EXPERIMENT 1:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score

# Set random seed for reproducibility
np.random.seed(42)
# Generate synthetic data
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)

# Plot the data
plt.scatter(X, y, c='blue', marker='x')
plt.title("House Prices vs. Size")
plt.xlabel("Size (1000 sqft)")
plt.ylabel("Price (in 1000s of dollars)")
plt.show()

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train the linear regression model
model = LinearRegression()
model.fit(X_train, y_train)
# Predict on the test set
y_pred = model.predict(X_test)
# Calculate the mean squared error
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error: {mse}")
# Calculate and print accuracy using R² score
r2 = r2_score(y_test, y_pred)
print(f"Model Accuracy (R² Score): {r2 * 100:.2f}%")
# Print the model parameters
print(f"Intercept: {model.intercept_}")
print(f"Coefficient: {model.coef_}")

# Plot the model's predictions
plt.scatter(X_test, y_test, c='blue', marker='x', label='Actual')
plt.plot(X_test, y_pred, c='red', label='Predicted')
plt.title("House Prices vs. Size")
plt.xlabel("Size (1000 sqft)")
plt.ylabel("Price (in 1000s of dollars)")
plt.legend()
plt.show()
```

EXPERIMENT 2:

```
import numpy as np
# Logic Gate Helper Functions
def perceptronModel(x, w, b): return 1 if np.dot(w, x) + b >= 0 else 0
def NOT(x): return perceptronModel(x, np.array([-1]), 0.5)
# Logic Gate Functions
def AND(x): return perceptronModel(x, np.array([1, 1]), -1.5)
def OR(x): return perceptronModel(x, np.array([1, 1]), -0.5)
def NAND(x): return NOT(AND(x))
def NOR(x): return NOT(OR(x))
```

```
def XOR(x): return OR([AND([x[0], NOT([x[1]])]), AND([NOT([x[0]]), x[1]])])
def XNOR(x): return NOT(XOR(x))
# Test All Logic Gates
def test_gate(gate, name):
    tests = [np.array([0, 0]), np.array([0, 1]), np.array([1, 0]), np.array([1, 1])]
    print(f"\n{name} Gate Results:")
    for t in tests:
        print(f"{name}({t[0]}, {t[1]}) = {gate(t)}")
# Test Cases
test_gate(AND, "AND")
test_gate(OR, "OR")
test_gate(NAND, "NAND")
test_gate(NOR, "NOR")
test_gate(XOR, "XOR")
test_gate(XNOR, "XNOR")
```

EXPERIMENT 3:

```
import numpy as np
import matplotlib.pyplot as plt
# Dataset
x_train = np.array([1.0, 2.0])  # Input feature
y_train = np.array([300.0, 500.0])  # Target output
# Compute cost function
def compute_cost(x, y, w, b):
    return np.mean(((w * x + b) - y) ** 2) / 2
# Gradient Descent
def gradient_descent(x, y, w, b, alpha, n_iterations):
    m = len(x)  # Number of training samples
    cost_history = []  # To store cost values
    for i in range(n_iterations):
        y_pred = w * x + b  # Predicted values
        # Compute gradients
        dw = np.mean((y_pred - y) * x)  # Gradient w.r.t weight
        db = np.mean(y_pred - y)       # Gradient w.r.t bias
        # Update parameters
        w -= alpha * dw        b -= alpha * db
        # Store cost
        cost = compute_cost(x, y, w, b)        cost_history.append(cost)
        # Print progress every 10 iterations
        if i % 10 == 0 or i == n_iterations - 1:
            print(f"Iteration {i}: Cost = {cost:.4f}, w = {w:.4f}, b = {b:.4f}")
    return w, b, cost_history
# Accuracy metrics
def calculate_metrics(y_true, y_pred):
    mae = np.mean(np.abs(y_true - y_pred))  # Mean Absolute Error
    rmse = np.sqrt(np.mean((y_true - y_pred) ** 2))  # Root Mean Squared Error
    return mae, rmse

