The Science of Deep Learning

Iddo Drori

Massachusetts Institute of Technology

Columbia University

Cambridge University Press, 2022.

1 Programming Exercises & Solutions

2 Forward and Backpropagation

Problem 1. Fully Connected Neural Network
In this exercise, you will implement a fully connected neural network using Python and NumPy. Your task is to implement the following functions:

- `init_weights(n_inputs, n_hidden, n_output)`

  This function should randomly initialize the neural network's weights using the normal distribution. It should return the weight matrices `W0`, `W1`, and `W2` for the input-to-hidden, hidden-to-hidden, and hidden-to-output layers, respectively. The number of input, hidden, and output units should be passed as arguments to the function.
- `feedforward(x, W0, W1, W2)`

  This function should implement the feedforward operation of the neural network. It should take as input an example `x` and the weight matrices `W0`, `W1`, and `W2`, and return the pre-activations `z0`, `z1`, and `z2`, and the activations `a0`, `a1`, and `a2` for the input, hidden, and output layers, respectively. The feedforward operation should consist of matrix multiplications and pointwise application of the non-linear function f, which can be chosen as the sigmoid function.
- `predict(x, W0, W1, W2)`

  This function should take as input an example `x` and the weight matrices `W0`, `W1`, and `W2` and return the prediction of the neural network for that example. This can be done by passing the example `x` through the feedforward operation and returning the network's output.
- `train(X_train, Y_train, n_inputs, n_hidden, n_output, n_epochs, learning_rate)`

  This function should train the neural network using the training data `X_train` and `Y_train`. The number of input, hidden, and output units should be passed as arguments, along with the number of training epochs and the learning rate. The function should return the trained weight matrices `W0`, `W1`, and `W2`.

You can test your implementation using the following code snippet:

```
# Generate toy data
X_train = np.random.randn(1000, 10)
Y_train = np.random.randn(1000, 3)
```

```python
# Initialize network
n_inputs = 10
n_hidden = 5
n_output = 3
W0, W1, W2 = init_weights(n_inputs, n_hidden, n_output)

# Train the network
W0, W1, W2 = train(X_train, Y_train, n_inputs, n_hidden, n_output,
    n_epochs=10, learning_rate=0.1)

# Test the network
x_test = np.random.randn(1, 10)
y_pred = predict(x_test, W0, W1, W2)
print(y_pred)
```

Answer:

```python
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def init_weights(n_inputs, n_hidden, n_output):
    W0 = np.random.randn(n_inputs, n_hidden)
    W1 = np.random.randn(n_hidden, n_hidden)
    W2 = np.random.randn(n_hidden, n_output)
    return W0, W1, W2

def feedforward(x, W0, W1, W2):
    z0 = np.dot(x, W0)
    a0 = sigmoid(z0)
    z1 = np.dot(a0, W1)
    a1 = sigmoid(z1)
    z2 = np.dot(a1, W2)
    a2 = sigmoid(z2)
    return z0, a0, z1, a1, z2, a2

def predict(x, W0, W1, W2):
    _,_, _, _, _, a2 = feedforward(x, W0, W1, W2)
    return a2

def backprop(X_train, Y_train, W0, W1, W2, learning_rate):
    m = len(X_train)
    for x, y in zip(X_train, Y_train):
        z0, a0, z1, a1, z2, a2 = feedforward(x, W0, W1, W2)
        dz2 = a2 - y
        dw2 = np.dot(a1.T, dz2) / m
        dz1 = np.dot(dz2, W2.T) * sigmoid(z1) * (1 - sigmoid(z1))
        dw1 = np.dot(a0.T, dz1) / m
        dz0 = np.dot(dz1, W1.T) * sigmoid(z0) * (1 - sigmoid(z0))
        dw0 = np.dot(x.T, dz0) / m
        W0 -= learning_rate * dw0
        W1 -= learning_rate * dw1
        W2 -= learning_rate * dw2
        return W0, W1, W2
```

```
def loss(Y_pred, Y):
    return np.mean((Y_pred - Y) ** 2)

def main(X_train, Y_train, n_inputs, n_hidden, n_output, n_epochs,
     learning_rate):
    W0, W1, W2 = init_weights(n_inputs, n_hidden, n_output)
    for epoch in range(n_epochs):
    W0, W1, W2 = backprop(X_train, Y_train, W0, W1, W2, learning_rate)
    Y_pred = predict(X_train, W0, W1, W2)
    train_loss = loss(Y_pred, Y_train)
    print(f'epoch {epoch}, train loss {train_loss}')
    return W0, W1, W2, train_loss
```

Problem 2. Forward Propagation

- Define a function `forward_propagation(W, A, f)` that takes in the following parameters:
  - `W`: a matrix of weights for a layer
  - `A`: the activation matrix for the previous layer
  - `f`: the activation function for the current layer
- The function should first augment the weight matrix `W` by appending the bias vector as the last column and update the `A` matrix by appending a column of ones.
- Next, compute the matrix product of the augmented `W` and `A` matrices and save the result in a variable `Z`.
- Apply the activation function `f` element-wise on `Z` and save the result in a variable `A_out`.
- Return the `A_out` matrix.
- Test your function by initializing a weight matrix
  `W = [[1, 2], [3, 4]]`
  an activation matrix
  `A = [[1, 2], [3, 4], [1, 1]]`
  and an activation function
  `f = lambda x: x ** 2.`
  - Your function should return a matrix
    `[[7, 10], [15, 22]].`
- Bonus: Allow the function to handle multiple layers by allowing for a list of weight matrices and activation functions, and loop through the layers, updating the activation matrix for each one.

Answer:

```
import numpy as np

# Define the linear function
def linear_function(W, A_prev):
    Z = np.dot(W.T, A_prev)
    return Z

# Define the non-linear function
def non_linear_function(Z, activation_function):
    A = activation_function(Z)
    return A

# Define the forward propagation function
def forward_propagation(W, A_prev, activation_function):
    Z = linear_function(W, A_prev)
    A = non_linear_function(Z, activation_function)
    return A
```

```
# Example usage:
W = np.array([[w11, w12], [w21, w22], [w31, w32]])
A_prev = np.array([a1, a2, a3])
b = np.array([b1, b2])

# Append bias vector to W
W = np.append(W, b.reshape(-1,1), axis=1)

# Append bias term to A_prev
A_prev = np.append(A_prev, [1])

# Define activation function
activation_function = np.sigmoid

# Perform forward propagation
A = forward_propagation(W, A_prev, activation_function)

# Output: A = sigmoid(W.T * A_prev)
```

In this example, the forward propagation function takes in the weight matrix `W`, the activation matrix from the previous layer `A_prev`, and the chosen activation function. The bias vector `b` is appended to the weight matrix `W` and the bias term is appended to the activation matrix `A_prev`. The forward propagation function then performs the linear function, which is the matrix multiplication of `W.T` and `A_prev`, and the non-linear function, which applies the chosen activation function to the output of the linear function.

Problem 3. Non-linear Activation Functions

Implement the sigmoid activation function in Python using the equation provided in the text.

- Define a function named sigmoid that takes in a variable `z`.
- Inside the function, calculate the value of `1/(1 + e**(-z))` and store it in a variable named `result`.
- Return the value of `result`.
- Test the function by calling it with the input `z = 0` and print the result. The output should be 0.5.
- Test the function again with the input `z = 2` and print the result. The output should be close to 0.88.
- Test the function one more time with the input `z = -2` and print the result. The output should be close to 0.12.

Answer:

```python
def sigmoid(z):
    result = 1/(1 + e**(-z))
    return result

print(sigmoid(0)) # Output should be 0.5
print(sigmoid(2)) # Output should be close to 0.88
print(sigmoid(-2)) # Output should be close to 0.12
```

Problem 4. Hyperbolic Tangent Activation Function

Implement the hyperbolic tangent function in Python.

- Create a function called `tanh(z)` that takes in a parameter `z`.
- Inside the function, use the equation provided in the text to calculate the value of the hyperbolic tangent function for the input `z`.
- Return the calculated value.

Test your implementation by calling the `tanh()` function with the following inputs:

- `z = 0`
- `z = 1`
- `z = -1`
- `z = 2`
- `z = -2`

Print the output of each function call to check if it matches the expected output. Bonus: Plot the hyperbolic tangent function using `matplotlib` to visualize the function.

Answer:

```
import numpy as np

def tanh(z):
    return (np.exp(z) - np.exp(-z)) / (np.exp(z) + np.exp(-z))

# Test the function
print(tanh(0)) # should print 0
print(tanh(1)) # should be close to 0.761594
print(tanh(-1)) # should be close to -0.761594
print(tanh(np.inf)) # should print 1
print(tanh(-np.inf)) # should print -1
```

Problem 5. Rectified Linear Unit Activation Function

Implement the Rectified Linear Unit (ReLU) activation function in Python.

- Define a function `relu(z)` that takes in a scalar or array-like input `z` and returns the ReLU activation by applying the ReLU function to each element of the input.
- Use the function to apply the ReLU activation to a scalar, a list, and a numpy array, and print the results.

Answer:

```python
import numpy as np

def relu(z):
    return np.maximum(0, z)

# Test the function with a scalar
z = -5
print(relu(z)) # Output: 0

# Test the function with a list
z = [-5, 0, 5]
print(relu(z)) # Output: [0, 0, 5]

# Test the function with a numpy array
z = np.array([-5, 0, 5])
print(relu(z)) # Output: [0 0 5]
```

Problem 6. Softmax Activation Function

- Write a Python function called `softmax(z)` that takes in a vector `z` and returns the softmax of that vector. The function should use the equation provided in the text above.
- Test your function using the following test cases:
  - `z = [1,2,3]`
  - `z = [0,1,2,3]`
  - `z = [-1,0,1]`
- Use the softmax function to predict the class of a given input `x` using the following steps:
  - Create a matrix `W` of shape `(k, n)`, where `k` is the number of classes and `n` is the number of features in the input `x`. The matrix should be initialized with random values.
  - Compute the dot product of `W` and `x`, and pass the result to the softmax function.
  - The output of the softmax function is a vector of probabilities for each class. The index of the highest probability is the predicted class.
  - Implement this in a function called `predict(W, x)` which takes in the matrix `W` and the input vector `x`, and returns the predicted class.
- Test your predict function using the following test cases:
  - `W = [[1,2,3], [4,5,6], [7,8,9]] and x = [0,1,2]`
  - `W = [[1,2,3], [4,5,6], [7,8,9]] and x = [-1,0,1]`
  - `W = [[1,2,3], [4,5,6], [7,8,9]] and x = [1,0,1]`

Answer:

```python
import numpy as np

def softmax(z):
    exp_z = np.exp(z)
    return exp_z / exp_z.sum()

# Test cases
print(softmax([1,2,3]))
print(softmax([0,1,2,3]))
print(softmax([-1,0,1]))

def predict(W, x):
    z = np.dot(W, x)
    prob = softmax(z)
    return np.argmax(prob)

# Test cases
W = [[1,2,3], [4,5,6], [7,8,9]]
x = [0,1,2]
print(predict(W, x))

W = [[1,2,3], [4,5,6], [7,8,9]]
```

```python
x = [-1,0,1]
print(predict(W, x))

W = [[1,2,3], [4,5,6], [7,8,9]]
x = [1,0,1]
print(predict(W, x))
```

Problem 7. Loss Functions

In this exercise, you will implement a simple neural network and compute the loss function.

- Create a `class NeuralNetwork` with a method
  `forward_propagation(self, X, W)`
  that takes in inputs `X` and weights `W` and returns the predicted output labels `Y_hat`. The forward propagation can be defined as follows:
  `Y_hat = sigmoid(X * W)`
- Implement a method
  `loss_function(self, Y, Y_hat)`
  that takes in ground truth labels `Y` and predicted output labels `Y_hat` and returns the mean squared error loss. The loss function can be defined as follows:
  `loss = np.mean((Y - Y_hat) ** 2)`
- Implement a method
  `train(self, X, Y, W, learning_rate)`
  that takes in inputs `X`, ground truth labels `Y`, initial weights `W`, and a learning rate. The method should perform one iteration of gradient descent to update the weights `W`. The gradient of the loss function with respect to the weights can be defined as:
  ```
  dW = -2 * X.T @ (Y - Y_hat)
  W = W - learning_rate * dW
  ```
- Create an instance of the NeuralNetwork class and initialize the weights `W` to random values. Then, run the train method for a certain number of iterations using some training data `X` and labels `Y`.
- Print out the final value of the loss function.

Answer:

```python
# Initialize Neural Network
nn = NeuralNetwork()

# Generate some random data
X = np.random.randn(10, 2)
Y = np.random.randn(10, 1)

# Initialize weights
W = np.random.randn(2, 1)

# Set the learning rate
learning_rate = 0.1

# Train the neural network
for i in range(1000):
    Y_hat = nn.forward_propagation(X, W)
    loss = nn.loss_function(Y, Y_hat)
    nn.train(X, Y, W, learning_rate)
```

```python
# Print final loss
print("Final Loss: ", loss)
```

Problem 8. Backpropagation

Implement a simple neural network using the backpropagation algorithm for training. Your neural network should have 1 input layer, 1 hidden layer, and 1 output layer. The input layer should have 2 neurons, the hidden layer should have 3 neurons, and the output layer should have 1 neuron. The training data should consist of 4 input-output pairs. The inputs should be 2-dimensional and the outputs should be 1-dimensional. Use the sigmoid activation function for all layers.

Answer:

```python
import numpy as np

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

# Derivative of sigmoid function
def sigmoid_derivative(x):
    return x * (1 - x)

# Initialize the weights randomly between -1 and 1
weights_input_to_hidden = np.random.uniform(-1, 1, (2, 3))
weights_hidden_to_output = np.random.uniform(-1, 1, (3, 1))

# Training data
inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
outputs = np.array([[0], [1], [1], [0]])

# Training loop
for iteration in range(10000):
    # Forward pass
    hidden_inputs = np.dot(inputs, weights_input_to_hidden)
    hidden_outputs = sigmoid(hidden_inputs)
    final_inputs = np.dot(hidden_outputs, weights_hidden_to_output)
    final_outputs = sigmoid(final_inputs)

    # Backward pass
    error = outputs - final_outputs
    error_term = error * sigmoid_derivative(final_outputs)
    hidden_error = np.dot(error_term, weights_hidden_to_output.T)
    hidden_error_term = hidden_error * sigmoid_derivative(hidden_outputs)

    # Update weights
    weights_hidden_to_output += np.dot(hidden_outputs.T, error_term)
    weights_input_to_hidden += np.dot(inputs.T, hidden_error_term)

# Print the final weights
print(weights_input_to_hidden)
print(weights_hidden_to_output)
```

Problem 9. Gradient Descent

Implement a simple gradient descent algorithm to minimize a given scalar function `f`.

```python
def gradient_descent(f, x0, alpha=0.01, max_iter=1000, tol=1e-6):
    """
    Minimize a scalar function using gradient descent algorithm.

    Parameters:
        f : function
            scalar function to be minimized
        x0 : float
            initial value
        alpha : float
            step size
        max_iter : int
            maximum number of iterations
        tol : float
            tolerance for stopping criterion

    Returns:
        x : float
            optimal value
        f_opt : float
            optimal function value
        n_iter : int
            number of iterations performed
    """
    pass
```

Answer:

```python
import numpy as np

def gradient_descent(f, x0, alpha=0.01, max_iter=1000, tol=1e-6):
    x = x0
    for n_iter in range(max_iter):
        # Compute gradient
        grad = np.gradient(f(x))

        # Update x
        x = x - alpha * grad

        # Check stopping criterion
        if np.linalg.norm(grad) < tol:
            break

    return x, f(x), n_iter
```

Problem 10. Initialization and Normalization

Given a dataset of input values, x, write a Python function that normalizes the data using the standard score method, and returns the normalized data. The function should also initialize the weights of the network using the normal distribution $\frac{N(0,1)}{\sqrt{n}}$ for the weights.

Answer:

```
import numpy as np

def normalize_and_initialize(x, n_prev, n_curr):
    # Normalize the data
    mu = np.mean(x)
    sigma = np.sqrt(np.var(x))
    x = (x - mu) / sigma

    # Initialize weights
    weight_init = np.random.normal(0, 1/np.sqrt(n), (n_prev, n_curr))

    return x, weight_init
```

3 Optimization

Problem 11. Step Size

- Implement a function `gradient_descent(f, x0, grad_f, alpha, max_iter)` that performs gradient descent on a given function `f` with a given initial point `x0`, gradient of the function `grad_f`, learning rate `alpha`, and maximum number of iterations `max_iter`. The function should return the final point `x` and the function value at that point.
- Implement a function `find_optimal_alpha(f, x0, grad_f, alphas, max_iter)` that takes a function `f`, initial point `x0`, gradient of the function `grad_f`, a list of learning rates `alphas`, and maximum number of iterations `max_iter`. The function should apply the `gradient_descent()` function with each learning rate in the list and return the learning rate that results in the lowest function value at the final point.

Answer:

```python
import numpy as np

def gradient_descent(f, x0, grad_f, alpha, max_iter):
    x = x0
    for i in range(max_iter):
        x = x - alpha*grad_f(x)
    return x, f(x)

def find_optimal_alpha(f, x0, grad_f, alphas, max_iter):
    best_alpha = 0
    best_f_val = float('inf')
    for alpha in alphas:
        x, f_val = gradient_descent(f, x0, grad_f, alpha, max_iter)
        if f_val < best_f_val:
            best_alpha = alpha
            best_f_val = f_val
    return best_alpha
```

Problem 12. Mini-Batch Gradient Descent

Write a function that implements mini-batch gradient descent for a simple linear regression model. The function should take in the following parameters:

- `X`: A 2D `numpy` array of shape `(n_samples, n_features)` representing the input data.
- `y`: A 1D `numpy` array of shape `(n_samples,)` representing the target values.
- `batch_size`: An integer representing the size of the mini-batch.
- `learning_rate`: A float representing the learning rate.
- `num_iterations`: An integer representing the number of iterations to run mini-batch gradient descent.

The function should return the following:

- A list of the cost at each iteration.
- The final weight vector.

Answer:

```python
import numpy as np

def mini_batch_gradient_descent(X, y, batch_size, learning_rate,
     num_iterations):
    n_samples, n_features = X.shape
    weight = np.random.randn(n_features) # initialize random weights
    cost_history = []

    for i in range(num_iterations):
        for j in range(0, n_samples, batch_size):
            X_batch = X[j:j+batch_size]
            y_batch = y[j:j+batch_size]

            y_pred = X_batch @ weight
            error = y_pred - y_batch
            cost = np.mean(error ** 2) / 2
            cost_history.append(cost)

            gradient = X_batch.T @ error / n_samples
            weight -= learning_rate * gradient

    return cost_history, weight
```

Problem 13. Stochastic Gradient Descent

- Create a function that takes in the following parameters:
  - A list of input data points (x)
  - A list of corresponding labels (y)
  - A learning rate (alpha)
  - The number of iterations (num_iter)
- Initialize a weight vector with random values.
- Implement the stochastic gradient descent algorithm as follows.
- For each iteration:
  - Shuffle the input data points and labels
  - For each data point and corresponding label:
    - Compute the gradient of the objective function with respect to the weight vector using the current data point and label
    - Update the weight vector using the computed gradient and the learning rate
- Return the final weight vector.

Answer:

```python
import numpy as np
from sklearn.utils import shuffle

def stochastic_gradient_descent(x, y, alpha, num_iter):
    # Initialize weight vector with random values
    weight_vector = np.random.rand(x.shape[1])

    for _ in range(num_iter):
        # Shuffle data points and labels
        x, y = shuffle(x, y)

        for i in range(len(x)):
            # Compute gradient
            gradient = x[i] * (np.dot(x[i], weight_vector) - y[i])

            # Update weight vector
            weight_vector -= alpha * gradient

    return weight_vector
```

Problem 14. Momentum

Implement gradient descent with momentum in Python. Your solution should take in the following parameters:

- An initial weight vector (`w_init`)
- A list of training `examples`, where each example is a tuple of the form (input, output)
- A learning rate (`alpha`)
- A momentum parameter (`beta`)
- The number of iterations to run (`num_iters`)

The function should return the final weight vector after the specified number of iterations.

