data.shape)

    # Load test data
    test_data = np.loadtxt("mnist_test_10.csv", delimiter=",")
    print("First test sample label:", test_data[0, 0])
    print("Test data shape:", test_data.shape)

    # Verification Step 3: Check weights and data
    print("Weights shape - Input to Hidden:", new_ANN.wih.shape)  # Should be (200, 784)
    print("Weights shape - Hidden to Output:", new_ANN.who.shape)  # Should be (10, 200)

    # Check first image pixels
    pixels = training_data[0, 1:]
    print("Min pixel:", np.min(pixels), "Max pixel:", np.max(pixels))
    plt.imshow(pixels.reshape(28, 28), cmap="gray")
    plt.title("First Training Sample")
    plt.show()

    # Proceed with training and testing as before
    loss_history = []
    for epoch in range(epochs):
        epoch_loss = 0
        for record in training_data:
            inputs = (record[1:] / 255.0 * 0.99) + 0.01
            targets = np.zeros(output_nodes) + 0.01
            targets[int(record[0])] = 0.99

            epoch_loss += new_ANN.train(inputs, targets)

        loss_history.append(epoch_loss / len(training_data))
        print(f"Epoch {epoch+1}/{epochs}, Loss: {loss_history[-1]:.4f}")

    # Plot training loss
    plt.figure(figsize=(10, 5))
    plt.plot(loss_history)
    plt.title("Training Loss Progression")
    plt.xlabel("Epoch")
    plt.ylabel("Mean Squared Error")
    plt.grid(True)
    plt.show()

    # Testing process
    y_true = []
    y_pred = []
    for record in test_data:
        correct_label = int(record[0])
        inputs = (record[1:] / 255.0 * 0.99) + 0.01
        outputs = new_ANN.predict(inputs)
        predicted_label = np.argmax(outputs)

        y_true.append(correct_label)
        y_pred.append(predicted_label)

    # Confusion matrix
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(10, 8))
    sns.heatmap(
        cm,
        annot=True,
        fmt="d",
        cmap="Blues",
        xticklabels=range(10),
        yticklabels=range(10),
    )
    plt.title("Confusion Matrix")
    plt.xlabel("Predicted")
    plt.ylabel("Actual")
    plt.show()

    # Calculate accuracy
    accuracy = np.mean(np.array(y_true) == np.array(y_pred))
    print(f"\nFinal Test Accuracy: {accuracy * 100:.2f}%")
```