```
--------------------------------1--------------------------------
import numpy as np

# Define the inputs and expected outputs for the AND gate
inputs = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])
expected_outputs = np.array([0, 0, 0, 1])

# Define weights and threshold
weights = np.array([1, 1])  # Initial weights
threshold = 1.5

# McCulloch-Pitts Perceptron Function
def perceptron(x, weights, threshold):
    # Calculate weighted sum
    weighted_sum = np.dot(x, weights)
    # Apply threshold
    return 1 if weighted_sum >= threshold else 0

# Test the Perceptron on each input and print results
print("AND Gate using McCulloch-Pitts Model:")
for i in range(len(inputs)):
    output = perceptron(inputs[i], weights, threshold)
    print(f"Input: {inputs[i]} -> Output: {output} (Expected: {expected_outputs[i]})")

--------------------------------2--------------------------------
import numpy as np

# Inputs and step function
INPUTS = np.array([[1, 1], [1, -1], [-1, 1], [-1, -1]])

def step_function(sum): return 1 if sum >= 0 else -1

def calculate_output(weights, instance, bias): return step_function(np.dot(instance, weights) + bias)

# Hebbian Learning Algorithm
def hebb(outputs):
    weights, bias = np.zeros(2), 0  # Initialize weights and bias
    for i in range(len(outputs)):
        weights += INPUTS[i] * outputs[i]
        bias += outputs[i]
    return weights, bias

# Train, test, and print results for both AND and OR gates
def train_and_print(gate_name, outputs):
    weights, bias = hebb(outputs)
    print(f"\n{gate_name.upper()} Gate:")
    for input_vec in INPUTS:
        output = calculate_output(weights, input_vec, bias)
        print(f"Input: {input_vec}, Output: {output}")

# AND and OR gate outputs
and_outputs = np.array([1, -1, -1, -1])
```

```
or_outputs = np.array([1, 1, 1, -1])

# Print results for both gates
train_and_print("AND", and_outputs)
train_and_print("OR", or_outputs)
--------------------------------3--------------------------------
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score

# Generate synthetic dataset (100 crabs: shell width, claw size, weight)
np.random.seed(123)  # New seed for better class separation
blue_crabs = np.random.normal([5.4, 3.1, 0.35], 0.4, (50, 3))
orange_crabs = np.random.normal([6.2, 3.6, 0.55], 0.4, (50, 3))
data = np.vstack((blue_crabs, orange_crabs))
labels = np.array([0] * 50 + [1] * 50)

# Split the data
X_train, X_test, y_train, y_test = train_test_split(data, labels, test_size=0.2, random_state=42)

# Build and train the Pattern Net (MLP)
model = MLPClassifier(
    hidden_layer_sizes=(8, 8),  # Increased hidden units for better learning
    activation='relu',
    solver='adam',
    learning_rate_init=0.01,  # Slightly increased learning rate for faster convergence
    max_iter=1000,
    random_state=42
)
model.fit(X_train, y_train)

# Evaluate the model
y_pred = model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Test Accuracy: {accuracy * 100:.2f}%")

# Predict species for a new crab
new_crab = np.array([[5.9, 3.3, 0.5]])
prediction = model.predict(new_crab)
species = ["Blue", "Orange"]
print(f"The predicted species for the new crab is: {species[prediction[0]]}")
--------------------------------4--------------------------------
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score

# Data
X, y = load_wine(return_X_y=True)
X = StandardScaler().fit_transform(X)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Model
model = MLPClassifier(hidden_layer_sizes=(10,), activation='relu', solver='sgd',
learning_rate_init=0.01, max_iter=1000, random_state=42)
```

```
# Training
model.fit(X_train, y_train)

# Evaluation
y_pred = model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)

print(f'Accuracy: {accuracy * 100:.2f}%')
--------------------------------5--------------------------------
import numpy as np
# Define the function f(x, y)
def func(x, y):
    return x**2 + y**2
# Compute Jacobian (first derivatives)
def compute_jacobian(x, y):
    df_dx = 2 * x  # ∂f/∂x
    df_dy = 2 * y  # ∂f/∂y
    return np.array([df_dx, df_dy])
# Compute Hessian (second derivatives)
def compute_hessian(x, y):
    d2f_dx2 = 2  # ∂²f/∂x²
    d2f_dy2 = 2  # ∂²f/∂y²
    d2f_dxdy = 0  # ∂²f/∂x∂y
    d2f_dydx = 0  # ∂²f/∂y∂x
    return np.array([[d2f_dx2, d2f_dxdy],
                     [d2f_dydx, d2f_dy2]])
# Example values
x_val, y_val = 1.0, 2.0
# Compute Jacobian and Hessian
jacobian = compute_jacobian(x_val, y_val)
hessian = compute_hessian(x_val, y_val)
print("Jacobian:", jacobian)
print("Hessian:\n", hessian)
--------------------------------6--------------------------------
import numpy as np

# Data and labels
X = np.array([[2, 3], [1, 1], [2, 1], [3, 3], [2, 2]])
y = np.array([1, -1, -1, 1, -1])

# LMS algorithm
w, b, lr = np.zeros(2), 0, 0.01