Answer:

```python
def gradient_descent_with_momentum(w_init, examples, alpha, beta,
     num_iters):
    # Initialize the weight vector and the velocity vector
    w = w_init
    v = [0 for _ in range(len(w_init))]

    # Run gradient descent for the specified number of iterations
    for i in range(num_iters):
        # Compute the gradient for the current weight vector
        grad = compute_gradient(w, examples)

        # Update the velocity vector
        v = [beta*v_i + grad_i for v_i, grad_i in zip(v, grad)]

        # Update the weight vector
        w = [w_i - alpha*v_i for w_i, v_i in zip(w, v)]

    return w
```

Problem 15. Adagrad

- Implement a function `adagrad(x_init, grad_func, alpha_init, beta, epsilon, max_iter)` that takes in the following parameters:
  - `x_init`: Initial values for `x`
  - `grad_func`: A function that computes the gradient of the objective function at a given point `x`
  - `alpha_init`: Initial value for the learning rate
  - `beta`: The decay rate for the accumulation of gradients
  - `epsilon`: A small value used to prevent division by zero
  - `max_iter`: The maximum number of iterations to perform
- The function should return the final value of `x` and the value of the objective function at that point after performing the Adagrad optimization algorithm.

Answer:

```python
import numpy as np

def adagrad(x_init, grad_func, alpha_init, beta, epsilon, max_iter):
    x = x_init
    s = 0
    for i in range(max_iter):
        grad = grad_func(x)
        s = beta * s + grad ** 2
        alpha = alpha_init / (np.sqrt(s) + epsilon)
        x = x - alpha * grad
    return x, grad_func(x)

def grad_func(x):
    return 2 * x

x_init = 10
alpha_init = 0.1
beta = 0.9
epsilon = 1e-8
max_iter = 100

final_x, final_val = adagrad(x_init, grad_func, alpha_init, beta,
     epsilon, max_iter)
print("Final x:", final_x)
print("Value of objective function at final x:", final_val)
```

Problem 16. Adam: Adaptive Moment Estimation

Implement the Adam optimization algorithm in Python. The function should take in the following inputs:

- `f`: a function that computes the value of the objective function
- `f_grad`: a function that computes the gradient of the objective function
- `x0`: the initial point
- `beta1`: the first moment decay rate
- `beta2`: the second moment decay rate
- `eps`: a small constant to prevent division by zero
- `num_iters`: the number of iterations to run the algorithm

The function should output:

- `x_opt`: the optimal point found by the algorithm
- `f_opt`: the value of the objective function at the optimal point

Answer:

```
def adam(f, f_grad, x0, beta1=0.9, beta2=0.99, eps=1e-8, num_iters=100):
    x = x0
    m = 0
    v = 0
    for i in range(num_iters):
        g = f_grad(x)
        m = beta1 * m + (1 - beta1) * g
        v = beta2 * v + (1 - beta2) * g**2
        m_hat = m / (1 - beta1**(i+1))
        v_hat = v / (1 - beta2**(i+1))
        x = x - (alpha / (np.sqrt(v_hat) + eps)) * m_hat
    x_opt = x
    f_opt = f(x_opt)
    return x_opt, f_opt
```

Problem 17. Newton's Method

Implement Newton's method in Python for finding the root of a given function. Your solution should take in the following parameters:

- An initial guess (`x_init`)
- A function `f`
- The derivative of `f` (`dfdx`)
- The number of iterations to run (`num_iters`)
- A tolerance level (`tol`)

The function should return the root of the function within the specified tolerance level after the specified number of iterations.

Answer:

```python
def newton_method(x_init, f, dfdx, num_iters, tol):
    x_prev = x_init
    for i in range(num_iters):
        x_curr = x_prev - f(x_prev) / dfdx(x_prev)
        if abs(x_curr - x_prev) < tol:
            return x_curr
        x_prev = x_curr
    return x_prev
```

Problem 18. Second-Order Taylor Approximation

Implement a function in Python that finds the minimum of a multivariate function using the second-order Taylor approximation. Your solution should take in the following parameters:

- An initial guess for the function's minimum (`x_init`)
- A function `f`
- The gradient of `f` (`grad_f`)
- The Hessian matrix of `f` (`hess_f`)
- The number of iterations to run (`num_iters`)
- A tolerance level (`tol`)

The function should return the minimum of the function within the specified tolerance level after the specified number of iterations.

Answer:

```python
import numpy as np
def second_order_taylor(x_init, f, grad_f, hess_f, num_iters, tol):
    x_prev = x_init
    for i in range(num_iters):
        gradient = grad_f(x_prev)
        hessian = hess_f(x_prev)
        x_curr = x_prev - np.linalg.inv(hessian).dot(gradient)
        if np.linalg.norm(x_curr - x_prev) < tol:
            return x_curr
        x_prev = x_curr
    return x_prev
```

Problem 19. Quasi-Newton Methods

Implement a Quasi-Newton method in Python that finds the minimum of a convex quadratic function. Your solution should take in the following parameters:

- An initial guess for the function's minimum (`x_init`)
- The Hessian matrix of the function (`H`)
- The gradient vector of the function (`b`)
- The constant term of the function (`c`)
- The number of iterations to run (`num_iters`)
- A tolerance level (`tol`)

The function should return the minimum of the function within the specified tolerance level after the specified number of iterations.

Answer:

```
import numpy as np
def quasi_newton(x_init, H, b, c, num_iters, tol):
    x_prev = x_init
    for i in range(num_iters):
        gradient = H.dot(x_prev) + b
        x_curr = x_prev - np.linalg.inv(H).dot(gradient)
        if np.linalg.norm(x_curr - x_prev) < tol:
            return x_curr
        x_prev = x_curr
    return x_prev
```

Problem 20. Evolution Strategies

Implement the Evolution Strategies algorithm using the cross-entropy method.

- Write a function that takes as input the dimension of the search space, the number of samples, the number of iterations, and the function to optimize. The function should return the optimal point in the search space and the optimal value of the function.
- Test the implemented algorithm on the sphere function: $f(x) = \sum_{i=1}^{n} x_i^2$.

Answer:

```
import numpy as np

def evolution_strategies(dim, num_samples, num_iterations, func):
    mean = np.zeros(dim)
    covariance = np.eye(dim)
    for i in range(num_iterations):
        samples = np.random.multivariate_normal(mean, covariance,
            num_samples)
        values = [func(sample) for sample in samples]
        best_samples = samples[np.argsort(values)[:int(num_samples/2)]]
        mean = np.mean(best_samples, axis=0)
        covariance = np.cov(best_samples.T)
    return mean, func(mean)

def sphere(x):
    return np.sum(x**2)

dim = 10
num_samples = 100
num_iterations = 50

optimal_point, optimal_value = evolution_strategies(dim, num_samples,
    num_iterations, sphere)
print("Optimal point:", optimal_point)
print("Optimal value:", optimal_value)
```

4 Regularization

Problem 21. Generalization

- Implement a neural network model using any deep learning library.
- Split the data into training and test sets.
- Train the model on the training set and calculate the training error.
- Evaluate the model on the test set and calculate the test error.
- Calculate the generalization gap.
- Experiment with different model architectures, regularization techniques and hyperparameters to reduce the generalization gap.

Answer:

```
import tensorflow as tf
from sklearn.model_selection import train_test_split

# load data
(X, Y) = load_data()

# split data into train and test sets
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2)

# define model architecture
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(32, input_shape=(X_train.shape[1],)))
model.add(tf.keras.layers.Activation('relu'))
model.add(tf.keras.layers.Dense(1))
model.add(tf.keras.layers.Activation('sigmoid'))

# compile model
model.compile(optimizer='adam', loss='binary_crossentropy',
    metrics=['accuracy'])

# train model
model.fit(X_train, Y_train, epochs=10, batch_size=32)

# evaluate model on train set
train_loss, train_acc = model.evaluate(X_train, Y_train)

# evaluate model on test set
test_loss, test_acc = model.evaluate(X_test, Y_test)

# calculate generalization gap
generalization_gap = train_loss - test_loss
print("Generalization gap:", generalization_gap)
```

Problem 22. Overfitting

- Create a neural network model with a varying number of layers and nodes.
- Train the model on a given dataset and calculate the training error and test error for each network complexity.
- Plot the training error and test error as a function of network complexity.
- Identify the point where the test error begins to increase and determine if the model is overfitting.

Answer:

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from keras.models import Sequential
from keras.layers import Dense
import matplotlib.pyplot as plt

# Generate a random classification dataset
X, y = make_classification(n_samples=1000, n_features=20, n_classes=2,
    random_state=1)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=1)

# Initialize lists to store training and test errors
training_errors = []
test_errors = []

# Loop through different network complexities
for i in range(1, 6):
    # Create a neural network with i layers and i*10 nodes
    model = Sequential()
    model.add(Dense(i*10, input_dim=20, activation='relu'))
    for j in range(1, i):
        model.add(Dense(i*10, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer='adam',
        metrics=['accuracy'])
    # Train the model and calculate the training error
    model.fit(X_train, y_train, epochs=50, batch_size=32, verbose=0)
    y_train_pred = model.predict(X_train)
    training_error = mean_squared_error(y_train, y_train_pred)
    training_errors.append(training_error)
    # Calculate the test error
    y_test_pred = model.predict(X_test)
    test_error = mean_squared_error(y_test, y_test_pred)
    test_errors.append(test_error)

# Plot the training and test errors
plt.plot(range(1, 6), training_errors, label='Training Error')
plt.plot(range(1, 6), test_errors, label='Test Error')
plt.xlabel('Network Complexity')
plt.ylabel('Error')
plt.legend()
```

```
plt.show()

# Identify the point where the test error begins to increase
for i in range(1, len(test_errors)):
    if test_errors[i] > test_errors[i-1]:
        print("Overfitting occurs at network complexity", i+1)
        break
```

Problem 23. Cross Validation

- Write a function in Python that performs `k`-fold cross validation on a given dataset. The function should take in the following parameters:
  - `data`: a list of `(x, y)` tuples where `x` is a feature vector and `y` is the target variable
  - `model`: a callable object that takes in `x` and `y` and returns a trained model
  - `k`: the number of folds for cross validation
  - `metrics`: a callable object that takes in a trained model and a test set and returns a scalar evaluation metric (e.g. accuracy)
- Use the above function to perform cross validation on a sample dataset using a simple linear regression model.
  - The sample dataset should contain at least 10 samples
  - The features should be randomly generated using `numpy`
  - The target variable should be `y = 2x + 1 + noise` where `noise` is randomly generated
  - Use mean squared error as the evaluation metric

Answer:

```
import numpy as np
from sklearn.metrics import mean_squared_error
from sklearn.linear_model import LinearRegression

# Helper function to generate sample dataset
def generate_data(n):
    x = np.random.rand(n, 1)
    noise = np.random.normal(0, 0.1, n)
    y = 2*x + 1 + noise
    return x, y

# Helper function to perform k-fold cross validation
def cross_validate(data, model, k, metrics):
    n = len(data)
    fold_size = int(n / k)
    errors = []
    for i in range(k):
        test_start = i * fold_size
        test_end = (i+1) * fold_size
        test_data = data[test_start:test_end]
        train_data = data[:test_start] + data[test_end:]
        x_train, y_train = zip(*train_data)
        x_test, y_test = zip(*test_data)
        trained_model = model(x_train, y_train)
        test_error = metrics(trained_model, x_test, y_test)
        errors.append(test_error)
    return np.mean(errors), np.var(errors)

# Generate sample dataset
x, y = generate_data(10)
data = list(zip(x, y))
```

```python
# Define linear regression model
def linear_regression_model(x, y):
    model = LinearRegression()
    model.fit(x, y)
    return model

# Define evaluation metric
def mse(model, x, y):
    y_pred = model.predict(x)
    return mean_squared_error(y, y_pred)

# Perform k-fold cross validation
k = 5
mean_error, var_error = cross_validate(data, linear_regression_model, k,
    mse)
print("Mean Error:", mean_error)
print("Variance of Errors:", var_error)
```

Problem 24. Cross Validation

Implement a k-fold cross validation for a simple linear regression model. The dataset consists of 100 data points with 1 input feature and 1 output value. The goal is to select the best value for the regularization parameter.

Create a function called `cross_validate(X, y, k, regularization_values)` that takes as input the feature matrix X, the target vector y, the number of folds k and a list of `regularization_values` to be tested. The function should return the best regularization value and the corresponding mean and variance of the generalization error.

Answer:

```
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import KFold

def cross_validate(X, y, k, regularization_values):
    kf = KFold(n_splits=k, shuffle=True)
    best_value = None
    best_score = float('inf')
    scores = []
    for reg_value in regularization_values:
        fold_scores = []
        for train_index, test_index in kf.split(X):
            X_train, X_test = X[train_index], X[test_index]
            y_train, y_test = y[train_index], y[test_index]
            model = LinearRegression(fit_intercept=True, normalize=False,
                copy_X=True)
            model.fit(X_train, y_train)
            y_pred = model.predict(X_test)
            fold_scores.append(mean_squared_error(y_test, y_pred))
        mean_score = np.mean(fold_scores)
        var_score = np.var(fold_scores)
        scores.append((mean_score, var_score))
        if mean_score < best_score:
            best_score = mean_score
            best_value = reg_value
    return best_value, scores
```

Problem 25. Bias and Variance

- Create a simple neural network with one hidden layer containing 8 neurons and train it on a given dataset. Measure the training error and test error.
- Increase the width of the hidden layer by adding 4 more neurons and retrain the network on the same dataset. Measure the training error and test error.
- Add another hidden layer with 8 neurons to the network and retrain it on the same dataset. Measure the training error and test error.
- Implement L1 or L2 regularization in the network and retrain it on the same dataset. Measure the training error and test error.
- Compare the training error and test error for all the above models and analyze how the bias and variance are affected by increasing the width and depth of the network and by regularization.

Answer:

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from keras.models import Sequential
from keras.layers import Dense
from keras.regularizers import l1, l2

# Generate a random dataset
X, y = make_classification(n_samples=1000, n_features=20, n_classes=2)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Initialize the neural network
model = Sequential()
model.add(Dense(8, input_dim=20, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam',
    metrics=['accuracy'])

# Train the network and measure the training and test error
model.fit(X_train, y_train, epochs=50, batch_size=32)
train_error = 1 - accuracy_score(y_train, model.predict(X_train).round())
test_error = 1 - accuracy_score(y_test, model.predict(X_test).round())
print("Training error: ", train_error)
print("Test error: ", test_error)

# Increase the width of the hidden layer
model = Sequential()
model.add(Dense(12, input_dim=20, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam',
    metrics=['accuracy'])

# Retrain the network and measure the training and test error
model.fit(X_train, y_train, epochs=50, batch_size=32)
train_error = 1 - accuracy_score(y_train, model.predict(X_train).round())
test_error = 1 - accuracy_score(y_test, model.predict(X_test).round())
```

```
print("Training error: ", train_error)
print("Test error: ", test_error)

# Add another hidden layer
model = Sequential()
model.add(Dense(8, input_dim=20, activation='relu'))
model.add(Dense(8, activation='relu'))
```

Problem 26. Vector Norms

- Implement a function `vector_norm(x, norm)` that takes in a vector `x` and a string norm representing the type of norm to be used. The function should return the norm of the vector `x`. The possible values for norm are `"L1"`, `"L2"`, and `"Max"`.
- Implement a function `regularize(x, norm, lambda)` that takes in a vector `x`, a string norm representing the type of norm to be used, and a scalar `lambda`. The function should return the regularized vector `x` by adding the `lambda` multiplied by the norm of the vector `x` to the original vector.
- Test your functions with the following vectors:
  - `x1 = [1, 2, 3, 4, 5]`
  - `x2 = [1, -2, 3, -4, 5]`
  - `x3 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`
- Compare the results obtained from the different norms and observe the effect of regularization on each norm.

Answer:

```
import numpy as np

def vector_norm(x, norm):
    if norm == "L1":
        return np.sum(np.abs(x))
    elif norm == "L2":
        return np.sqrt(np.sum(np.square(x)))
    elif norm == "Max":
        return np.max(np.abs(x))
    else:
        raise ValueError("Invalid norm type")

def regularize(x, norm, lambda):
    return x + lambda * vector_norm(x, norm)

x1 = [1, 2, 3, 4, 5]
x2 = [1, -2, 3, -4, 5]
x3 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print("Vector norm of x1 using L1: ", vector_norm(x1, "L1"))
print("Vector norm of x1 using L2: ", vector_norm(x1, "L2"))
print("Vector norm of x1 using Max: ", vector_norm(x1, "Max"))
print("Regularized x1 using L1 with lambda = 0.1: ", regularize(x1,
    "L1", 0.1))
print("Regularized x1 using L2 with lambda = 0.1: ", regularize(x1,
    "L2", 0.1))
print("Regularized x1 using Max with lambda = 0.1: ", regularize(x1,
    "Max", 0.1))

print("Vector norm of x2 using L1: ", vector_norm(x2, "L1"))
print("Vector norm of x2 using L2: ", vector_norm(x2, "L2"))
print("Vector norm of x2 using Max: ", vector_norm(x2, "Max"))
```

```
print("Regularized x2 using L1 with lambda = 0.1: ", regularize(x2,
     "L1", 0.1))
print("Regularized x2 using L2 with lambda = 0.1: ", regularize(x2,
     "L2", 0.1))
print("Regularized x2 using Max with lambda =
```

Problem 27. Ridge Regression

- Implement a linear regression model using ridge regression as the objective function.
- Generate a synthetic dataset with 100 samples, where the input feature has 10 dimensions and the output label is a scalar value.
- Split the dataset into a training set (80 samples) and a test set (20 samples).
- Train the model using the training set and different values of the regularization hyperparameter `lambda` (e.g. $0.1, 1, 10$).
- Evaluate the model on the test set and report the mean squared error (MSE) for each value of `lambda`.
- Plot the MSE on the test set as a function of `lambda`.

Answer:

```
import numpy as np
from sklearn.datasets import make_regression
from sklearn.linear_model import Ridge
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

# Generate synthetic data
X, y = make_regression(n_samples=100, n_features=10, noise=0.1)

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

# Define a list of lambda values for regularization
lambdas = [0.1, 1, 10]

# Initialize a list to store MSE for each value of lambda
mse_test = []

# Loop over lambda values
for l in lambdas:
    # Initialize and fit the model
    model = Ridge(alpha=l)
    model.fit(X_train, y_train)

    # Make predictions on the test set
    y_pred = model.predict(X_test)

    # Compute MSE
    mse = mean_squared_error(y_test, y_pred)

    # Append MSE to the list
    mse_test.append(mse)

# Plot MSE as a function of lambda
plt.plot(lambdas, mse_test)
plt.xlabel('Lambda')
plt.ylabel('MSE on test set')
```

```
plt.show()
```

Problem 28. Ridge Regression

- Create a function `ridge_regression(X, Y, lambda_)` that takes in matrices `X` and `Y` and a scalar `lambda_` and returns the optimal parameters theta that minimize the ridge regression objective.
- Create a function `predict(X, theta)` that takes in a matrix `X` and `theta` and returns the predicted labels.
- Create a function `evaluate(X, Y, theta)` that takes in a matrix `X`, vector `Y` and `theta`, and returns the mean squared error of the predictions.
- Create a function `ridge_regression_evaluate(X, Y, lambda_)` that takes in matrices `X` and `Y` and a scalar `lambda_`, and returns the optimal `theta` found by ridge regression and the mean squared error of the predictions on the input data.

Answer:

```
import numpy as np

def ridge_regression(X, Y, lambda_):
    I = np.eye(X.shape[1])
    return np.linalg.inv(X.T.dot(X) + lambda_*I).dot(X.T).dot(Y)

def predict(X, theta):
    return X.dot(theta)

def evaluate(X, Y, theta):
    return np.mean((predict(X, theta) - Y)**2)

def ridge_regression_evaluate(X, Y, lambda_):
    theta = ridge_regression(X, Y, lambda_)
    return theta, evaluate(X, Y, theta)
```

Problem 29. Lasso Regression

Implement Lasso regression using the `numpy` library in Python.

- Your function should take in the following inputs: a matrix of feature data `X`, a vector of labels `Y`, a regularization parameter `lambda`, and `num_iterations` to run the gradient descent algorithm.
- Your function should output the optimal values of `theta`, the coefficients of the model.