# Hyperparameters
alpha = 0.01  # Learning rate
n_iterations = 100  # Number of iterations
w_init = 0  # Initial weight         b_init = 0  # Initial bias
# Train the model
w_opt, b_opt, cost_history = gradient_descent(x_train, y_train, w_init, b_init, alpha, n_iterations)
# Predictions
y_pred_train = w_opt * x_train + b_opt
# Calculate accuracy metrics
mae, rmse = calculate_metrics(y_train, y_pred_train)
```

```
# Output optimized parameters and metrics
print(f"\nOptimized parameters: w = {w_opt:.4f}, b = {b_opt:.4f}")
print(f"Mean Absolute Error (MAE): {mae:.4f}")
print(f"Root Mean Squared Error (RMSE): {rmse:.4f}")

# Plot cost history
plt.plot(range(n_iterations), cost_history, label='Cost (MSE)')
plt.xlabel('Iterations')
plt.ylabel('Cost (MSE)')
plt.title('Cost over Iterations')
plt.legend()
plt.grid(True)
plt.show()
```

EXPERIMENT 4:

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
# Load and normalize CIFAR-10 dataset
(train_images,train_labels),(test_images,test_labels)=datasets.cifar10.loaddata()
train_images, test_images = train_images / 255.0, test_images / 255.0

# Define the CNN model
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10)])
# Compile the model
model.compile(
    optimizer='adam',
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=['accuracy'])
# Train the model
history = model.fit(train_images, train_labels, epochs=10, validation_data=(test_images, test_labels), verbose=2)
# Evaluate the model and print accuracy
test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=0)
print(f"Test Accuracy: {test_acc:.4f}")
# Plot training and validation accuracy
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.title('Accuracy Over Epochs')
plt.show()
```

EXPERIMENT 5:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense
from tensorflow.keras.preprocessing.sequence import pad_sequences
import matplotlib.pyplot as plt
```

```
# Sample text data
text = "hello world"
# Create a character-to-index mapping
chars = sorted(list(set(text)))
char_to_index = {char: idx for idx, char in enumerate(chars)}
index_to_char = {idx: char for idx, char in enumerate(chars)}

# Convert text to sequences of integers
sequences = [char_to_index[char] for char in text]
# Prepare input-output pairs
X = []    y = []      seq_length = 3
for i in range(len(sequences) - seq_length):
    X.append(sequences[i:i + seq_length])
    y.append(sequences[i + seq_length])
X = np.array(X)
y = np.array(y)
# Reshape X to be [samples, time steps, features] and normalize
X = np.reshape(X, (X.shape[0], X.shape[1], 1))
X = X / float(len(chars))
# One-hot encode the output
y = tf.keras.utils.to_categorical(y, num_classes=len(chars))

# Define and compile the RNN model
model = Sequential([SimpleRNN(50, input_shape=(seq_length, 1)), Dense(len(chars), activation='softmax')])
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
# Train the model and store the training history
history = model.fit(X, y, epochs=200, verbose=1)
# Function to predict the next character
def predict_next_char(model, input_text, char_to_index, index_to_char, seq_length):
    input_seq = [char_to_index[char] for char in input_text]
    input_seq = pad_sequences([input_seq], maxlen=seq_length, truncating='pre')
    input_seq = np.reshape(input_seq, (1, seq_length, 1))
    input_seq = input_seq / float(len(chars))
    predicted_index = np.argmax(model.predict(input_seq, verbose=0))
    return index_to_char[predicted_index]

# Test the prediction
input_text = "hel"
predicted_char = predict_next_char(model, input_text, char_to_index, index_to_char, seq_length)
print(f"Input: {input_text}, Predicted next character: {predicted_char}")
# Plot training loss and accuracy
plt.figure(figsize=(12, 4))
# Plot loss
plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='Loss')
plt.title('Training Loss')plt.xlabel('Epochs')plt.ylabel('Loss')plt.legend()
# Plot accuracy
plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], label='Accuracy')
plt.title('Training Accuracy')plt.xlabel('Epochs')plt.ylabel('Accuracy')plt.legend()plt.show()
```