for _ in range(1000):
    for i in range(len(X)):
        y_pred = np.dot(X[i], w) + b
        error = y[i] - y_pred
        w += lr * error * X[i]
        b += lr * error

# Prediction
pred = np.sign(np.dot(X, w) + b)
print(f"Final Weights: {w}, Bias: {b}, Predictions: {pred}")
--------------------------------7--------------------------------
import numpy as np

# Simple LSTM cell forward pass (single timestep)
```

```
def simple_lstm(x, h_prev, c_prev, Wf, Wi, Wo, Wc, bf, bi, bo, bc):
    z = np.concatenate((x, h_prev))  # Concatenate input and previous hidden state

    f = sigmoid(np.dot(Wf, z) + bf)  # Forget gate
    i = sigmoid(np.dot(Wi, z) + bi)  # Input gate
    o = sigmoid(np.dot(Wo, z) + bo)  # Output gate
    c_tilde = np.tanh(np.dot(Wc, z) + bc)  # Candidate cell state

    c = f * c_prev + i * c_tilde  # New cell state
    h = o * np.tanh(c)  # New hidden state

    return h, c

# Sigmoid activation function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Example data: a simple sine wave
x = np.array([0.5])  # Input at time step t
h_prev = np.zeros(1)  # Initial hidden state
c_prev = np.zeros(1)  # Initial cell state

# Random LSTM weights and biases for a single hidden unit
Wf = np.random.randn(1, 2)  # Forget gate weight
Wi = np.random.randn(1, 2)  # Input gate weight
Wo = np.random.randn(1, 2)  # Output gate weight
Wc = np.random.randn(1, 2)  # Candidate cell state weight

bf = np.zeros(1)  # Forget gate bias
bi = np.zeros(1)  # Input gate bias
bo = np.zeros(1)  # Output gate bias
bc = np.zeros(1)  # Cell state bias

# Forward pass through the LSTM
h, c = simple_lstm(x, h_prev, c_prev, Wf, Wi, Wo, Wc, bf, bi, bo, bc)

# Output the results
print("New hidden state:", h)
print("New cell state:", c)
--------------------------------8--------------------------------
import numpy as np

# Simple RNN parameters
input_size = 3
hidden_size = 5
output_size = 2
seq_length = 4

# Random data
X = np.random.randn(seq_length, input_size)
y = np.random.randint(0, output_size, size=(1,))

# Initialize weights and biases
Wh = np.random.randn(hidden_size, hidden_size)
Wx = np.random.randn(input_size, hidden_size)
Wy = np.random.randn(hidden_size, output_size)
bh = np.zeros((1, hidden_size))
```

```
by = np.zeros((1, output_size))

# Forward pass
h = np.zeros((1, hidden_size))
for t in range(seq_length):
    h = np.tanh(X[t].dot(Wx) + h.dot(Wh) + bh)  # RNN step
output = h.dot(Wy) + by  # Output layer
pred = np.argmax(output, axis=1)

print(f"Predicted class: {pred}, Actual class: {y}")
--------------------------------9--------------------------------
import numpy as np

# Random 5x5 input and 3x3 kernel
X = np.random.randn(5, 5)
W = np.random.randn(3, 3)

# Convolution operation (without padding, stride = 1)
conv_out = np.array([[np.sum(X[i:i+3, j:j+3] * W) for j in range(3)] for i in range(3)])

# ReLU activation
relu_out = np.maximum(0, conv_out)

# Max pooling (2x2)
pool_out = np.max(relu_out[:2, :2])

print("Convolution Output:\n", conv_out)
print("ReLU Output:\n", relu_out)
print("Max Pooling Output:", pool_out)
--------------------------------10--------------------------------
import numpy as np

# Sigmoid activation function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))  # Sigmoid function

# GRU parameters (random initialization)
input_size = 3  # Size of input vector
hidden_size = 2  # Size of hidden state

# Define the input and previous hidden state
X = np.array([0.5, -0.2, 0.1])  # Example input vector (3-dimensional)
h_prev = np.array([0.0, 0.0])   # Initial hidden state (2-dimensional)

# Random weights and biases
Wz = np.random.randn(input_size, hidden_size)  # Update gate weights
Wr = np.random.randn(input_size, hidden_size)  # Reset gate weights
Wh = np.random.randn(input_size, hidden_size)  # Candidate hidden state weights
Uz = np.random.randn(hidden_size, hidden_size)  # Update gate recurrent weights
Ur = np.random.randn(hidden_size, hidden_size)  # Reset gate recurrent weights
Uh = np.random.randn(hidden_size, hidden_size)  # Candidate hidden state recurrent weights
bz = np.zeros(hidden_size)  # Bias for update gate
br = np.zeros(hidden_size)  # Bias for reset gate
bh = np.zeros(hidden_size)  # Bias for candidate hidden state

# GRU operations (single step)
z = sigmoid(X.dot(Wz) + h_prev.dot(Uz) + bz)  # Update gate
```

```
r = sigmoid(X.dot(Wr) + h_prev.dot(Ur) + br)  # Reset gate
h_tilde = np.tanh(X.dot(Wh) + (r * h_prev).dot(Uh) + bh)  # Candidate hidden state
h = (1 - z) * h_prev + z * h_tilde  # New hidden state

# Print input, output, and hidden state update
print("Input Vector (X):", X)
print("Previous Hidden State (h_prev):", h_prev)
print("Update Gate (z):", z)
print("Reset Gate (r):", r)
print("Candidate Hidden State (h_tilde):", h_tilde)
print("New Hidden State (h):", h)
```