Answer:

```python
import numpy as np

def lasso_regression(X, Y, reg_param, num_iterations):
    m, n = X.shape
    theta = np.zeros(n)
    for i in range(num_iterations):
        prediction = np.dot(X, theta)
        error = prediction - Y
        gradient = (2/m)*np.dot(X.T, error) + reg_param*np.sign(theta)
        theta = theta - gradient
    return theta
```

Problem 30. Regularized Loss Functions

- Implement a function `regularized_loss(W, X, Y, lam, p)` that calculates the regularized loss for a given weight matrix `W`, input data `X`, output labels `Y`, regularization parameter `lam` and the p-norm `p`.
- Test the function with a weight matrix `W`, input data `X`, output labels `Y`, regularization parameter `lam = 0.1` and `p = 2`.

Answer:

```
import numpy as np

def regularized_loss(W, X, Y, lam, p):
    m = X.shape[0]
    loss = 1/m * np.sum((Y - np.dot(X, W))**2)
    regularization = lam * np.linalg.norm(W, ord=p)
    return loss + regularization

# Test
W = np.random.rand(5, 3)
X = np.random.rand(100, 5)
Y = np.random.rand(100, 3)
lam = 0.1
p = 2

print(regularized_loss(W, X, Y, lam, p))
```

Problem 31. Regularized Loss Functions

Implement a simple linear regression model with L1 regularization. The dataset consists of 100 data points with 2 input features and 1 output value. The goal is to train the model using stochastic gradient descent (SGD) and use L1 regularization with different regularization strength values.

Create a function called `train_l1_regression(X, y, regularization_strengths, n_iter)` that takes as input the feature matrix `X`, the target vector `y`, a list of `regularization_strengths` and the number of SGD iterations `n_iter`. The function should return a dictionary that contains the model's weight coefficients for each regularization strength value.

Answer:

```python
import numpy as np
from sklearn.linear_model import SGDRegressor

def train_l1_regression(X, y, regularization_strengths, n_iter):
    models = {}
    for reg_strength in regularization_strengths:
        model = SGDRegressor(penalty='l1', alpha=reg_strength,
            max_iter=n_iter)
        model.fit(X, y)
        models[reg_strength] = model.coef_
    return models
```

Problem 32. Dropout Regularization

Create a simple neural network with one input layer, one hidden layer, and one output layer using a library of your choice (e.g., TensorFlow, Keras, PyTorch).

- Implement dropout regularization on the hidden layer with a dropout rate of 0.5.
- Train the network on a dataset of your choice and evaluate its performance using accuracy as the metric.

Answer:

```
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.optimizers import Adam
from keras.datasets import mnist

# Load the dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# Normalize the data
X_train = X_train.astype('float32') / 255
X_test = X_test.astype('float32') / 255

# Create the model
model = Sequential()
model.add(Dense(64, input_shape=(784,), activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

# Compile the model
model.compile(optimizer=Adam(), loss='categorical_crossentropy',
    metrics=['accuracy'])

# Train the model
model.fit(X_train, y_train, epochs=10, batch_size=32)

# Evaluate the model
score = model.evaluate(X_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

Problem 33. Dropout Regularization

Implement a dropout regularization technique for a simple feedforward neural network. The network should have one hidden layer with 100 units, and use sigmoid activation functions. The network should be trained using stochastic gradient descent (SGD) with a dropout rate of 0.5 for the hidden layer.

Create a function called `train_dropout_nn(X, y, n_iter, dropout_rate)` that takes as input the feature matrix `X`, the target vector `y`, the number of SGD iterations `n_iter`, and the `dropout_rate`. The function should return the trained model.

Answer:

```
import numpy as np
from sklearn.neural_network import MLPClassifier

def train_dropout_nn(X, y, n_iter, dropout_rate):
    model = MLPClassifier(hidden_layer_sizes=(100,),
        activation='logistic', solver='sgd', max_iter=n_iter,
        dropout=dropout_rate)
    model.fit(X, y)
    return model
```

Problem 34. Random Least Squares

Implement a function called random_least_squares_dropout(X, y, p) that takes in a matrix of features X, a vector of labels y, and a probability p and returns the solution for beta using the random least squares with dropout method. You should use the equation: $\hat{\beta} = (X^T X + \frac{p}{1+p} D)^{-1} X^T y$, where D is the diagonal matrix of X with the ith entry being the norm of the ith column of X.

Answer:

```
import numpy as np

def random_least_squares_dropout(X, y, p):
    D = np.diag(np.linalg.norm(X, axis=0)**2)
    beta_hat = np.linalg.inv(X.T @ X + (p/(1+p))*D) @ X.T @ y
    return beta_hat

X = np.array([[1,2,3], [4,5,6], [7,8,9]])
y = np.array([1,2,3])
p = 0.5

beta_hat = random_least_squares_dropout(X, y, p)
print(beta_hat)
```

Problem 35. Least Squares with Noise Input Distortion

- Create a synthetic dataset with 1000 samples and 100 features.
- Split the dataset into training and testing set with the ratio of 80:20.
- Implement least squares with noise input distortion by adding random noise to the input with `N(0, lambda)` and fit the model to the training data.
- Test the model on the testing data and calculate the mean squared error (MSE).
- Try different values of lambda and find the optimal value that gives the lowest MSE on the testing data.

Answer:

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# Create a synthetic dataset
np.random.seed(0)
X = np.random.randn(1000, 100)
y = np.random.randn(1000)

# Split the dataset into training and testing set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Add random noise to the input
noise_std = 0.1
X_train_noise = X_train + noise_std * np.random.randn(X_train.shape[0],
    X_train.shape[1])

# Fit the model to the training data
beta = np.linalg.inv(X_train_noise.T @ X_train_noise) @ X_train_noise.T
    @ y_train

# Test the model on the testing data
y_pred = X_test @ beta
mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error:", mse)

# Try different values of lambda
lambdas = [0.01, 0.1, 1, 10, 100]
best_lambda = None
best_mse = None
for l in lambdas:
    noise_std = l
    X_train_noise = X_train + noise_std *
        np.random.randn(X_train.shape[0], X_train.shape[1])
    beta = np.linalg.inv(X_train_noise.T @ X_train_noise) @
        X_train_noise.T @ y_train
    y_pred = X_test @ beta
    mse = mean_squared_error(y_test, y_pred)
    if best_mse is None or mse < best_mse:
        best_lambda = l
        best_mse = mse
```

```python
print("Best lambda:", best_lambda)
print("Lowest MSE:", best_mse)
```

Problem 36. Data Augmentation

- Given a dataset of images of cars, create a data augmentation function that rotates the images by a random angle between -15 and 15 degrees.
- Use the augmented dataset to train a neural network and compare the performance to a network trained on the original dataset.

Answer: To implement the data augmentation function, we can use the Python Imaging Library (PIL) to open the images and rotate them by the desired angle. Here is an example implementation:

```
from PIL import Image
import numpy as np

def augment_data(image_path):
    # Open the image
    image = Image.open(image_path)
    # Rotate the image by a random angle
    angle = np.random.uniform(-15, 15)
    image = image.rotate(angle)
    # Save the augmented image
    image.save(image_path)
```

To use the augmented dataset to train a neural network, we can apply the data augmentation function to each image in the dataset before training. Here is an example of how this might be done using the Keras library:

```
from keras.preprocessing.image import ImageDataGenerator

# Apply the data augmentation function to each image in the dataset
for image_path in image_paths:
    augment_data(image_path)

# Create a data generator for the augmented dataset
data_gen = ImageDataGenerator()

# Create the neural network
model = ...

# Train the network using the augmented dataset
model.fit_generator(data_gen.flow(X_train, y_train, batch_size=32),
                    steps_per_epoch=len(X_train) / 32, epochs=10)
```

After training the neural network with an augmented dataset, we can compare the performance of the network to one that was trained on the original dataset. We can compare the accuracy and loss of both the models and check which model is performing better.

Problem 37. Data Augmentation

- Create a function `augment_data(X_train, y_train, B)` that takes in a training dataset `X_train` and `y_train` and a number of augmentations `B` as input. The function should return a new dataset that is augmented with `B` new data points for each original data point.
- The new dataset should be created by applying random transformations to each original data point. The random transformations can include rotation, reflection, translation, shear, crop, color transformation, and added noise.
- The new dataset should be in the same format as the original dataset, with the same number of features and the same labels.
- The function should be able to handle a variety of data types, including images and text.

Answer:

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from skimage.transform import rotate, warp, AffineTransform

def augment_data(X_train, y_train, B):
    X_augmented = []
    y_augmented = []
    for i in range(X_train.shape[0]):
        for b in range(B):
            # Randomly apply a transformation
            transformed_X = random_transformation(X_train[i])
            X_augmented.append(transformed_X)
            y_augmented.append(y_train[i])
    X_augmented = np.array(X_augmented)
    y_augmented = np.array(y_augmented)
    return X_augmented, y_augmented

def random_transformation(X):
    # Randomly choose a transformation
    transformation = np.random.choice(["rotate", "warp", "affine"])
    if transformation == "rotate":
        angle = np.random.uniform(-45, 45)
        X_transformed = rotate(X, angle)
    elif transformation == "warp":
        tform = warp(X, AffineTransform(scale=(0.8, 0.9)))
        X_transformed = tform
    else:
        tform = AffineTransform(scale=(0.8, 1.2))
        X_transformed = warp(X, tform)
    return X_transformed

X, y = make_classification(n_samples=100, n_features=20, n_classes=2)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
X_augmented, y_augmented = augment_data(X_train, y_train, B=5)
```

```
# Use the augmented data for training and test with the original test
    data
# model.fit(X_augmented, y_augmented)
# model.score(X_test, y_test)
```

Problem 38. Data Augmentation:

- Given a dataset of images of handwritten digits, implement a data augmentation function that takes as input an image and performs a random transformation on the image (e.g. rotation, reflection, translation, etc.).
- Use the function to generate a new dataset of augmented images by applying the function to each image in the original dataset.
- Train a neural network on the augmented dataset and compare its performance to a neural network trained on the original dataset.

Answer: Implementing a data augmentation function can be done using a library such as OpenCV or Pillow. Here is an example implementation using OpenCV:

```
import cv2
import numpy as np

def data_augmentation(image):
    # randomly choose a transformation
    transformation = np.random.randint(1, 5)
    if transformation == 1:
        # perform rotation
        angle = np.random.uniform(-45, 45)
        image = cv2.rotate(image, cv2.ROTATE_90_CLOCKWISE)
    elif transformation == 2:
        # perform reflection
        image = cv2.flip(image, 1)
    elif transformation == 3:
        # perform translation
        x_translation = np.random.randint(-5, 5)
        y_translation = np.random.randint(-5, 5)
        image = cv2.warpAffine(image,
            np.float32([[1,0,x_translation],[0,1,y_translation]]), (28,
            28))
    elif transformation == 4:
        # perform shear
        shear = np.random.uniform(-0.5, 0.5)
        image = cv2.warpAffine(image, np.float32([[1,shear,0],[0,1,0]]),
            (28, 28))

    return image
```

To generate the new dataset of augmented images, we can use the function in a loop, applying the function to each image in the original dataset:

```
augmented_images = []
for image in original_images:
    augmented_images.append(data_augmentation(image))
```

We can then use the augmented dataset to train a neural network, comparing its performance to a neural network trained on the original dataset:

```
from keras.models import Sequential
from keras.layers import Dense, Flatten

# train model on original dataset
model = Sequential()
```

```
model.add(Flatten(input_shape=(28, 28)))
model.add(Dense(128, activation='relu'))
model.add(Dense(10, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam',
    metrics=['accuracy'])
model.fit(original_images, labels, epochs=10)

# train model on augmented dataset
model2 = Sequential()
model2.add(Flatten(input_shape=(28, 28)))
model2.add(Dense(128, activation='relu'))
model2.add(Dense(10, activation='softmax'))
model2.compile(loss='categorical_crossentropy', optimizer='adam',
    metrics=
```

Problem 39. Batch Normalization

Implement the Batch Normalization technique in a neural network. The neural network will have at least one hidden layer and will be trained on a dataset of your choice. The goal is to observe the effect of Batch Normalization on the training process and the final accuracy of the model.

Answer:

```
# Import necessary libraries
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, BatchNormalization
from tensorflow.keras.optimizers import Adam

# Generate a toy dataset
X, y = make_classification(n_samples=1000, n_features=20, n_classes=2)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Create a neural network with one hidden layer and Batch Normalization
model = Sequential()
model.add(Dense(64, input_dim=20, activation='relu'))
model.add(BatchNormalization())
model.add(Dense(1, activation='sigmoid'))

# Compile and train the model
model.compile(optimizer=Adam(), loss='binary_crossentropy',
    metrics=['accuracy'])
history = model.fit(X_train, y_train, epochs=50,
    validation_data=(X_test, y_test))

# Evaluate the model on the test set
y_pred = model.predict(X_test)
y_pred = np.round(y_pred)
print("Test accuracy:", accuracy_score(y_test, y_pred))
```

Problem 40. Batch Normalization

- Implement a batch normalization function for a neural network that takes in the input data and batch size as parameters.
- Apply the batch normalization function to the input data before passing it through the network.
- Train the network using the normalized data and compare the results with a network trained without batch normalization.

Answer:

```
import numpy as np

def batch_normalization(input_data, batch_size):
    # Step 1: Calculate the mean and standard deviation of the input data
    mean = np.mean(input_data, axis=0)
    std = np.std(input_data, axis=0)

    # Step 2: Normalize the input data by subtracting the mean and
        dividing by the standard deviation
    normalized_data = (input_data - mean) / (std / np.sqrt(batch_size))

    return normalized_data

# Example usage
input_data = np.random.randn(100, 32)
batch_size = 32
normalized_data = batch_normalization(input_data, batch_size)

# Train a neural network with normalized data
model = NeuralNetwork()
model.train(normalized_data)

# Train a neural network without normalization
model_no_norm = NeuralNetwork()
model_no_norm.train(input_data)

# Compare the results
print("Accuracy with batch normalization:", model.evaluate())
print("Accuracy without batch normalization:", model_no_norm.evaluate())
```

5 Convolutional Neural Networks

Problem 41. Convolution

- Implement a function that takes in an `image` represented as a 2D matrix and a `kernel` represented as a 2D matrix and returns the `convolution` of the `image` with the `kernel`.
- Test your function on the following `image` and `kernel`:
  - Image: `[[1, 2, 3], [4, 5, 6], [7, 8, 9]]`
  - Kernel: `[[1, 0, -1], [2, 0, -2], [1, 0, -1]]`
  - Expected Output: `[[-8, -8, -8], [-8, -8, -8], [-8, -8, -8]]`

Answer:

```
import numpy as np

def convolution(image, kernel):
    # Get the dimensions of the image and kernel
    image_rows, image_cols = image.shape
    kernel_rows, kernel_cols = kernel.shape

    # Initialize a matrix to store the convolution
    convolution = np.zeros((image_rows - kernel_rows + 1, image_cols -
        kernel_cols + 1))

    # Perform convolution
    for i in range(image_rows - kernel_rows + 1):
        for j in range(image_cols - kernel_cols + 1):
            convolution[i][j] = np.sum(image[i:i+kernel_rows,
                j:j+kernel_cols] * kernel)

    return convolution

# Test the function
image = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
kernel = np.array([[1, 0, -1], [2, 0, -2], [1, 0, -1]])
print(convolution(image, kernel))
```

This will output:

```
[[-8. -8. -8.]
 [-8. -8. -8.]
 [-8. -8. -8.]]
```

Problem 42. Convolution

- Given two arrays, `f` and `g`, implement a function `convolution(f, g)` that computes the discrete one-dimensional convolution of the two arrays. The function should return the convolution as a new array.
- Test your function on the following `f` and `g`:
  - `f = [1, 2, 3]`
  - `g = [2, 3, 4]`
  - Expected Output: `[4, 9, 14, 8, 6]`

Answer:

```
def convolution(f, g):
    s = len(g) - 1
    convolution = []
    for i in range(len(f) + s):
        sum = 0
        for u in range(-s, s+1):
            if i-u < 0 or i-u >= len(f):
                continue
            sum += g[u+s] * f[i-u]
        convolution.append(sum)
    return convolution

f = [1, 2, 3]
g = [2, 3, 4]
print(convolution(f, g))
```

Problem 43. Convolution

- Create a function `convolution_matrix(k: List[float]) -> Tuple[np.ndarray, List[np.ndarray]]` that takes in a 1-D filter `k` and returns a tuple of the convolution matrix `K` and a list of matrices `S` such that the matrix-vector product `Kx` is the convolution operation `k` convolved with `x` and `S` is a list of matrices used in the computation of `K`.
- Create a function `convolution_derivative(x: np.ndarray, k: List[float]) -> List[np.ndarray]` that takes in a 1-D input vector `x` and a 1-D filter `k`, computes the convolution matrix `K` using the `convolution_matrix()` function, and returns a list of the derivative of the output `y` with respect to each of the kernel weights.

Answer:

```
import numpy as np
from typing import List, Tuple

def convolution_matrix(k: List[float]) -> Tuple[np.ndarray,
    List[np.ndarray]]:
    n = len(k)
    K = np.zeros((n, n))
    S = []
    for i in range(n):
        S.append(np.eye(n, k = i))
        K += k[i]*S[i]
    return K, S

def convolution_derivative(x: np.ndarray, k: List[float]) ->
    List[np.ndarray]:
    K, S = convolution_matrix(k)
    y = np.dot(K, x)
    dy_dk = [np.dot(S[i], x) for i in range(len(k))]
    return dy_dk
```

Example:

```
k = [1, 2, 3]
x = np.array([1, 2, 3])
K, S = convolution_matrix(k)
print(f'Convolution Matrix: \n{K}')
print(f'List of Matrices S: \n{S}')
dy_dk = convolution_derivative(x, k)
print(f'Derivative of y wrt k: \n{dy_dk}')
```

Output:

```
Convolution Matrix:
[[2. 3. 1.]
 [1. 2. 3.]
 [3. 1. 2.]]
List of Matrices S:
[array([[0., 1., 0.],
       [0., 0., 1.],
       [0., 0., 0.]]),
array([[1., 0., 0.],
```

```
        [0., 1., 0.],
        [0., 0., 1.]]),
array([[0., 0., 0.],
        [1., 0., 0.],
        [0., 1., 0.]])]
Derivative of y wrt k:
[array([1., 2., 3.]), array([1., 2., 3.]), array([1., 2., 3.])]
```

Problem 44. Convolution

Create a function called `conv1d(f, g, s)` that takes in two one-dimensional arrays, `f` and `g`, and an integer `s`. The function should return the one-dimensional convolution of the two arrays as defined in the equation: $(f \star g)(i) = \sum_{u=-s}^{s} g(u) f(i-u)$.

Answer:

```
def conv1d(f, g, s):
    conv = []
    for i in range(len(f)):
        sum = 0
        for u in range(-s, s+1):
            if i-u < 0 or i-u >= len(f):
                continue
            sum += g[u+s]*f[i-u]
        conv.append(sum)
    return conv
```

Problem 45. Convolution

Create a function called `conv2d_separable(f, g1, g2)` that takes in a two-dimensional array `f`, and two one-dimensional arrays `g1` and `g2`. The function should return the two-dimensional convolution of the array `f` with the outer product of `g1` and `g2`.

Answer:

```
def conv2d_separable(f, g1, g2):
    conv = []
    for i in range(len(f)):
        row = []
        for j in range(len(f[0])):
            sum = 0
            for u in range(-len(g1)//2, len(g1)//2+1):
                if i-u < 0 or i-u >= len(f):
                    continue
                for v in range(-len(g2)//2, len(g2)//2+1):
                    if j-v < 0 or j-v >= len(f[0]):
                        continue
                    sum += g1[u+len(g1)//2]*g2[v+len(g2)//2]*f[i-u][j-v]
            row.append(sum)
        conv.append(row)
    return conv
```

Problem 46. Convolution

Create a 1D convolution function that takes in two 1D arrays, `f` and `g`, and a parameter `s` representing the range of the convolution. The function should output the convolution of the two arrays according to the equation: $(f \star g)(i) = \sum_{u=-s}^{s} g(u)f(i-u)$.

- Create a 2D convolution function that takes in two 2D arrays and a kernel represented by a 2D array. The function should output the convolution of the two arrays using the kernel.
- Test the two functions using the following test cases:
  - `f = [1, 2, 3], g = [2, 3, 4], s = 1`
  - `f = [[1, 2], [3, 4]], g = [[2, 3], [4, 5]], kernel = [[1, 0], [0, 1]]`
- Verify that the two functions satisfy the property of commutativity by creating test cases and using the functions from steps 1 and 2.

Answer:

```
def conv1D(f, g, s):
    conv = []
    for i in range(len(f)):
        temp = 0
        for u in range(-s, s+1):
            if i-u < 0 or i-u >= len(f):
                continue
            temp += g[u] * f[i-u]
        conv.append(temp)
    return conv

def conv2D(f, g, kernel):
    import numpy as np
    conv = np.zeros_like(f)
    for i in range(f.shape[0] - kernel.shape[0] + 1):
        for j in range(f.shape[1] - kernel.shape[1] + 1):
            conv[i][j] = np.sum(f[i:i+kernel.shape[0],
                j:j+kernel.shape[1]] * kernel)
    return conv

# Test case for 1D convolution
f = [1, 2, 3]
g = [2, 3, 4]
s = 1
print(conv1D(f, g, s)) # Output: [8, 10, 12]

# Test case for 2D convolution
f = [[1, 2], [3, 4]]
g = [[2, 3], [4, 5]]
kernel = [[1, 0], [0, 1]]
print(conv2D(f, g, kernel)) # Output: [[6, 8], [10, 12]]

# Commutativity test
f = [1, 2, 3]
g = [2, 3, 4]
```

```
s = 1
assert conv1D(f, g, s) == conv1D(g, f, s)

f = [[1, 2], [3, 4]]
g = [[2, 3], [4, 5]]
kernel = [[1, 0], [0, 1]]
assert np.array_equal(conv2D(f, g, kernel), conv2D(g, f, kernel))
```

Problem 47. Convolution

- Create a function `repeated_convolution(image: np.ndarray, kernel_size: int, num_repeats: int) -> np.ndarray` that performs repeated convolution of an input `image` with a square kernel of a given size `kernel_size` and number of repeats `num_repeats`.
- Test the function using the following test case:
  - `image`: a 10×10 matrix of random values between 0 and 1
  - `kernel_size`: 3
  - `num_repeats`: 2
  - Expected Output: a 10×10 matrix resulting from two convolutions of the input `image` with a `kernel_size` of 3

Answer:

```
import numpy as np
from scipy.signal import convolve2d

def repeated_convolution(image: np.ndarray, kernel_size: int,
     num_repeats: int) -> np.ndarray:
    kernel = np.ones((kernel_size, kernel_size)) / (kernel_size *
        kernel_size)
    output = image.copy()
    for i in range(num_repeats):
        output = convolve2d(output, kernel, mode='same')
    return output

# test
np.random.seed(0)
image = np.random.rand(10, 10)
kernel_size = 3
num_repeats = 2
output = repeated_convolution(image, kernel_size, num_repeats)
print(output)
```

Problem 48. Convolution Layers

Create a Python function called `convolution_layers` that takes in the following parameters:

- `image`: a 3D `numpy` array representing a color image with shape `(n, n, 3)`.
- `filters`: a list of 4D `numpy` arrays representing the filters to be applied to the `image`. Each filter has shape `(k, k, 3, f)` where `k` is the filter size, and `f` is the number of filters.
- `padding`: a boolean value indicating whether or not padding should be applied to the image before convolution.

The function should perform the convolution operation on the input `image` with the specified `filters`, with or without `padding`, and return the resulting volume of activations with shape `(n, n, f)`.

Answer:

```python
import numpy as np
from scipy.signal import convolve2d

def convolution_layers(image, filters, padding=False):
    if padding:
        # Add padding to the image
        image = np.pad(image, ((1,1),(1,1),(0,0)), mode='constant')
    n, _, _, f = filters.shape
    activations = np.zeros((image.shape[0], image.shape[1], f))
    for i in range(n):
        for j in range(f):
            activations[:,:,j] += convolve2d(image[:,:,0],
                filters[i,:,:,j], mode='valid')
            activations[:,:,j] += convolve2d(image[:,:,1],
                filters[i,:,:,j], mode='valid')
            activations[:,:,j] += convolve2d(image[:,:,2],
                filters[i,:,:,j], mode='valid')
    return activations
```

Problem 49. Pooling Layer

Implement a function `max_pooling(image: np.ndarray, kernel_size: int) -> np.ndarray` that applies max pooling to a 2D grayscale `image` by taking the maximum value over a square kernel of size `kernel_size` and stride of `kernel_size`. The function should return the resulting pooled image.

Answer:

```
import numpy as np

def max_pooling(image: np.ndarray, kernel_size: int) -> np.ndarray:
    n, m = image.shape
    pooled_image = np.zeros((n//kernel_size, m//kernel_size))
    for i in range(0, n, kernel_size):
        for j in range(0, m, kernel_size):
            pooled_image[i//kernel_size, j//kernel_size] =
                np.max(image[i:i+kernel_size, j:j+kernel_size])
    return pooled_image
```

Example:

```
image = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14,
     15, 16]])
kernel_size = 2
print(max_pooling(image, kernel_size))
```

Output:

```
[[ 6.  8.]
 [14. 16.]]
```

6 Sequence Models

Problem 50. Bag of Words

- Write a function that takes in a list of `documents` and returns a dictionary `term_frequency` containing the term frequency of each word in the corpus.
- Write a function that takes in the term frequency dictionary `term_frequency` and the total number of documents `total_documents` and returns a dictionary `inverse_document_frequency` containing the inverse document frequency of each word in the corpus.
- Write a function that takes in the term frequency dictionary `term_frequency` and the inverse document frequency dictionary `inverse_document_frequency` and returns a dictionary `tf_idf` containing the TF-IDF values of each word in the corpus.

Answer:

```
import re
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer

stop_words = set(stopwords.words('english'))
stemmer = PorterStemmer()

def get_term_frequency(documents):
    term_frequency = {}
    for document in documents:
        # Normalize to lowercase and remove non-alphanumeric characters
        document = re.sub(r'[^a-zA-Z0-9]', ' ', document.lower())
        # Tokenize the document
        tokens = document.split()
        # Remove stop words and stem the tokens
        tokens = [stemmer.stem(token) for token in tokens if token not in
            stop_words]
        # Count the frequency of each token in the document
        for token in tokens:
            if token in term_frequency:
                term_frequency[token] += 1
            else:
                term_frequency[token] = 1
    return term_frequency

def get_inverse_document_frequency(term_frequency, total_documents):
    inverse_document_frequency = {}
    for token, frequency in term_frequency.items():
        inverse_document_frequency[token] = 1 + log(total_documents /
            frequency)
    return inverse_document_frequency

def get_tf_idf(term_frequency, inverse_document_frequency):
    tf_idf = {}
    for token, frequency in term_frequency.items():
        tf_idf[token] = frequency * inverse_document_frequency[token]
```

```
    return tf_idf
```

Problem 51. Bag of Words

Write a program that takes a list of `sentences` and a `query` sentence as input, and outputs the sentence from the list that is most similar to the query sentence using TF-IDF. Your program should tokenize the sentences, calculate the TF-IDF weight for each word in each sentence, and use the cosine similarity to measure the similarity between the query sentence and each sentence in the list.

Answer:

```python
import math
from collections import defaultdict
from typing import List, Tuple

def tokenize(sentence: str) -> List[str]:
    # Tokenize the sentence by splitting on whitespace and punctuation
    return sentence.split()

def tf(word: str, sentence: List[str]) -> float:
    # Calculate the term frequency of a word in a sentence
    return sentence.count(word) / len(sentence)

def idf(word: str, sentences: List[List[str]]) -> float:
    # Calculate the inverse document frequency of a word across a list of
        sentences
    n = len(sentences)
    df = sum(1 for sentence in sentences if word in sentence)
    return math.log(n / df)

def tfidf(word: str, sentence: List[str], sentences: List[List[str]]) ->
    float:
    # Calculate the TF-IDF weight of a word in a sentence
    return tf(word, sentence) * idf(word, sentences)

def cosine_similarity(vec1: dict, vec2: dict) -> float:
    # Calculate the cosine similarity between two vectors represented as
        dictionaries
    dot_product = sum(vec1.get(word, 0) * vec2.get(word, 0) for word in
        set(vec1).union(vec2))
    norm1 = math.sqrt(sum(vec1.get(word, 0) ** 2 for word in vec1))
    norm2 = math.sqrt(sum(vec2.get(word, 0) ** 2 for word in vec2))
    return dot_product / (norm1 * norm2)

def most_similar_sentence(query: str, sentences: List[str]) ->
    Tuple[str, float]:
    # Tokenize the sentences
    tokenized_sentences = [tokenize(sentence) for sentence in sentences]
    tokenized_query = tokenize(query)
    # Calculate the TF-IDF weight for each word in each sentence
    sentence_vectors = [defaultdict(float) for _ in tokenized_sentences]
    for i, sentence in enumerate(tokenized_sentences):
        for word in sentence:
            sentence_vectors[i][word] = tfidf(word, sentence,
                tokenized_sentences)
    # Calculate the TF-IDF weight for each word in the query sentence
    query_vector = defaultdict(float)
```

```
for word in tokenized_query:
    query_vector[word] = tfidf(word, tokenized_query,
        tokenized_sentences)
# Find the sentence with the highest cosine similarity to the query
    sentence
most_similar = max((cosine_similarity(query_vector, vector),
    sentence) for vector, sentence in zip(sentence_vectors,
    sentences))
return most_similar
```

Problem 52. Feature Vector

Write a program that takes a list of `sentences` as input, and outputs a list of `feature_vectors`, one for each sentence. Each feature vector should be a list of integers representing the position of each word in the sentence, with the first word being at position 0.

Your program should also include a function that takes two feature vectors `vec1` and `vec2` as input, and outputs the distance between them using the Levenshtein distance metric, which measures the minimum number of single-character edits (insertions, deletions, or substitutions) required to change one sentence into the other.

Answer:

```python
from typing import List

def feature_vectors(sentences: List[str]) -> List[List[int]]:
    feature_vectors = []
    for sentence in sentences:
        words = sentence.split()
        feature_vector = [words.index(word) for word in words]
        feature_vectors.append(feature_vector)
    return feature_vectors

def levenshtein_distance(vec1: List[int], vec2: List[int]) -> int:
    m, n = len(vec1), len(vec2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]
    for i in range(m + 1):
        for j in range(n + 1):
            if i == 0:
                dp[i][j] = j
            elif j == 0:
                dp[i][j] = i
            elif vec1[i - 1] == vec2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]
            else:
                dp[i][j] = 1 + min(dp[i][j - 1], dp[i - 1][j], dp[i - 1][j
                    - 1])
    return dp[m][n]

# Test
sentences = ["Alice sent a message on Sunday", "On Sunday Alice sent a
    message"]
feature_vectors_list = feature_vectors(sentences)
print(feature_vectors_list)
# Output: [[0, 1, 2, 3, 4, 5], [5, 0, 1, 2, 3, 4]]
print(levenshtein_distance(feature_vectors_list[0],
    feature_vectors_list[1]))
# Output: 2
```

Problem 53. N-grams

Write a program that takes in a string of `text` and an integer `n` as input, and outputs a dictionary `ngrams` containing all the n-grams from the text and the number of times they appear in the text.

Your program should also include a function that takes in a string of `text` and an integer `k` as input, and outputs the probability of each word in the text given the previous k words.

Answer:

```
from collections import defaultdict
from typing import Dict

def get_ngrams(text: str, n: int) -> Dict[str, int]:
    ngrams = defaultdict(int)
    words = text.split()
    for i in range(len(words) - (n - 1)):
        ngram = ' '.join(words[i:i+n])
        ngrams[ngram] += 1
    return ngrams

def get_word_probabilities(text: str, k: int) -> Dict[str, float]:
    word_probs = defaultdict(float)
    words = text.split()
    kgrams = get_ngrams(text, k + 1)
    for kgram, count in kgrams.items():
        word = kgram.split()[-1]
        prefix = ' '.join(kgram.split()[:-1])
        prefix_count = sum(val for key, val in kgrams.items() if
            key.startswith(prefix))
        word_probs[word] = count / prefix_count
    return word_probs

# Test
text = "the quick brown fox jumps over the lazy dog"
print(get_ngrams(text, 2))
# Output: {"the quick": 1, "quick brown": 1, "brown fox": 1, "fox
    jumps": 1, "jumps over": 1, "over the": 1, "the lazy": 1, "lazy
    dog": 1}
print(get_word_probabilities(text, 2))
# Output: {"quick": 0.5, "brown": 0.5, "fox": 0.5, "jumps": 0.5, "over":
    0.5, "the": 1.0, "lazy": 0.5, "dog": 1.0}
```

Problem 54. Markov Model

Implement a function `markov_model(sentence: str) -> dict` that takes in a `sentence` and returns a dictionary `model` that represents the Markov model of the `sentence`. The keys in the dictionary should be the words in the `sentence` and the values should be a list of words that immediately follow the key word in the `sentence`.

Implement a function `predict_word(model: dict, word: str) -> str` that takes in the Markov `model` and a `word` and returns the word that is most likely to come next in the sentence. If multiple words have the same probability of coming next, return any of them.

Answer:

```python
from typing import List, Tuple, Dict

def markov_model(sentence: str) -> Dict[str, List[str]]:
    words = sentence.split()
    model = {}
    for i in range(len(words) - 1):
        if words[i] in model:
            model[words[i]].append(words[i + 1])
        else:
            model[words[i]] = [words[i + 1]]
    return model

def predict_word(model: dict, word: str) -> str:
    return max(model[word], key=model[word].count)

sentence = "Alice and Bob communicate. Alice sent Bob a message"
model = markov_model(sentence)
print(predict_word(model, "Alice")) # should return 'sent'
print(predict_word(model, "Bob")) # should return 'a'
```

Problem 55. State Machine

Create a class called `StateMachine` that implements the following methods:

- `__init__(self, states: List[str], inputs: List[str], transition: Dict [Tuple[str, str], str], outputs: List[str], mapping: Dict[str, str], initial_state: str) -> None`: Initializes the class with the following attributes:
  - `states`: A list of possible states.
  - `inputs`: A list of possible inputs.
  - `transition`: A dictionary that maps from a tuple of state and input to the next state.
  - `outputs`: A list of possible outputs.
  - `mapping`: A dictionary that maps from a state to an output.
  - `initial_state`: The initial state of the state machine.
- `get_next_state(self, state: str, input: str) -> str`: Given a `state` and an `input`, returns the next state according to the transition function.
- `get_output(self, state: str) -> str`: Given a `state`, returns the output according to the mapping function.
- `run(self, inputs: List[str]) -> List[str]`: Given a list of `inputs`, simulates the state machine and returns a list of outputs.

Example:

```
states = ['A', 'B', 'C']
inputs = ['a', 'b']
transition = {('A', 'a'): 'B', ('B', 'a'): 'C', ('B', 'b'): 'A', ('C',
    'b'): 'B'}
outputs = ['x', 'y']
mapping = {'A': 'x', 'B': 'y', 'C': 'x'}
initial_state = 'A'

sm = StateMachine(states, inputs, transition, outputs, mapping,
    initial\_state)
print(sm.get\_next\_state('A', 'a')) # B
print(sm.get\_output('B')) # y
print(sm.run(['a', 'b', 'a', 'b'])) # ['x', 'y', 'x', 'y']
```

Answer:

```
from typing import List, Tuple, Dict

class StateMachine:
    def __init__(self, states: List[str], inputs: List[str], transition:
        Dict[Tuple[str, str], str], outputs: List[str], mapping:
        Dict[str, str], initial_state: str) -> None:
        self.states = states
        self.inputs = inputs
        self.transition = transition
        self.outputs = outputs
        self.mapping = mapping
        self.initial_state = initial_state
```

```python
def get_next_state(self, state: str, input: str) -> str:
    return self.transition[(state, input)]

def get_output(self, state: str) -> str:
    return self.mapping[state]

def run(self, inputs: List[str]) -> List[str]:
    current_state = self.initial_state
    outputs = []
    for i in inputs:
        current_state = self.get_next_state(current_state, i)
```

Problem 56. Recurrent Neural Network

- Create a class `RNN` that takes in 3 matrices `U`, `W`, `V` and a non-linear activation function `g` as inputs.
- The class should have a method `process_sequence` that takes in a list of input `sequences` $x_1, x_2, \ldots, x_t$ and returns a list of `outputs` $y_1, y_2, \ldots, y_t$.
- The class should also have a method `predict` that takes in a single input `x` and returns a single output `y`.
- The class should keep track of the hidden state h_t and update it at each time step using the matrices `U`, `W`, and the non-linear activation function `g`.
- The class should use the matrices `V` to compute the output y_t.

Answer:

```
import numpy as np

class RNN:
    def __init__(self, U, W, V, g):
        self.U = U
        self.W = W
        self.V = V
        self.g = g
        self.h = None

    def process_sequence(self, sequences):
        self.h = np.zeros(self.W.shape[0])
        outputs = []
        for x in sequences:
            self.h = self.g(np.dot(self.W, self.h) + np.dot(self.U, x))
            y = np.dot(self.V, self.h)
            outputs.append(y)
        return outputs

    def predict(self, x):
        self.h = self.g(np.dot(self.W, self.h) + np.dot(self.U, x))
        y = np.dot(self.V, self.h)
        return y
```

Problem 57. Recurrent Neural Network

Given an input sequence `input_sequence` and a pre-trained RNN model, generate an output sequence `output_sequence` using the one-to-many mapping.

```
import numpy as np

def generate_output_sequence(input_sequence, model):
    output_sequence = []
    # Use the pre-trained model to generate an output sequence from the
        input sequence
    # Append the generated outputs to the output_sequence list
    # Return the output_sequence
    return output_sequence

Example:
input_sequence = np.random.rand(10, 256)
trained_model = ... # load a pre-trained RNN model
output_sequence = generate_output_sequence(input_sequence, trained_model)
print(output_sequence)
```

Answer:

```
import numpy as np

def generate_output_sequence(input_sequence, model):
    output_sequence = []
    # Use the pre-trained model to generate an output sequence from the
        input sequence
    hidden_state = model.init_hidden(input_sequence.shape[0])
    for i in range(input_sequence.shape[0]):
        output, hidden_state = model(input_sequence[i], hidden_state)
        output_sequence.append(output)
    # Return the output_sequence
    return output_sequence
```

Problem 58. Recurrent Neural Network

Implement a bidirectional RNN in a programming language of your choice. The network should take in a sequence of inputs and output a prediction for each timestep. The network should have the following architecture:

- An input layer that takes in a sequence of vectors of length `n`.
- A forward LSTM layer with `h` hidden units.
- A backward LSTM layer with `h` hidden units.
- A concatenation layer that concatenates the outputs from the forward and backward LSTM layers.
- A fully connected layer that outputs a prediction for each timestep.

The network should be trained on a dataset of sequences and corresponding labels. The loss function used for training should be the mean squared error between the network's predictions and the true labels.

Answer:

```
class BidirectionalRNN:
    def __init__(self, n, h):
        self.n = n
        self.h = h
        self.W = np.random.randn(h, h) # weight matrix for forward LSTM
            layer
        self.U = np.random.randn(h, n) # weight matrix for input layer
        self.V = np.random.randn(2*h, 1) # weight matrix for fully
            connected layer
        self.barW = np.random.randn(h, h) # weight matrix for backward
            LSTM layer

    def forward(self, x):
        T = len(x)
        h = np.zeros((T, self.h))
        for t in range(T):
            h[t] = np.tanh(self.W @ h[t-1] + self.U @ x[t])
        return h

    def backward(self, x):
        T = len(x)
        barh = np.zeros((T, self.h))
        for t in range(T-1, -1, -1):
            barh[t] = np.tanh(self.barW @ barh[t+1] + self.U @ x[t])
        return barh

    def predict(self, x):
        h = self.forward(x)
        barh = self.backward(x)
        concat = np.concatenate((h, barh), axis=1)
        o = self.V.T @ concat
        return o

    def train(self, x, y, lr):
        o = self.predict(x)
        error = y - o
```

```
dV = error @ np.concatenate((h, barh), axis=1).T
dW = error @ h[:-1].T
dbarW = error @ barh[1:].T
dU = error @ x.T
self.W += lr * dW
self.U += lr * dU
self.V += lr * dV
self.barW += lr * dbarW
```

Problem 59. Recurrent Neural Network

Implement a function `rnn_backpropagation(W, U, V, x, y, k, g, g_prime, e_prime)` that performs the RNN backward propagation through time algorithm described in the text. The function should take in the following inputs:

- `W`: The weight matrix for the hidden units moving forward in time
- `U`: The weight matrix for the input units
- `V`: The weight matrix for the output units
- `x`: The input sequence
- `y`: The true output sequence
- `k`: The length of the input and output sequences
- `g`: The non-linear activation function
- `g_prime`: The derivative of the non-linear activation function
- `e_prime`: The derivative of the error function

The function should return a tuple of the gradients for `W`, `U`, and `V`.

Answer:

```
def rnn_backpropagation(W, U, V, x, y, k, g, g_prime, e_prime):
    do = np.zeros((k, V.shape[0]))
    dV = np.zeros(V.shape)
    dh = np.zeros((k, W.shape[0]))
    dz = np.zeros((k, W.shape[0]))
    dU = np.zeros(U.shape)
    dW = np.zeros(W.shape)
    for t in range(k-1, -1, -1):
        o = np.dot(V, np.concatenate((h[t], h_bar[t])))
        do[t] = e_prime(o) * (y[t] - o)
        dV += np.dot(do[t].reshape(-1, 1), np.concatenate((h[t],
            h_bar[t]), axis=1).reshape(1, -1))
        dh[t] = np.dot(V.T, do[t])
        dz[t] = g_prime(np.dot(W, h[t-1]) + np.dot(U, x[t])) * dh[t]
        dU += np.dot(dz[t].reshape(-1, 1), x[t].reshape(1, -1))
        dW += np.dot(dz[t].reshape(-1, 1), h[t-1].reshape(1, -1))
        if t > 0:
            dh[t-1] = np.dot(W.T, dz[t])
    return dW, dU, dV
```

Problem 60. Recurrent Neural Network

Implement a basic RNN in your preferred programming language. The RNN should take in a sequence of `inputs` and a set of corresponding `outputs`, and it should be trained using backpropagation through time. The input data should be a sequence of vectors, and the output data should be a corresponding sequence of vectors. The RNN should use the `tanh` activation function and the `softmax` loss function.

Answer:

```
import numpy as np

class RNN:
    def __init__(self, input_size, hidden_size, output_size):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        self.W = np.random.randn(hidden_size, hidden_size)
        self.U = np.random.randn(hidden_size, input_size)
        self.V = np.random.randn(output_size, hidden_size)

    def forward(self, inputs):
        timesteps = len(inputs)
        h = np.zeros((timesteps+1, self.hidden_size))
        o = np.zeros((timesteps, self.output_size))

        for t in range(timesteps):
            h[t] = np.tanh(np.dot(self.W, h[t-1]) + np.dot(self.U,
                inputs[t]))
            o[t] = np.dot(self.V, h[t])

        return o

    def backward(self, inputs, outputs, learning_rate):
        timesteps = len(inputs)
        dV = np.zeros_like(self.V)
        dU = np.zeros_like(self.U)
        dW = np.zeros_like(self.W)
        dh = np.zeros((timesteps+1, self.hidden_size))
        do = np.zeros((timesteps, self.output_size))

        for t in range(timesteps-1, -1, -1):
            do[t] = (outputs[t] - self.forward(inputs)[t]) * (1 -
                np.tanh(self.forward(inputs)[t]) ** 2)
            dV += np.dot(do[t][:, None], h[t][None, :])
            dh[t] = np.dot(self.V.T, do[t]) + np.dot(self.W.T, dh[t+1])
            dU += np.dot(dh[t][:, None], inputs[t][None, :])
            dW += np.dot(dh[t][:, None], h[t-1][None, :])

        self.V += learning_rate * dV
        self.U += learning_rate * dU
        self.W += learning_rate * dW
```

Problem 61. Gated Recurrent Unit

- Implement a Gated Recurrent Unit (GRU) in Python using `numpy`.
- Train the GRU on a dataset of sequential data, such as a time series dataset or a language dataset.
- Compare the performance of the GRU to a traditional RNN on the same dataset.

Answer:

```python
import numpy as np
class GRU:
    def __init__(self, input_size, hidden_size):
        self.W_z = np.random.randn(input_size, hidden_size)
        self.U_z = np.random.randn(hidden_size, hidden_size)
        self.W_r = np.random.randn(input_size, hidden_size)
        self.U_r = np.random.randn(hidden_size, hidden_size)
        self.W_h = np.random.randn(input_size, hidden_size)
        self.U_h = np.random.randn(hidden_size, hidden_size)
    def forward(self, x, h):
        z = sigmoid(np.dot(x, self.W_z) + np.dot(h, self.U_z))
        r = sigmoid(np.dot(x, self.W_r) + np.dot(h, self.U_r))
        h_hat = np.tanh(np.dot(x, self.W_h) + np.dot(r * h, self.U_h))
        h = (1 - z) * h + z * h_hat
        return h
```

Problem 62. Gated Recurrent Unit

- Implement a Gated Recurrent Unit (GRU) in Python using `numpy`.
- Given a sequence of inputs and corresponding hidden states, the GRU should compute the output hidden states at each time step using the equations provided above.
- Implement a function that takes in the input sequence `inputs`, initial hidden state `h_prev`, and the weight matrices `Wz`, `Uz`, `Wr`, `Ur`, `W`, and `U` as inputs and returns the sequence of hidden states.
- The GRU should use `sigmoid` as the activation function for the update and reset gates, and `tanh` as the activation function for the candidate hidden state.

Answer:

```
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def tanh(x):
    return np.tanh(x)

def gru(inputs, h_prev, Wz, Uz, Wr, Ur, W, U):
    T = len(inputs)
    h_states = np.zeros((T+1, h_prev.shape[0]))
    h_states[0] = h_prev
    for t in range(T):
        x_t = inputs[t]
        z_t = sigmoid(np.dot(Wz, h_prev) + np.dot(Uz, x_t))
        r_t = sigmoid(np.dot(Wr, h_prev) + np.dot(Ur, x_t))
        h_tilde = tanh(np.dot(W, r_t * h_prev) + np.dot(U, x_t))
        h_states[t+1] = z_t * h_prev + (1 - z_t) * h_tilde
        h_prev = h_states[t+1]
    return h_states[1:]

inputs = np.random.rand(10, 3)
h_prev = np.random.rand(5)
Wz = np.random.rand(5, 5)
Uz = np.random.rand(5, 3)
Wr = np.random.rand(5, 5)
Ur = np.random.rand(5, 3)
W = np.random.rand(5, 5)
U = np.random.rand(5, 3)

h_states = gru(inputs, h_prev, Wz, Uz, Wr, Ur, W, U)
```

Problem 63. Long Short-Term Memory

Implement a simple LSTM cell in Python. The cell should take in the current input state x_t, the previous hidden state h_{t-1}, and previous memory cell c_{t-1} as inputs, and output the next hidden state h_t and memory cell c_t.

You can initialize the weights and biases of the LSTM cell randomly.

Answer:

```python
import numpy as np

class LSTM:
    def __init__(self, input_size, hidden_size):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.W_f = np.random.randn(hidden_size, input_size + hidden_size)
        self.W_i = np.random.randn(hidden_size, input_size + hidden_size)
        self.W_c = np.random.randn(hidden_size, input_size + hidden_size)
        self.W_o = np.random.randn(hidden_size, input_size + hidden_size)
        self.b_f = np.random.randn(hidden_size)
        self.b_i = np.random.randn(hidden_size)
        self.b_c = np.random.randn(hidden_size)
        self.b_o = np.random.randn(hidden_size)

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def forward(self, x, h, c):
        concat = np.concatenate((x, h), axis=1)
        f = self.sigmoid(np.dot(self.W_f, concat) + self.b_f)
        i = self.sigmoid(np.dot(self.W_i, concat) + self.b_i)
        c_ = np.tanh(np.dot(self.W_c, concat) + self.b_c)
        o = self.sigmoid(np.dot(self.W_o, concat) + self.b_o)
        c = f * c + i * c_
        h = o * np.tanh(c)
        return h, c

lstm = LSTM(input_size=3, hidden_size=4)
x = np.random.randn(3)
h = np.random.randn(4)
c = np.random.randn(4)
h_out, c_out = lstm.forward(x, h, c)
print(h_out)
print(c_out)
```

Problem 64. Long Short-Term Memory

- Implement a LSTM cell in Python using the equations provided in the text.
- Create a LSTM network with one LSTM cell and pass a sequence of input data through it.
- Print the output hidden state `h` and memory cell `c` at each time step.

Answer:

```
import numpy as np

class LSTMCell:
    def __init__(self, input_size, hidden_size):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.Wf = np.random.randn(hidden_size, input_size)
        self.Uf = np.random.randn(hidden_size, hidden_size)
        self.Wi = np.random.randn(hidden_size, input_size)
        self.Ui = np.random.randn(hidden_size, hidden_size)
        self.Wo = np.random.randn(hidden_size, input_size)
        self.Uo = np.random.randn(hidden_size, hidden_size)
        self.W = np.random.randn(hidden_size, input_size)
        self.U = np.random.randn(hidden_size, hidden_size)

    def forward(self, x, h, c):
        ft = sigmoid(np.dot(self.Wf, x) + np.dot(self.Uf, h))
        it = sigmoid(np.dot(self.Wi, x) + np.dot(self.Ui, h))
        ot = sigmoid(np.dot(self.Wo, x) + np.dot(self.Uo, h))
        c_tilda = np.tanh(np.dot(self.W, x) + np.dot(self.U, h))
        c = ft * c + it * c_tilda
        h = ot * np.tanh(c)
        return h, c

class LSTM:
    def __init__(self, input_size, hidden_size):
        self.lstm_cell = LSTMCell(input_size, hidden_size)

    def forward(self, x):
        h, c = np.zeros((hidden_size, 1)), np.zeros((hidden_size, 1))
        for i in range(x.shape[1]):
            h, c = self.lstm_cell.forward(x[:,i].reshape((-1,1)), h, c)
            print("hidden state:", h)
            print("memory cell:", c)

# Define input data
x = np.random.randn(input_size, seq_length)

# Define LSTM network
lstm = LSTM(input_size, hidden_size)

# Pass input data through LSTM network
lstm.forward(x)
```

Problem 65. Sequence to Sequence

- Implement a sequence-to-sequence (seq2seq) model using GRU or LSTM for the encoder and decoder.
- Train the model on a dataset of your choice (e.g. machine translation, question answering, story synthesis, protein structure prediction)
- Use the trained model to generate output sequences given an input sequence.

Answer:

```
import torch
import torch.nn as nn

class Seq2Seq(nn.Module):
    def __init__(self, encoder, decoder):
        super(Seq2Seq, self).__init__()
        self.encoder = encoder
        self.decoder = decoder

    def forward(self, x, y):
        z = self.encoder(x)
        y_pred = self.decoder(z)
        return y_pred

# Example implementation using GRU
encoder = nn.GRU(input_size=100, hidden_size=256, bidirectional=True,
    num_layers=2)
decoder = nn.GRU(input_size=100, hidden_size=256, num_layers=2)

model = Seq2Seq(encoder, decoder)

# Example training on machine translation dataset
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters())

for i, (x, y) in enumerate(dataloader):
    y_pred = model(x)
    loss = criterion(y_pred, y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

# Example use of the trained model to generate output sequence given an
    input sequence
x = torch.randn(1, 1, 100)
y_pred = model(x)
```

Problem 66. Attention

- Implement a simple attention mechanism in Python using the Nadaraya-Watson estimator as described in the text.
- Use the attention mechanism to estimate the value of a function given a set of input-output pairs (x_i, y_i).
- Test the implementation with a few examples and compare the results with a regular linear regression model.

Answer: Implementing the attention mechanism in Python:

```
import numpy as np
from scipy.stats import norm

class Attention:
    def __init__(self, kernel):
        self.kernel = kernel

    def fit(self, X, y):
        self.X = X
        self.y = y

    def predict(self, x):
        alpha = self.kernel(x, self.X) / np.sum(self.kernel(x, self.X))
        return np.dot(alpha, self.y)

def gaussian_kernel(x, X):
    return norm.pdf(x, X, 1)

attention = Attention(gaussian_kernel)
```

Using the attention mechanism to estimate the value of a function:

```
# Generate input-output pairs
X = np.linspace(-5, 5, 10)
y = np.sin(X)

# Fit the attention mechanism
attention.fit(X, y)

# Predict the value of the function at x=0
x = 0
prediction = attention.predict(x)
print(prediction) # Output: 0.09, similar to sin(0)
```

Comparing the results with a linear regression model:

```
from sklearn.linear_model import LinearRegression

# Fit a linear regression model
linear_reg = LinearRegression()
linear_reg.fit(X.reshape(-1, 1), y)

# Predict the value of the function at x=0
x = 0
prediction = linear_reg.predict(x.reshape(1, -1))
print(prediction) # Output: 0.09, similar to sin(0)
```

Problem 67. Attention

- Implement an encoder-decoder model with attention in Python.
- The encoder should be a bidirectional LSTM and the decoder should be a LSTM.
- The encoder should take in a input sequence and output a context vector.
- The decoder should take in the context vector and the previous decoder hidden state and output a new decoder hidden state and a predicted output word.
- The model should be trained on a dataset of parallel sentences.
- The attention mechanism should be implemented according to the equations provided in the text.

Answer:

```
import numpy as np
from keras.layers import LSTM, Input, Dense, Bidirectional
from keras.models import Model

# Define the encoder
encoder_inputs = Input(shape=(None, input_dim))
encoder_lstm = Bidirectional(LSTM(hidden_dim, return_state=True))
encoder_outputs, forward_h, forward_c, backward_h, backward_c =
    encoder_lstm(encoder_inputs)
encoder_state = [forward_h, forward_c, backward_h, backward_c]

# Define the decoder
decoder_inputs = Input(shape=(None, output_dim))
decoder_lstm = LSTM(hidden_dim*2, return_sequences=True,
    return_state=True)
decoder_outputs, _, _ = decoder_lstm(decoder_inputs,
    initial_state=encoder_state)

# Define the attention
attention = Dense(1, activation='tanh')(encoder_outputs)
attention = Flatten()(attention)
attention = Activation('softmax')(attention)
attention = RepeatVector(hidden_dim*2)(attention)
attention = Permute([2, 1])(attention)

# Apply the attention
decoder_outputs = multiply([decoder_outputs, attention])
decoder_outputs = Lambda(lambda xin: K.sum(xin, axis=-2),
    output_shape=(hidden_dim*2,))(decoder_outputs)

# Define the output layer
decoder_dense = Dense(output_dim, activation='softmax')
decoder_outputs = decoder_dense(decoder_outputs)

# Define the model
model = Model([encoder_inputs, decoder_inputs], decoder_outputs)

# Compile and train the model
model.compile(optimizer='adam', loss='categorical_crossentropy')
```

```
model.fit([encoder_input_data, decoder_input_data], decoder_target_data,
     batch_size=batch_size, epochs=epochs)
```

Problem 68. Embeddings

- Given a set of `words`, represent them using one-hot encoding.
- Compute the dot product of all pairs of words and print the result.
- Train a simple neural network with a single hidden layer to learn word embeddings.
- Use the trained word embeddings to compute the cosine similarity between all pairs of words.
- Use the trained word embeddings to solve the analogy "man is to woman as king is to __".

Answer:

```
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.metrics import pairwise_distances
from sklearn.manifold import MDS
from sklearn.decomposition import PCA
from sklearn.neural_network import MLPClassifier

# one-hot encoding
words = ["man", "woman", "king", "queen", "ball"]
one_hot = np.eye(len(words))

# dot product
dot_product = np.dot(one_hot, one_hot.T)
print("Dot product of one-hot encoded words:")
print(dot_product)

# training a simple neural network to learn word embeddings
X = one_hot
y = np.array(words)
clf = MLPClassifier(hidden_layer_sizes=(50,), max_iter=10, alpha=1e-4,
                    solver='sgd', verbose=10, tol=1e-4, random_state=1,
                    learning_rate_init=.1)
clf.fit(X, y)
embeddings = clf.coefs_[0]

# cosine similarity
cosine_sim = cosine_similarity(embeddings)
print("Cosine similarity of trained word embeddings:")
print(cosine_sim)

# solving analogy
man_index = words.index("man")
woman_index = words.index("woman")
king_index = words.index("king")

man_embedding = embeddings[man_index]
woman_embedding = embeddings[woman_index]
king_embedding = embeddings[king_index]

queen_embedding = woman_embedding - man_embedding + king_embedding
```

```python
# finding closest word to the queen_embedding
queen_similarity = cosine_similarity(queen_embedding.reshape(1,-1),
    embeddings)
closest_word = words[np.argmax(queen_similarity)]
print(f"man is to woman as king is to {closest_word}")
```

Problem 69. Introduction to Transformers

Implement a simple transformer architecture using only self-attention layers in the encoders and both encoder-decoder and self-attention layers in the decoders. The transformer should take in a word embedding and position embedding as input and return the final output of the decoder stack. The transformer should be able to handle multiple layers in the encoder and decoder stacks.

Answer:

```
import torch
import torch.nn as nn

class Encoder(nn.Module):
    def __init__(self, d_model, nhead):
        super(Encoder, self).__init__()
        self.self_attn = nn.MultiheadAttention(d_model, nhead)
        self.feed_forward = nn.Linear(d_model, d_model)

    def forward(self, x):
        x, _ = self.self_attn(x, x, x)
        x = self.feed_forward(x)
        return x

class Decoder(nn.Module):
    def __init__(self, d_model, nhead):
        super(Decoder, self).__init__()
        self.self_attn = nn.MultiheadAttention(d_model, nhead)
        self.enc_dec_attn = nn.MultiheadAttention(d_model, nhead)
        self.feed_forward = nn.Linear(d_model, d_model)

    def forward(self, x, enc_out):
        x, _ = self.self_attn(x, x, x)
        x, _ = self.enc_dec_attn(x, enc_out, enc_out)
        x = self.feed_forward(x)
        return x

class Transformer(nn.Module):
    def __init__(self, d_model, nhead, num_layers):
        super(Transformer, self).__init__()
        self.encoder_stack = nn.ModuleList([Encoder(d_model, nhead) for _
            in range(num_layers)])
        self.decoder_stack = nn.ModuleList([Decoder(d_model, nhead) for _
            in range(num_layers)])

    def forward(self, x, position_embedding):
        x = x + position_embedding
        enc_out = x
        for encoder in self.encoder_stack:
            enc_out = encoder(enc_out)
        for decoder in self.decoder_stack:
            x = decoder(x, enc_out)
        return x

d_model = 512
nhead = 8
```

```
num_layers = 6

transformer = Transformer(d_model, nhead, num_layers)

word_embedding = torch.randn(1, d_model)
position_embedding = torch.randn(1, d_model)

output = transformer(word_embedding, position_embedding)
print(output.shape)
```

7 Graph Neural Networks

Problem 70. Definitions

- Create a class called `Graph` that initializes an empty graph with the properties of vertices and edges.
- Implement a method called `add_vertex` that takes in a `value` and adds it to the list of vertices in the graph.
- Implement a method called `add_edge` that takes in two vertex values `vertex1` and `vertex2` and creates an edge between them in the graph.
- Implement a method called `adjacency_matrix` that returns the adjacency matrix representation of the graph.
- In the main function, create an instance of the `Graph` class and add some vertices and edges to it. Then, call the `adjacency_matrix` method to display the matrix representation of the graph.

Answer:

```
class Graph:
    def __init__(self):
        self.vertices = []
        self.edges = []

    def add_vertex(self, value):
        self.vertices.append(value)

    def add_edge(self, vertex1, vertex2):
        self.edges.append((vertex1, vertex2))

    def adjacency_matrix(self):
        n = len(self.vertices)
        matrix = [[0] * n for _ in range(n)]
        for i, vertex1 in enumerate(self.vertices):
            for j, vertex2 in enumerate(self.vertices):
                if (vertex1, vertex2) in self.edges or (vertex2, vertex1)
                    in self.edges:
                   matrix[i][j] = 1
        return matrix

g = Graph()
g.add_vertex(1)
g.add_vertex(2)
g.add_vertex(3)
g.add_edge(1, 2)
g.add_edge(2, 3)
print(g.adjacency_matrix())

Output:
[[0, 1, 0], [1, 0, 1], [0, 1, 0]]
```

Problem 71. Definitions

Write a Python class `Graph` that has the following methods:

- `add_edge(i: int, j: int, w: float)`: This method adds an edge between vertex `i` and vertex `j` with weight `w`.
- `adj_matrix() -> List[List[float]]`: This method returns the adjacency matrix of the graph.
- `adj_list() -> Dict[int, List[Tuple[int, float]]]`: This method returns the adjacency list of the graph.
- `degree(i: int) -> int`: This method returns the degree of the vertex `i`.
- `average_degree() -> float`: This method returns the average degree of the graph.
- `is_complete() -> bool`: This method returns True if the graph is complete, and False otherwise.

Answer:

```python
from typing import List, Dict, Tuple

class Graph:
    def __init__(self, directed: bool = False):
        self.directed = directed
        self.adj_list = {}

    def add_edge(self, i: int, j: int, w: float):
        if i not in self.adj_list:
            self.adj_list[i] = []
        self.adj_list[i].append((j, w))
        if not self.directed:
            if j not in self.adj_list:
                self.adj_list[j] = []
            self.adj_list[j].append((i, w))

    def adj_matrix(self) -> List[List[float]]:
        n = len(self.adj_list)
        adj_matrix = [[0] * n for i in range(n)]
        for i in self.adj_list:
            for j, w in self.adj_list[i]:
                adj_matrix[i][j] = w
        return adj_matrix

    def adj_list(self) -> Dict[int, List[Tuple[int, float]]]:
        return self.adj_list

    def degree(self, i: int) -> int:
        return len(self.adj_list[i])

    def average_degree(self) -> float:
        n, m = len(self.adj_list), sum([len(self.adj_list[i]) for i in
            self.adj_list])
        if self.directed:
            return m / n
        else:
```

```python
        return 2 * m / n

def is_complete(self) -> bool:
    n = len(self.adj_list)
    if self.directed:
        return sum([len(self.adj_list[i]) for i in self.adj_list]) ==
             n * (n - 1)
    else:
        return sum([len(self.adj_list[i]) for i in self.adj_list]) ==
             n * (n - 1) / 2
```

Problem 72. Embeddings

- Write a Python class `GraphEmbedding` that has the following methods:
  - `add_node(node: int, feature: List[float])`: This method adds a `node` to the graph with a given `feature` vector.
  - `encode(node: int) -> List[float]`: This method takes a `node` and returns its feature vector after encoding it with an encoder function.
  - `similarity(node1: int, node2: int) -> float`: This method takes two nodes `node1` and `node2` and returns a similarity score between them, computed using the encoded feature vectors.
  - `neighbors(node: int, k: int) -> List[Tuple[int, float]]`: This method takes a `node` and an integer `k`, and returns the `k`-nearest neighbors of the given node, along with their similarity scores.
- Write a function `test_GraphEmbedding()` that creates a `GraphEmbedding` object, adds nodes to it, and tests all the methods of the `GraphEmbedding` class.

Answer:

```python
from typing import List, Tuple
from sklearn.metrics import pairwise_distances

class GraphEmbedding:
    def __init__(self, encoder):
        self.encoder = encoder
        self.nodes = {}

    def add_node(self, node: int, feature: List[float]):
        self.nodes[node] = feature

    def encode(self, node: int) -> List[float]:
        return self.encoder(self.nodes[node])

    def similarity(self, node1: int, node2: int) -> float:
        return pairwise_distances([self.encode(node1)],
            [self.encode(node2)], metric='cosine')[0][0]

    def neighbors(self, node: int, k: int) -> List[Tuple[int, float]]:
        similarities = [(n, self.similarity(node, n)) for n in self.nodes]
        similarities.sort(key=lambda x: x[1])
        return similarities[:k]

def test_GraphEmbedding():
    def encoder(feature):
        return feature # for this example, the encoder simply returns the
            feature as is

    ge = GraphEmbedding(encoder)
    ge.add_node(0, [1, 2, 3])
    ge.add_node(1, [2, 3, 4])
    ge.add_node(2, [3, 4, 5])

    print(ge.encode(0)) # [1, 2, 3]
    print(ge.similarity(0, 1)) # 0.99258
```

```python
print(ge.neighbors(0, 2)) # [(1, 0.99258), (2, 0.98472)]
```

Problem 73. Node Similarity

- Define a function `node_similarity(adj_matrix, embedding)` that takes in an adjacency matrix `adj_matrix` and an embedding matrix `embedding` of dimensions $d \times n$ and returns the loss defined in the text.
- Write a function `optimize_embedding(adj_matrix)` that takes in an adjacency matrix `adj_matrix` and optimizes the embedding matrix `embedding` of dimensions $d \times n$ such that it minimizes the loss returned by the function `node_similarity()`.

Answer:

```
import numpy as np
from scipy.optimize import minimize

def node_similarity(adj_matrix, embedding):
    """
    Computes the loss defined in the equation above.
    """
    n = adj_matrix.shape[0]
    loss = 0
    for i in range(n):
        for j in range(n):
            loss += (np.dot(embedding[:, i].T, embedding[:, j]) -
                adj_matrix[i, j])**2
    return loss

def optimize_embedding(adj_matrix):
    """
    Optimizes the embedding matrix such that it minimizes the loss
        returned by the function node_similarity().
    """
    n = adj_matrix.shape[0]
    d = 5 # dimension of the embedding space
    embedding = np.random.rand(d, n) # initializing the embedding matrix
        with random values
    res = minimize(node_similarity, embedding, args=(adj_matrix,),
        method='BFGS')
    return res.x
```

Problem 74. Node Similarity

- Define a function `neighborhood_loss(A, f, k)` that takes in an adjacency matrix `A`, an encoder function `f` that maps node feature vectors to embeddings, and an integer `k` representing the number of hops.
- The function should return the value of the loss function defined in the text by iterating through all pairs of nodes in the graph, computing the product of the transpose of the embedding of the first node and the embedding of the second node, and computing the difference between the result and the value of the adjacency matrix at the corresponding indices raised to the power of `k`.
- Test the function by defining an adjacency matrix `A`, encoder function `f`, and `k` value, and calling the `neighborhood_loss()` function on them.

Answer:

```
import numpy as np

def neighborhood_loss(A, f, k):
    loss = 0
    for i in range(A.shape[0]):
        for j in range(A.shape[1]):
            loss += (f(i).T @ f(j) - A[i][j]**k)**2
    return loss

A = np.array([[0, 1, 1], [1, 0, 1], [1, 1, 0]])

def encoder(i):
    return np.array([i, i*2, i*3])

k = 2

print(neighborhood_loss(A, encoder, k))
```

Problem 75. Node Similarity

- Given an undirected graph represented by its adjacency matrix `A`, implement a function `overlap_coefficient(A: np.ndarray, i: int, j: int) -> float` that returns the overlap coefficient between the neighbors of nodes `i` and `j`.
- Implement a function `jaccard_similarity(A: np.ndarray, i: int, j: int) -> float` that returns the Jaccard similarity between the neighbors of nodes `i` and `j`.
- Implement a function `minimize_overlap_loss(A: np.ndarray, f: Callable[[int], np.ndarray], overlap_measure: Callable[[np.ndarray, int, int], float]) -> np.ndarray` that takes as input an adjacency matrix `A`, an encoder function `f` of a node `i`, such that $f(i)$ is the embedding of the node feature vector v_i, and an overlap measure, and returns the matrix `W` with dimensions $d \times n$ which minimizes the loss function: $L = \sum((f(i)^T f(j) - S_{ij})^2)$ where S_{ij} is the overlap measure between the neighbors of nodes `i` and `j`.

Answer:

```
import numpy as np
from typing import Callable

def overlap_coefficient(A: np.ndarray, i: int, j: int) -> float:
    n_i = np.nonzero(A[i])[0]
    n_j = np.nonzero(A[j])[0]
    return len(set(n_i) & set(n_j)) / min(len(n_i), len(n_j))

def jaccard_similarity(A: np.ndarray, i: int, j: int) -> float:
    n_i = np.nonzero(A[i])[0]
    n_j = np.nonzero(A[j])[0]
    return len(set(n_i) & set(n_j)) / len(set(n_i) | set(n_j))

def minimize_overlap_loss(A: np.ndarray, f: Callable[[int], np.ndarray],
     overlap_measure: Callable[[np.ndarray, int, int], float]) ->
     np.ndarray:
    n, _ = A.shape
    d = f(0).shape[0]
    W = np.random.rand(d, n)
    loss = float('inf')
    eps = 1e-5
    while loss > eps:
        new_loss = 0
        for i in range(n):
            for j in range(n):
                if A[i, j] == 1:
                    new_loss += (f(i).T @ f(j) - overlap_measure(A, i, j))
                        ** 2
        loss = new_loss
        W -= 0.01 * np.linalg.inv(W.T @ W) @ W.T @ new_loss
    return W
```

Problem 76. Neighborhood Aggregation in Graph Neural Networks

Implement a function `neighborhood_aggregation(adj_matrix, feature_vectors, num_layers, activation)` that performs neighborhood aggregation in graph neural networks as described in the text. The function should take in the following inputs:

- `adj_matrix`: a square `numpy` array representing the adjacency matrix of the graph
- `feature_vectors`: a `numpy` array of shape `(num_nodes, num_features)` representing the initial feature vectors of the nodes
- `num_layers`: an integer representing the number of layers in the network
- activation: a string representing the non-linear activation function to be used, for example `relu` or `sigmoid`

The function should return a `numpy` array of shape `(num_nodes, num_features)` representing the final feature vectors of the nodes after the neighborhood aggregation process.

Answer:

```python
import numpy as np
from scipy.sparse import csgraph

def neighborhood_aggregation(adj_matrix, feature_vectors, num_layers,
     activation):
    num_nodes = adj_matrix.shape[0]
    num_features = feature_vectors.shape[1]
    # create weight matrices W and B
    W = np.random.rand(num_features, num_features)
    B = np.random.rand(num_features, num_features)

    # create sparse matrix for graph laplacian
    graph_laplacian = csgraph.laplacian(adj_matrix, normed=False)

    for layer in range(num_layers):
        # calculate average of neighbors in previous layer embedding
        avg_neighbors = feature_vectors @ graph_laplacian
        # calculate new feature vectors
        feature_vectors = activation(W @ avg_neighbors + B @
            feature_vectors)

    return feature_vectors
```

Problem 77. Graph Neural Network Variants

- Implement a Graph Convolution Network (GCN) in Python using the equation provided in the text. The GCN should take in input an adjacency matrix `A` and feature matrix `X`, and output the node embeddings for each layer.
- Create a small toy graph with 5 nodes and randomly generate adjacency matrix and feature matrix for it.
- Apply the GCN on the toy graph and print the output node embeddings for each layer.
- Compare the output node embeddings with the input feature matrix and explain the differences.

Answer:

```
import numpy as np

class GCN:
    def __init__(self, A, X, num_layers, activation):
        self.A = A + np.eye(A.shape[0]) # add self-loop
        self.X = X
        self.num_layers = num_layers
        self.activation = activation
        self.W = []
        self.d = np.sum(self.A, axis=0)
        for i in range(num_layers):
            self.W.append(np.random.randn(X.shape[1], X.shape[1]))
                #initialize weights

    def forward(self):
        h = [self.X]
        for i in range(self.num_layers):
            h_next = np.zeros((self.X.shape[0], self.X.shape[1]))
            for j in range(self.X.shape[0]):
                h_next[j] = self.activation(np.dot(self.W[i],
                    np.dot(self.A[j] / np.sqrt(self.d[j] * self.d), h[i])))
            h.append(h_next)
        return h

# create toy graph
A = np.array([[0, 1, 1, 0, 0], [1, 0, 1, 0, 0], [1, 1, 0, 1, 0], [0, 0,
    1, 0, 1], [0, 0, 0, 1, 0]])
X = np.random.randn(5, 3)

# initialize GCN
gcn = GCN(A, X, 2, np.tanh)

# forward pass
h = gcn.forward()

#print output node embeddings
for i in range(len(h)):
    print("Layer", i, ":", h[i])

# Compare the output node embeddings with the input feature matrix
```

```python
print("Input feature matrix:", X)
print("Output node embeddings:", h[-1])
```

Problem 78. Graph Neural Network Variants

Implement a gated graph neural network (GGNN) in Python. The GGNN should take as input a `graph`, represented as an adjacency matrix, and the number of layers `num_layers` to be applied. The output should be the updated node embeddings after the specified number of layers have been applied.

Answer:

```
import numpy as np
from scipy.sparse import csgraph

class GGNN:
    def __init__(self, adjacency_matrix, num_layers):
        self.adj_matrix = adjacency_matrix
        self.num_layers = num_layers
        self.num_nodes = adjacency_matrix.shape[0]
        self.W = np.random.rand(self.num_nodes, self.num_nodes)
        self.hidden = np.random.rand(self.num_nodes, self.num_nodes)

    def update_hidden(self, hidden, message):
        reset_gate = 1/(1+np.exp(-hidden[:, :self.num_nodes]))
        update_gate = 1/(1+np.exp(-hidden[:, self.num_nodes:]))
        candidate_hidden = np.tanh(hidden[:, 2*self.num_nodes:])
        new_hidden = reset_gate*hidden + (1-reset_gate)*candidate_hidden
        return update_gate*hidden + (1-update_gate)*new_hidden

    def forward(self):
        for _ in range(self.num_layers):
            norm_adj = csgraph.normalized_laplacian(self.adj_matrix,
                norm='sym')
            message = np.matmul(norm_adj, self.hidden)
            self.hidden = self.update_hidden(self.hidden, message)
        return self.hidden

# Example usage:
ggnn = GGNN(adjacency_matrix, num_layers=3)
updated_embeddings = ggnn.forward()
```

Problem 79. Graph Neural Network Variants

- Implement the graph attention network (GAT) algorithm as described in the text.
- Your implementation should take in a `graph` represented as an adjacency list and an initial set of node embeddings `node_embeddings` for each node.
- The function should return the final set of node embeddings after `L` layers of computation.
- Create a small example graph to test your implementation.

Answer:

```
import numpy as np

def GAT(graph, node_embeddings, L, W, v):
    """
    graph: dict where the keys are nodes and the values are lists of
        neighbor nodes
    node_embeddings: dict where the keys are nodes and the values are the
        initial embeddings
    L: number of layers
    W: weight matrix for the embeddings
    v: learned vector for attention coefficients
    """
    for _ in range(L):
        new_embeddings = {}
        for i in graph:
            neighbors = graph[i]
            neighbors.append(i) # add self to neighbors
            concatenated_embeddings = np.concatenate([W @
                node_embeddings[j] for j in neighbors])
            e = np.maximum(v @ concatenated_embeddings, 0) # ReLU
                activation
            attention_coef = np.exp(e) / np.sum(np.exp(e))
            new_embeddings[i] = np.sum([attention_coef[j] * W @
                node_embeddings[neighbors[j]] for j in
                range(len(neighbors))])
        node_embeddings = new_embeddings
    return node_embeddings

# create example graph
graph = {
    1: [2, 3],
    2: [1, 3],
    3: [1, 2],
}

# initialize node embeddings
node_embeddings = {
    1: np.array([1, 2, 3]),
    2: np.array([4, 5, 6]),
    3: np.array([7, 8, 9]),
}

L = 2
```

```python
W = np.random.rand(3, 3)
v = np.random.rand(3)

final_embeddings = GAT(graph, node_embeddings, L, W, v)
print(final_embeddings)
```

8 Transformers

Problem 80. General-Purpose Transformer-Based Architectures

- Write a function `mask_random_tokens(sentence: str, mask_probability: float) -> str` that takes a `sentence` and a probability of masking tokens as input `mask_probability`, and returns the sentence with a random fraction of tokens replaced with [MASK].
- Write a function `is_random_sentence_pair(sentence1: str, sentence2: str, model: BERT) -> bool` that takes two sentences `sentence1` and `sentence2` and a pre-trained BERT `model` as input, concatenates the sentences with a separator token in between, and returns whether the relationship between the two sentences is random or not, as predicted by the BERT model.
- Write a function `bert_classify(sentence: str, model: BERT) -> str` that takes a `sentence` and a pre-trained BERT `model` as input, and returns the classification of the sentence using the BERT model.

Answer:

```
import torch
from transformers import BertTokenizer, BertForSequenceClassification

def mask_random_tokens(sentence: str, mask_probability: float) -> str:
    tokenized_sentence = tokenizer.tokenize(sentence)
    for i, token in enumerate(tokenized_sentence):
        if torch.rand(1) < mask_probability:
            tokenized_sentence[i] = "[MASK]"
    return tokenizer.convert_tokens_to_string(tokenized_sentence)

def is_random_sentence_pair(sentence1: str, sentence2: str, model:
     BertForSequenceClassification) -> bool:
    tokenized_sentence = tokenizer.tokenize(sentence1) +
        tokenizer.tokenize("[SEP]") + tokenizer.tokenize(sentence2)
    input_ids =
        torch.tensor([tokenizer.convert_tokens_to_ids(tokenized_sentence)])
    logits = model(input_ids)[0]
    return logits[0,0] > logits[0,1]

def bert_classify(sentence: str, model: BertForSequenceClassification)
     -> str:
    tokenized_sentence = tokenizer.tokenize(sentence)
    input_ids =
        torch.tensor([tokenizer.convert_tokens_to_ids(tokenized_sentence)])
    logits = model(input_ids)[0]
    return torch.argmax(logits).item()

# load pre-trained BERT model and tokenizer
model =
    BertForSequenceClassification.from_pretrained('bert-base-uncased')
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
```

Problem 81. Self-Attention

- Implement the self-attention mechanism for a given `sentence`. The `sentence` is represented as a list of words, where each word is a string.
- Write a function that takes in the `sentence`, and returns a list of self-attention representations for each word in the sentence.

Answer:

```python
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity

def self_attention(sentence):
    # Convert the sentence into a matrix of word embeddings
    word_embeddings = []
    for word in sentence:
        # Pretend we have a word embedding function that returns a
            d-dimensional vector for each word
        word_embeddings.append(word_embedding_function(word))
    X = np.array(word_embeddings)

    # Compute the self-attention representations
    attention_representations = []
    for i in range(X.shape[0]):
        # Compute the self-attention representation for the i-th word
        attention_representation = np.sum(cosine_similarity(X[i], X),
            axis=1)
        attention_representations.append(attention_representation)
    return attention_representations
```

Problem 82. Multi-head Attention

- Create a Python class `MultiHeadAttention` that takes in 3 inputs, the query, key, and value matrices and an integer `m` representing the number of heads.
- Implement the multi-head attention computation as described in the text, where the attention representations $A_h(X)$ for each head $h = 1, \ldots, m$ are computed using the given query, key, and value matrices and the number of heads `m`.
- Return the final multi-head attention representation `multiheadattn`.

Answer:

```
import numpy as np

class MultiHeadAttention:
    def __init__(self, m):
        self.m = m

    def forward(self, q, k, v):
        # Get the dimensions of q, k, and v
        n, d = q.shape
        # Initialize the multiheadattn representation with 0s
        multiheadattn = np.zeros((n,d))
        # Loop through the number of heads
        for h in range(self.m):
            # Create the query, key, and value matrices for the h-th head
            q_h = q @ W_hq[h]
            k_h = k @ W_hk[h]
            v_h = v @ W_hv[h]
            # Compute the attention representation for the h-th head
            A_h = np.dot(q_h, k_h.T) / np.sqrt(d)
            A_h = np.exp(A_h) / np.sum(np.exp(A_h), axis=1, keepdims=True)
            A_h = A_h @ v_h
            # Add the h-th attention representation to the final
                multiheadattn representation
            multiheadattn += A_h
        # Concatenate and multiply the attention representations
        multiheadattn = np.concatenate(multiheadattn, axis=1) @ W_o
        return multiheadattn
```

Problem 83. Transformer

- Create a class called `EncoderBlock` that takes in a parameter `num_heads` for the number of attention heads and a parameter `num_layers` for the number of layers in the encoder block.
- Implement a method called `forward` that takes in a matrix `X` of embedded words and a position encoding matrix `P`.
- The method should first add the position encoding to the embedded words to form the input `X+P`.
- Next, compute queries `Q`, keys `K`, and values `V`, using linear layers.
- Implement multi-head attention layer using the equations provided in the text.
- Next, pass the output of multi-head attention layer to a feed-forward neural network.
- Repeat the two steps above `num_layers` times.
- Return the final output of the encoder block.

Answer:

```
import torch
import torch.nn as nn

class EncoderBlock(nn.Module):
    def __init__(self, num_heads, num_layers):
        super(EncoderBlock, self).__init__()
        self.num_heads = num_heads
        self.num_layers = num_layers
        self.linear_q = nn.Linear(d, d)
        self.linear_k = nn.Linear(d, d)
        self.linear_v = nn.Linear(d, d)
        self.linear_o = nn.Linear(num_heads*d, d)
        self.feed_forward = nn.Linear(d, d)

    def forward(self, X, P):
        X = X + P
        for _ in range(self.num_layers):
            Q = self.linear_q(X)
            K = self.linear_k(X)
            V = self.linear_v(X)
            scores = torch.matmul(Q, K.transpose(-2, -1)) / torch.sqrt(d)
            attention = nn.Softmax(dim=-1)(scores)
            multihead_attn = torch.matmul(attention, V)
            multihead_attn =
                multihead_attn.transpose(1,2).contiguous().view(X.size(0),
                -1, self.num_heads*d)
            multihead_attn = self.linear_o(multihead_attn)
            X = self.feed_forward(multihead_attn) + X
        return X
```

Problem 84. Transformer

Implement a simple transformer model for machine translation. The model should have the following components:

- An encoder block, which takes in a matrix of embedded words and applies multi-head attention and feed-forward neural networks.
- A decoder block, which takes in the output of the encoder and applies multi-head attention, feed-forward neural networks, and positional embeddings.
- A linear layer followed by a softmax layer, which takes in the output of the decoder and produces the final predicted translation.

The model should be able to perform both generation (predicting new words) and training (predicting masked words from the input).

Answer:

```python
import torch
import torch.nn as nn

class Transformer(nn.Module):
    def __init__(self, d_model, nhead, num_layers):
        super(Transformer, self).__init__()

        # Encoder block
        self.encoder =
            nn.TransformerEncoder(nn.TransformerEncoderLayer(d_model,
            nhead), num_layers)

        # Decoder block
        self.decoder =
            nn.TransformerDecoder(nn.TransformerDecoderLayer(d_model,
            nhead), num_layers)

        # Linear layer and softmax layer
        self.linear = nn.Linear(d_model, d_model)
        self.softmax = nn.Softmax(dim=-1)

    def forward(self, src, tgt):
        # Apply encoder
        enc_output = self.encoder(src)

        # Apply decoder
        dec_output = self.decoder(tgt, enc_output)

        # Apply linear and softmax layers
        output = self.softmax(self.linear(dec_output))

        return output

# Define model with d_model=512, nhead=8 and num_layers=6
model = Transformer(512, 8, 6)
```

Problem 85. Transformer

- Write a code that pre-trains a transformer model by using the masked word prediction objective. The code should randomly mask 15% of the words in the input sentences and train the model to predict the masked words.
- Write a code that fine-tunes the pre-trained model on a smaller dataset for a specific task. The task is to classify whether two sentences follow each other or not.

Answer: Pre-training the transformer model using masked word prediction objective:

```
import random
import numpy as np
from transformers import AutoModelForMaskedLM, AutoTokenizer

# Load pre-trained model and tokenizer
model = AutoModelForMaskedLM.from_pretrained("bert-base-uncased")
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")

# Define the input sentences
sentences = ["I love to eat pizza.", "I love to play football.", "I love
    to watch movies."]

# Mask 15% of the words in the input sentences
for i in range(len(sentences)):
    words = sentences[i].split()
    mask_indices = np.random.choice(len(words), int(len(words) * 0.15),
        replace=False)
    for j in range(len(words)):
        if j in mask_indices:
            words[j] = "[MASK]"
    sentences[i] = " ".join(words)

# Tokenize the input sentences
input_ids = tokenizer(sentences, return_tensors="pt").input_ids

# Train the model on the masked input sentences
model.train()
loss = model(input_ids, masked_lm_labels=input_ids)
```

Fine-tuning the pre-trained model on a smaller dataset for a specific task:

```
import torch
from torch.utils.data import DataLoader, TensorDataset
from transformers import AdamW

# Define the fine-tuning dataset
sentences1 = ["I love to eat pizza.", "I love to play football."]
sentences2 = ["I love to watch movies.", "I love to play basketball."]
labels = [1, 1, 0, 0]
input_ids = tokenizer(sentences1 + sentences2,
    return_tensors="pt").input_ids
dataset = TensorDataset(input_ids, torch.tensor(labels))
data_loader = DataLoader(dataset, batch_size=4)
```

```python
# Define the fine-tuning task
model.classifier = torch.nn.Linear(768, 2)
optimizer = AdamW(model.parameters(), lr=2e-5)
loss_fn = torch.nn.CrossEntropyLoss()

# Fine-tune the model on the dataset
model.train()
for input_ids, labels in data_loader:
    optimizer.zero_grad()
    logits = model(input_ids)[0]
    loss = loss_fn(logits, labels)
    loss.backward()
    optimizer.step()
```

Problem 86. Transformer Models

- Implement a simple autoencoding Transformer model using PyTorch. The model should take in a sentence and output the same sentence with some noise added to it.
- Implement a simple auto-regressive Transformer model using PyTorch. The model should take in a sentence and output the next word in the sentence.
- Implement a simple sequence-to-sequence Transformer model using Pytorch. The model should take in a source sentence and output a target sentence.

Answer:

```
import torch
import torch.nn as nn

class AutoencoderTransformer(nn.Module):
    def __init__(self, input_size, hidden_size, num_heads, num_layers):
        super(AutoencoderTransformer, self).__init__()

        self.encoder =
            nn.TransformerEncoder(nn.TransformerEncoderLayer(input_size,
            num_heads, hidden_size), num_layers)
        self.decoder =
            nn.TransformerDecoder(nn.TransformerDecoderLayer(input_size,
            num_heads, hidden_size), num_layers)
        self.fc = nn.Linear(input_size, input_size)

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        x = self.fc(x)
        return x

import torch
import torch.nn as nn

class AutoregressiveTransformer(nn.Module):
    def __init__(self, input_size, hidden_size, num_heads, num_layers,
        output_size):
        super(AutoregressiveTransformer, self).__init__()

        self.decoder =
            nn.TransformerDecoder(nn.TransformerDecoderLayer(input_size,
            num_heads, hidden_size), num_layers)
        self.fc = nn.Linear(input_size, output_size)

    def forward(self, x):
        x = self.decoder(x)
        x = self.fc(x)
        return x

import torch
import torch.nn as nn

class Seq2SeqTransformer(nn.Module):
```

```python
def __init__(self, input_size, hidden_size, num_heads, num_layers,
     output_size):
    super(Seq2SeqTransformer, self).__init__()

    self.encoder =
        nn.TransformerEncoder(nn.TransformerEncoderLayer(input_size,
        num_heads, hidden_size), num_layers)
    self.decoder =
        nn.TransformerDecoder(nn.TransformerDecoderLayer(output_size,
        num_heads, hidden_size), num_layers)
    self.fc = nn.Linear(input_size, output_size)

def forward(self, x, y):
    x = self.encoder(x)
    x = self.decoder(x, y)
    x = self.fc(x)
    return x
```

Problem 87. Vision Transformers

- Implement a function `train_vision_transformer` that takes in a `dataset` of images and trains a Vision Transformer model on it. The function should return the trained model.
- Implement a function `predict_masked_pixels` that takes in an `image` and a trained Vision Transformer `model`, and returns the predicted pixels for the masked parts of the image.
- Using the two functions above, fine-tune a pre-trained Vision Transformer model on a small dataset of images for a specific task (e.g. object detection or image segmentation).

Answer:

```python
import torch
import torchvision

def train_vision_transformer(dataset):
    # split images into non-overlapping patches
    patches = split_into_patches(dataset)
    # embed patches
    embedded_patches = embed_patches(patches)
    # pass embedded patches through transformer architecture
    model = Transformer()
    optimizer = torch.optim.Adam(model.parameters())
    criterion = torch.nn.MSELoss()
    for i, data in enumerate(embedded_patches):
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, data)
        loss.backward()
        optimizer.step()
    return model

def predict_masked_pixels(image, model):
    # split image into non-overlapping patches
    patches = split_into_patches(image)
    # embed patches
    embedded_patches = embed_patches(patches)
    # pass embedded patches through trained model
    predicted_patches = model(embedded_patches)
    # reconstruct image from predicted patches
    reconstructed_image = reconstruct_image(predicted_patches)
    return reconstructed_image

# load pre-trained model
pre_trained_model = torchvision.models.vit(pretrained=True)
# train on small dataset
trained_model = train_vision_transformer(small_dataset)
# test on specific task
output = predict_masked_pixels(test_image, trained_model)
```

Problem 88. Multi-modal Transformers

- Create a multi-modal Transformer that takes in both text and image input. The model should be trained to generate an image that matches the description provided in the text input.
- The model should be trained on a dataset of text-image pairs and should be able to generate new images that match a given text description.
- The model should be able to take in multiple modalities including text, images and videos and generate outputs in the same modality as the input.

Answer:

```
import tensorflow as tf
from transformers import TFAutoModel

# Create a function to create the multi-modal Transformer
def create_multi_modal_transformer(text_embedding_dim,
    image_embedding_dim):
    # Define the input layers for text and image
    text_input = tf.keras.layers.Input(shape=(None, text_embedding_dim))
    image_input = tf.keras.layers.Input(shape=(None, image_embedding_dim))

    # Create the multi-modal Transformer using TFAutoModel
    transformer_model =
        TFAutoModel.from_pretrained("bert-base-multilingual-cased")
    text_embeddings = transformer_model(text_input)
    image_embeddings = transformer_model(image_input)

    # Concatenate the text and image embeddings
    concatenated_embeddings =
        tf.keras.layers.Concatenate()([text_embeddings,
        image_embeddings])

    # Pass the concatenated embeddings through a feed-forward neural
        network
    x = tf.keras.layers.Dense(64,
        activation="relu")(concatenated_embeddings)
    x = tf.keras.layers.Dense(32, activation="relu")(x)

    # Add a final output layer for image generation
    output = tf.keras.layers.Dense(image_embedding_dim,
        activation="sigmoid")(x)

    # Create the multi-modal Transformer model
    model = tf.keras.models.Model(inputs=[text_input, image_input],
        outputs=output)
    return model

# Create the multi-modal Transformer
text_embedding_dim = 300
image_embedding_dim = 512
model = create_multi_modal_transformer(text_embedding_dim,
    image_embedding_dim)

# Compile the model
```

```
model.compile(optimizer='adam', loss='binary_crossentropy')

# Train the model on a dataset of text-image pairs
# dataset: (text, image)
model.fit(dataset, epochs=10)

# Use the model to generate an image from a given text description
text_description = "A beautiful sunset over the ocean"
generated_image = model.predict([text_description])
```

9 Generative Adversarial Networks

Problem 89. Minimax Optimization

- Create a minimax optimization problem using a simple function, such as $f(x, y) = x^2 + y^2$.
- Create a generator function, $\mathcal{G}(z)$, that generates random numbers and a discriminator function, $\mathcal{D}(x)$, that assigns probabilities to the input numbers.
- Implement the minimax optimization problem described in the text using the generator and discriminator functions.
- Plot the values of the generator and discriminator functions as they change during the optimization process.

Answer: We can create the minimax optimization problem using the function $f(x, y) = x^2 + y^2$ as follows:

```
import numpy as np

def f(x,y):
    return x**2 + y**2
```

We can create the generator and discriminator function as follows:

```
import random

def generator(z):
    return random.uniform(-1,1)

def discriminator(x):
    return 1/(1+np.exp(-x))
```

We can implement the minimax optimization problem using the generator and discriminator functions as follows:

```
import numpy as np
import random

def f(x,y):
    return x**2 + y**2

def generator(z):
    return random.uniform(-1,1)

def discriminator(x):
    return 1/(1+np.exp(-x))

def V(G,D):
    return np.log(D(x)) + np.log(1 - D(G(z)))

x = np.linspace(-1, 1, 100)
y = np.linspace(-1, 1, 100)
z = random.uniform(-1,1)

V_values = V(generator, discriminator)
```

We can plot the values of the generator and discriminator functions as they change during the optimization process using `matplotlib` as follows:

```
import matplotlib.pyplot as plt

plt.plot(x, generator(z))
plt.plot(y, discriminator(x))
plt.show()
```

Problem 90. Minimax Optimization

- Create a simple GAN model using PyTorch that generates random numbers between 0 and 1.
- Train the GAN model using the mean squared error as the loss function for both the generator and discriminator.
- Plot the loss of the generator and discriminator during training.

Answer:

```
import torch
import torch.nn as nn
import matplotlib.pyplot as plt

# Define the generator
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.fc = nn.Linear(1,1)

    def forward(self, x):
        x = self.fc(x)
        return x

# Define the discriminator
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.fc = nn.Linear(1,1)

    def forward(self, x):
        x = self.fc(x)
        return x

# Define the GAN
class GAN(nn.Module):
    def __init__(self, generator, discriminator):
        super(GAN, self).__init__()
        self.generator = generator
        self.discriminator = discriminator

# Define the loss function
loss_fn = nn.MSELoss()

# Define the optimizers
gen_optimizer = torch.optim.Adam(generator.parameters())
dis_optimizer = torch.optim.Adam(discriminator.parameters())

# Define the number of training steps
num_steps = 1000

# Define the lists to store the losses
gen_losses = []
dis_losses = []
```

```python
# Train the GAN
for step in range(num_steps):
    # Generate fake data
    fake_data = torch.rand(1)
    # Generate real data
    real_data = torch.rand(1)

    # Train the generator
    gen_optimizer.zero_grad()
    fake_output = generator(fake_data)
    gen_loss = loss_fn(fake_output, real_data)
    gen_loss.backward()
    gen_optimizer.step()
    gen_losses.append(gen_loss.item())

    # Train the discriminator
    dis_optimizer.zero_grad()
    real_output = discriminator(real_data)
    fake_output = discriminator(fake_output)
    dis_loss = loss_fn(real_output, fake_output)
    dis_loss.backward()
    dis_optimizer.step()
    dis_losses.append(dis_loss.item())

# Plot the losses
plt.plot(gen_losses, label='Generator Loss')
plt.plot(dis_losses, label='Discriminator Loss')
plt.legend()
plt.show()
```

Problem 91. Divergence between Distributions

- Implement a function that takes in two probability distributions `p` and `q` and calculates the KL divergence between them using the equation provided in the text.
- Implement a function that takes in two probability distributions `p` and `q` and calculates the JS divergence between them using the equation provided in the text.
- Implement a function that takes in two probability distributions `p` and `q` and a convex function `F` and calculates the Bregman divergence between them using the equation provided in the text.

Answer:

```
import numpy as np

def kl_divergence(p, q):
    return np.sum(np.where(p != 0, p * np.log(p / q), 0))

def js_divergence(p, q):
    m = (p + q) / 2
    return (kl_divergence(p, m) + kl_divergence(q, m)) / 2

def bregman_divergence(p, q, F, grad_F):
    return F(p) - F(q) - np.dot(grad_F(q), p - q)
```

Problem 92. Optimal Objective Value

Create a function `optimal_value(p_data, p_g)` that calculates the optimal value of `V` according to the equation in the text, given the probability distribution of real data `p_data` and the probability distribution of the generator `p_g`.

Answer:

```
import numpy as np

def kl_divergence(p, q):
    return np.sum(np.where(p != 0, p * np.log(p / q), 0))

def js_divergence(p, q):
    m = (p + q) / 2
    return (kl_divergence(p, m) + kl_divergence(q, m)) / 2

def optimal_value(p_data, p_g):
    js = js_divergence(p_data, p_g)
    return 2 * js - 2 * np.log(2)

p_data = [0.2, 0.3, 0.5]
p_g = [0.1, 0.4, 0.5]
print(kl_divergence(p_data, p_g)) # 0.09151622184943522
print(js_divergence(p_data, p_g)) # 0.04575811092471762
print(optimal_value(p_data, p_g)) # -1.3862943611198906
```

Problem 93. Gradient Descent Ascent

- Define a function `V(G, D)` that calculates the value of the GAN objective given a generator `G` and a discriminator `D`.
- Define a function `gradient_descent_ascent(G, D, x_data, z_data, eta_x, eta_y, num_iterations, m)` that performs gradient descent ascent on the generator `G` and the discriminator `D` using the GAN objective and the update rules provided in the text. The function should take as input the generator and discriminator neural networks, the real data `x_data`, the noise data `z_data`, the learning rates `eta_x` and `eta_y`, the number of iterations to perform `num_iterations`, and the mini-batch size `m`. The function should return the updated generator and discriminator neural networks.
- Using the provided data `x_data` and `z_data`, and initial generator and discriminator networks `G` and `D`, train the generator and discriminator networks for 100 iterations with a mini-batch size of 64 and learning rates of 0.001 for the generator and 0.0001 for the discriminator.

Answer:

```python
import numpy as np

def V(G, D):
    return np.mean(np.log(1 - D(G(z_data))))

def gradient_descent_ascent(G, D, x_data, z_data, eta_x, eta_y,
     num_iterations, m):
    for i in range(num_iterations):
        # Sample mini-batch
        idx = np.random.randint(0, x_data.shape[0], m)
        x_batch = x_data[idx]
        z_batch = z_data[idx]

        # Update generator
        grad_theta = (1/m) * np.gradient(np.log(1 - D(G(z_batch))),
            G.theta)
        G.theta -= eta_x * grad_theta

        # Update discriminator
        grad_phi = (1/m) * np.gradient(np.log(D(x_batch)) + np.log(1 -
            D(G(z_batch))), D.phi)
        D.phi += eta_y * grad_phi

    return G, D

# Initialize generator and discriminator networks
G = Generator()
D = Discriminator()

# Train the GAN
G, D = gradient_descent_ascent(G, D, x_data, z_data, 0.001, 0.0001, 100,
    64)
```

Problem 94. Optimistic Gradient Descent Ascent

- Implement the optimistic gradient descent ascent algorithm in Python.
- Test the algorithm on a simple two-dimensional function (e.g. a quadratic function) and compare the results with those obtained using the standard gradient descent ascent algorithm.

Answer:

```
import numpy as np

def optimistic_gradient_descent_ascent(x_init, y_init, grad_x, grad_y,
     eta_x, eta_y, num_iterations):
    x_prev = x_init
    y_prev = y_init
    x_current = x_init
    y_current = y_init
    for i in range(num_iterations):
        x_prev = x_current
        y_prev = y_current
        x_current = x_current - eta_x * grad_x(x_current, y_current) -
             eta_x * (grad_x(x_current, y_current) - grad_x(x_prev,
             y_prev))
        y_current = y_current + eta_y * grad_y(x_current, y_current) +
             eta_y * (grad_y(x_current, y_current) - grad_y(x_prev,
             y_prev))
    return x_current, y_current

# Define the gradient functions
def grad_x(x, y):
    return 2*x

def grad_y(x, y):
    return 2*y

# Initialize variables
x_init = 1
y_init = 1
eta_x = 0.1
eta_y = 0.1
num_iterations = 10

# Run the algorithm
x_opt, y_opt = optimistic_gradient_descent_ascent(x_init, y_init,
     grad_x, grad_y, eta_x, eta_y, num_iterations)
print("Optimized x: ", x_opt)
print("Optimized y: ", y_opt)
```

Problem 95. Generative Adversarial Network Training

- Create a simple generator neural network with a single hidden layer and an output layer with one unit. Initialize the weights randomly.
- Create a simple discriminator neural network with a single hidden layer and an output layer with one unit. Initialize the weights randomly.
- Generate synthetic data samples using the generator and concatenate them with real data samples.
- Train the discriminator network on the concatenated data samples.
- Generate new synthetic data samples and use the trained discriminator to classify them as real or fake.
- Use the classified synthetic data samples to train the generator network.
- Repeat the four steps above for a fixed number of iterations or until the generator produces samples that the discriminator cannot distinguish from real data.

Answer:

```
import numpy as np

# Step 1: Create generator network
generator = NeuralNetwork(input_size=100, hidden_size=256, output_size=1)
generator.initialize_weights()

# Step 2: Create discriminator network
discriminator = NeuralNetwork(input_size=1, hidden_size=256,
    output_size=1)
discriminator.initialize_weights()

# Step 3: Generate synthetic data samples
synth_data = generator.forward(np.random.randn(100))

# Step 4: Train discriminator on concatenated data samples
real_data = np.random.randn(100)
data = np.concatenate((synth_data, real_data))
labels = np.concatenate((np.zeros(100), np.ones(100)))
discriminator.train(data, labels)

# Step 5: Generate new synthetic data samples and classify them
synth_data = generator.forward(np.random.randn(100))
classification = discriminator.forward(synth_data)

# Step 6: Train generator on classified synthetic data samples
error = classification - np.ones(100)
generator.backward(error)

# Step 7: Repeat steps 3-6 for a fixed number of iterations
iterations = 100
for i in range(iterations):
    synth_data = generator.forward(np.random.randn(100))
    real_data = np.random.randn(100)
    data = np.concatenate((synth_data, real_data))
    labels = np.concatenate((np.zeros(100), np.ones(100)))
```

```
discriminator.train(data, labels)
synth_data = generator.forward(np.random.randn(100))
classification = discriminator.forward(synth_data)
error = classification - np.ones(100)
generator.backward(error)
```

Problem 96. Generative Adversarial Network Training

- Create a generator and discriminator neural network in PyTorch.
- Use the binary cross-entropy loss for the generator and discriminator.
- Train the discriminator using real data as positive examples and samples synthesized by the generator as negative examples.
- Once the discriminator is trained, hold the generator's parameters fixed and use the generator to synthesize samples for the discriminator to train on.
- Print the final generator and discriminator loss after training.

Answer:

```python
import torch
import torch.nn as nn
import torch.optim as optim

# Create generator and discriminator
class Generator(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc = nn.Linear(100, 784)

    def forward(self, x):
        x = self.fc(x)
        return x

class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc = nn.Linear(784, 1)

    def forward(self, x):
        x = self.fc(x)
        return x

# Create optimizers
generator = Generator()
discriminator = Discriminator()
gen_opt = optim.Adam(generator.parameters())
dis_opt = optim.Adam(discriminator.parameters())

# Binary cross-entropy loss
loss_fn = nn.BCELoss()

# Training loop
for epoch in range(50):
    for real_data in dataloader:
        # Train discriminator
        dis_opt.zero_grad()
        real_pred = discriminator(real_data)
        real_loss = loss_fn(real_pred, torch.ones(real_pred.size(0)))
        real_loss.backward()

        # Generate fake data
        noise = torch.randn(real_data.size(0), 100)
```

```
        fake_data = generator(noise)
        fake_pred = discriminator(fake_data)
        fake_loss = loss_fn(fake_pred, torch.zeros(fake_pred.size(0)))
        fake_loss.backward()
        dis_opt.step()

    # Print discriminator loss
    print("Epoch:", epoch, "Discriminator Loss:", (real_loss +
        fake_loss).item())

    # Train generator
    if epoch % 5 == 0:
        gen_opt.zero_grad()
        noise = torch.randn(real_data.size(0), 100)
        fake_data = generator(noise)
        fake_pred = discriminator(fake_data)
        gen_loss = loss_fn(fake_pred, torch.ones(fake_pred.size(0)))
        gen_loss.backward()
        gen_opt.step()

# Print final loss
print("Final Generator Loss:", gen_loss.item())
print("Final Discriminator Loss:", (real_loss + fake_loss).item())
```

Problem 97. Generative Adversarial Network Training

- Create a generator and discriminator neural network in PyTorch.
- Use the binary cross-entropy loss for the generator and discriminator.
- Train the GAN for 50 epochs and track the generator and discriminator loss.
- Add noise to the input of the generator during training as a regularization method.
- Print the final generator and discriminator loss.

Answer:

```python
import torch
import torch.nn as nn
import torch.optim as optim

# Create generator and discriminator
class Generator(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc = nn.Linear(100, 784)

    def forward(self, x):
        x = self.fc(x)
        return x

class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc = nn.Linear(784, 1)

    def forward(self, x):
        x = self.fc(x)
        return x

# Create optimizers
generator = Generator()
discriminator = Discriminator()
gen_opt = optim.Adam(generator.parameters())
dis_opt = optim.Adam(discriminator.parameters())

# Binary cross-entropy loss
loss_fn = nn.BCELoss()

# Training loop
for epoch in range(50):
    for real_data in dataloader:
        # Train discriminator
        dis_opt.zero_grad()
        real_pred = discriminator(real_data)
        real_loss = loss_fn(real_pred, torch.ones(real_pred.size(0)))
        real_loss.backward()

        # Generate fake data
        noise = torch.randn(real_data.size(0), 100)
        fake_data = generator(noise)
```

```
        # Add noise to the input
        noise = torch.randn(fake_data.size()) * 0.1
        fake_data = fake_data + noise

        fake_pred = discriminator(fake_data)
        fake_loss = loss_fn(fake_pred, torch.zeros(fake_pred.size(0)))
        fake_loss.backward()
        dis_opt.step()

        # Train generator
        gen_opt.zero_grad()
        noise = torch.randn(real_data.size(0), 100)
        fake_data = generator(noise)
        fake_pred = discriminator(fake_data)
        gen_loss = loss_fn(fake_pred, torch.ones(fake_pred.size(0)))
        gen_loss.backward()
        gen_opt.step()

    print("Epoch:", epoch, "Generator Loss:", gen_loss.item(),
         "Discriminator Loss:", (real_loss + fake_loss).item())

# Print final loss
print("Final Generator Loss:", gen_loss.item())
print("Final Discriminator Loss:", (real_loss + fake_loss).item())
```

Problem 98. Generative Adversarial Network Losses

- Implement the Wasserstein loss function as described in the text.
- Compare the Wasserstein loss with the JS divergence loss in a GAN training setup.

Answer:

```
import numpy as np
from scipy.optimize import linear_sum_assignment

def wasserstein_loss(real_data, generated_data):
    n = real_data.shape[0]
    m = generated_data.shape[0]
    distance_matrix = np.zeros((n, m))
    for i in range(n):
        for j in range(m):
            distance_matrix[i, j] = np.linalg.norm(real_data[i] -
                generated_data[j])
    row_ind, col_ind = linear_sum_assignment(distance_matrix)
    return np.sum(distance_matrix[row_ind, col_ind])

def js_divergence(real_data, generated_data):
    real_data_mean = np.mean(real_data, axis=0)
    generated_data_mean = np.mean(generated_data, axis=0)
    m = 0.5 * (real_data_mean + generated_data_mean)
    return 0.5 * (entropy(real_data_mean, m) +
        entropy(generated_data_mean, m))

def train_gan(real_data, generator, discriminator, wasserstein=True):
    if wasserstein:
        loss_fn = wasserstein_loss
    else:
        loss_fn = js_divergence
    for i in range(num_iterations):
        generated_data = generator.generate()
        discriminator_loss = loss_fn(real_data, generated_data)
        generator_loss = loss_fn(generated_data, real_data)
        generator.backpropagate(generator_loss)
        discriminator.backpropagate(discriminator_loss)
```

Problem 99. Generative Adversarial Network Losses

Implement the WGAN loss function in Python.

Answer:

```
import numpy as np

def wgan_loss(real_data, generated_data, critic):
    """
    Calculates the Wasserstein loss for a WGAN.

    Parameters:
        - real_data: numpy array of shape (batch_size, data_dim)
          representing a batch of real data.
        - generated_data: numpy array of shape (batch_size, data_dim)
          representing a batch of generated data.
        - critic: a function that takes in a numpy array and returns a
           scalar
          representing the critic's output for that input.

    Returns:
        - loss: a scalar representing the Wasserstein loss.
    """
    critic_real = critic(real_data)
    critic_generated = critic(generated_data)
    loss = np.mean(critic_real) - np.mean(critic_generated)
    return loss

# Example usage
real_data = np.random.normal(size=(32, 100))
generated_data = np.random.normal(size=(32, 100))

def critic(data):
    return np.sum(data, axis=1)

loss = wgan_loss(real_data, generated_data, critic)
print(loss)
```

Problem 100. Generative Adversarial Network Losses

- Create a GAN using TensorFlow that generates images of handwritten digits (MNIST dataset).
- Modify the generator loss function to include multiple subsequent discriminators.
- Experiment with different number of unrolling steps and observe the tradeoff between approximation quality and computation time.

Answer:

```
import tensorflow as tf
from tensorflow.keras import layers

# Load MNIST dataset
(x_train, y_train), (x_test, y_test) =
    tf.keras.datasets.mnist.load_data()
x_train = x_train.reshape(-1, 784).astype('float32') / 255

# Create generator and discriminator models
generator = tf.keras.Sequential([
    layers.Dense(256, input_dim=100, activation='relu'),
    layers.Dense(512, activation='relu'),
    layers.Dense(784, activation='sigmoid')
])

discriminator = tf.keras.Sequential([
    layers.Dense(512, input_dim=784, activation='relu'),
    layers.Dense(256, activation='relu'),
    layers.Dense(1, activation='sigmoid')
])

# Compile the models
generator.compile(optimizer='adam', loss='binary_crossentropy')
discriminator.compile(optimizer='adam', loss='binary_crossentropy')

# Create GAN by combining generator and discriminator
gan = tf.keras.Sequential([generator, discriminator])
gan.compile(optimizer='adam', loss='binary_crossentropy')

# Modify generator loss function to include multiple subsequent
    discriminators
unrolling_steps = 3
def generator_loss(y_true, y_pred):
    loss = 0
    for i in range(unrolling_steps):
        loss += -tf.math.log(discriminator(y_pred))
    return loss

# Train the GAN
for epoch in range(200):
    noise = tf.random.normal([batch_size, 100])
    generated_images = generator(noise)
    real_images = x_train[np.random.randint(0, x_train.shape[0],
        size=batch_size)]
```

```
X = np.concatenate((generated_images, real_images))
y = np.concatenate((np.zeros(batch_size), np.ones(batch_size)))

d_loss = discriminator.train_on_batch(X, y)
noise = tf.random.normal([batch_size, 100])
g_loss = gan.train_on_batch(noise, np.ones(batch_size))

if epoch % 20 == 0:
    print(f'epoch: {epoch}, discriminator loss: {d_loss}, generator
        loss: {g_loss}')
```

Problem 101. Generative Adversarial Network Architectures

Create a function that takes in a dataset of low-resolution images and a generator and discriminator neural network, and trains the generator and discriminator using a coarse-to-fine approach. The function should incrementally add layers of higher-resolution images during training, and return the trained generator and discriminator.

Answer:

```
import numpy as np
import tensorflow as tf

def train_coarse_to_fine(data, generator, discriminator):
    # Initialize generator and discriminator training
    generator.compile(optimizer='adam', loss='binary_crossentropy')
    discriminator.compile(optimizer='adam', loss='binary_crossentropy')

    # Iterate through each resolution level
    for resolution in range(1, max_resolution+1):
        # Add layers to generator for current resolution level
        generator.add(tf.keras.layers.Conv2DTranspose(filters=resolution*32,
            kernel_size=3, strides=2, padding='same', activation='relu'))

        # Select current resolution level images from dataset
        current_resolution_data = data[data[:,0] == resolution]

        # Train generator and discriminator on current resolution level
            images
        for epoch in range(1, max_epochs+1):
            # Generate fake images at current resolution
            fake_images =
                generator.predict(np.random.rand(current_resolution_data.shape[0],
                100))

            # Concatenate real and fake images
            real_images = current_resolution_data[:,1]
            X = np.concatenate((fake_images, real_images))

            # Create labels for real and fake images
            y = np.concatenate((np.zeros(fake_images.shape[0]),
                np.ones(real_images.shape[0])))

            # Train discriminator on real and fake images
            d_loss = discriminator.train_on_batch(X, y)

            # Generate random noise for generator input
            noise = np.random.rand(current_resolution_data.shape[0], 100)

            # Train generator to fool discriminator
            g_loss = generator.train_on_batch(noise,
                np.ones(current_resolution_data.shape[0]))

            # Print losses for monitoring training
            print(f'Resolution: {resolution}, Epoch: {epoch},
                Discriminator Loss: {d_loss}, Generator Loss: {g_loss}')
```

```
return generator, discriminator
```

Problem 102. Generative Adversarial Network Architectures

Implement a deep convolutional GAN (DCGAN) using a CNN as the discriminator and a deconvolution neural network as the generator. Train the DCGAN on a dataset of images and evaluate the quality of the synthesized images.

Answer:

```
import tensorflow as tf
from tensorflow.keras import layers

# Discriminator
discriminator = tf.keras.Sequential()
discriminator.add(layers.Conv2D(64, (5, 5), strides=(2, 2),
    padding='same', input_shape=[28, 28, 1]))
discriminator.add(layers.LeakyReLU())
discriminator.add(layers.Dropout(0.3))
discriminator.add(layers.Conv2D(128, (5, 5), strides=(2, 2),
    padding='same'))
discriminator.add(layers.LeakyReLU())
discriminator.add(layers.Dropout(0.3))
discriminator.add(layers.Flatten())
discriminator.add(layers.Dense(1))

# Generator
generator = tf.keras.Sequential()
generator.add(layers.Dense(7*7*256, use_bias=False, input_shape=(100,)))
generator.add(layers.BatchNormalization())
generator.add(layers.LeakyReLU())
generator.add(layers.Reshape((7, 7, 256)))
generator.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1),
    padding='same', use_bias=False))
generator.add(layers.BatchNormalization())
generator.add(layers.LeakyReLU())
generator.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2),
    padding='same', use_bias=False))
generator.add(layers.BatchNormalization())
generator.add(layers.LeakyReLU())
generator.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2),
    padding='same', use_bias=False, activation='tanh'))

# GAN
gan = tf.keras.Sequential()
gan.add(generator)
gan.add(discriminator)

# Compile and train the GAN
gan.compile(optimizer='adam', loss='binary_crossentropy')
gan.fit(x_train, y_train, epochs=10)

# Evaluate the quality of the synthesized images
synthesized_images = generator.predict(x_test)
```

Problem 103. Generative Adversarial Network Architectures

- Define a class `SGAN` that implements a semi-supervised GAN architecture.
- The class should have the following methods:
  - `__init__(self, num_classes, generator, discriminator)`: Initializes the SGAN with the given number of classes and the generator and discriminator models.
  - `train(self, real_samples, real_labels)`: Trains the SGAN using the given real samples and real labels.
  - `generate(self)`: Generates fake samples using the generator.
  - `evaluate(self, samples)`: Evaluates the samples and returns the probability of the samples being synthesized or real, and if the sample is real, the probabilities of the k classes.

Answer:

```python
import torch
from torch import nn

class SGAN:
    def __init__(self, num_classes, generator, discriminator):
        self.num_classes = num_classes
        self.generator = generator
        self.discriminator = discriminator
        self.unsupervised_loss = nn.BCEWithLogitsLoss()
        self.supervised_loss = nn.CrossEntropyLoss()

    def train(self, real_samples, real_labels):
        fake_samples = self.generator()

        # Train discriminator
        real_outputs = self.discriminator(real_samples)
        fake_outputs = self.discriminator(fake_samples)

        real_labels = torch.ones(real_samples.size(0), 1)
        fake_labels = torch.zeros(fake_samples.size(0), 1)

        # Unsupervised loss
        real_loss = self.unsupervised_loss(real_outputs, real_labels)
        fake_loss = self.unsupervised_loss(fake_outputs, fake_labels)
        unsupervised_loss = (real_loss + fake_loss) / 2

        # Supervised loss
        class_outputs = real_outputs[:, :self.num_classes]
        class_loss = self.supervised_loss(class_outputs, real_labels)
        supervised_loss = class_loss

        # Update weights
        total_loss = unsupervised_loss + supervised_loss
        self.discriminator.optimizer.zero_grad()
        total_loss.backward()
        self.discriminator.optimizer.step()
```

```python
        # Train generator
        fake_samples = self.generator()
        fake_outputs = self.discriminator(fake_samples)
        fake_loss = self.unsupervised_loss(fake_outputs, real_labels)
        self.generator.optimizer.zero_grad()
        fake_loss.backward()
        self.generator.optimizer.step()

    def generate(self):
        return self.generator()

    def evaluate(self, samples):
        outputs = self.discriminator(samples)
        synthesis_prob = torch.sigmoid(outputs[:, 0])
        class_prob = torch.softmax(outputs[:, 1:], dim=1)
        return synthesis_prob
```

Problem 104. Generative Adversarial Network Architectures

- Implement a conditional GAN in Python.
- Train the GAN on a dataset of labeled images.
- Use the trained GAN to generate new images of a specific class or with a specific attribute.

Answer:

```
import tensorflow as tf
from tensorflow import keras

# Define the generator and discriminator
generator = keras.Sequential([
    keras.layers.Dense(7*7*256, input_shape=(100,), activation='relu'),
    keras.layers.Reshape((7, 7, 256)),
    keras.layers.Conv2DTranspose(128, (5, 5), (1, 1), padding='same',
        activation='relu'),
    keras.layers.Conv2DTranspose(64, (5, 5), (2, 2), padding='same',
        activation='relu'),
    keras.layers.Conv2DTranspose(32, (5, 5), (2, 2), padding='same',
        activation='relu'),
    keras.layers.Conv2DTranspose(3, (5, 5), (2, 2), padding='same',
        activation='tanh')
])

discriminator = keras.Sequential([
    keras.layers.InputLayer(input_shape=(28, 28, 3)),
    keras.layers.Conv2D(32, (5, 5), (2, 2), padding='same',
        activation='relu'),
    keras.layers.Conv2D(64, (5, 5), (2, 2), padding='same',
        activation='relu'),
    keras.layers.Conv2D(128, (5, 5), (2, 2), padding='same',
        activation='relu'),
    keras.layers.Flatten(),
    keras.layers.Dense(1)
])

# Compile the GAN
model = keras.Sequential([generator, discriminator])
model.compile(optimizer='adam', loss='binary_crossentropy')

# Load the labeled data
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
x_train = x_train.reshape((-1, 28, 28, 1))
x_test = x_test.reshape((-1, 28, 28, 1))

# Train the GAN
model.fit(x_train, y_train, epochs=10)

# Generate new images of a specific class
class_label = 3
latent_vector = tf.random.normal((1, 100))
class_vector = tf.one_hot(class_label, depth=10)
generated_images = generator.predict([latent_vector, class_vector])
```

Problem 105. Generative Adversarial Network Architectures

Implement a Pix2Pix model using GANs to translate an input image to a synthesized image with different properties. Use the loss function provided in the text.

Answer:

```python
import tensorflow as tf
from tensorflow.keras import layers

# Define the generator model
def generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2),
        padding='same', input_shape=(100, 100, 3)))
    model.add(layers.LeakyReLU())
    model.add(layers.Conv2DTranspose(32, (5, 5), strides=(2, 2),
        padding='same'))
    model.add(layers.LeakyReLU())
    model.add(layers.Conv2DTranspose(3, (5, 5), strides=(2, 2),
        padding='same', activation='tanh'))
    return model

# Define the discriminator model
def discriminator_model():
    model = tf.keras.Sequential()
    model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',
        input_shape=[100, 100, 3]))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))
    model.add(layers.Conv2D(32, (5, 5), strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))
    model.add(layers.Flatten())
    model.add(layers.Dense(1))
    return model

# Define the Pix2Pix model
def pix2pix_model(discriminator, generator):
    model = tf.keras.Sequential()
    model.add(generator)
    model.add(discriminator)
    return model

# Define the loss function
def pix2pix_loss(y_true, y_pred, lambda_value):
    cgan_loss = tf.keras.losses.BinaryCrossentropy()(y_true, y_pred)
    pixelwise_loss = tf.keras.losses.MeanAbsoluteError()(y_true, y_pred)
    total_loss = cgan_loss + lambda_value * pixelwise_loss
    return total_loss

# Create the generator and discriminator
generator = generator_model()
discriminator = discriminator_model()

# Create the Pix2Pix model
```

```
pix2pix = pix2pix_model(discriminator, generator)

# Compile the model
pix2pix.compile(optimizer='adam', loss=lambda y_true, y_pred:
    pix2pix_loss(y_true, y_pred, 0.1))

# Train the model
pix2pix.fit(x_train, y_train, epochs=100)
```

Problem 106. Generative Adversarial Network Architectures

- Implement a function `cycle_consistency(X,Y,F,G)` where:
  - `X` is a tensor of real images from domain A
  - `Y` is a tensor of real images from domain B
  - `F` is a generator that maps images from domain A to domain B
  - `G` is a generator that maps images from domain B to domain A
- The function should apply the generators `F` and `G` to the real images `X` and `Y` respectively, and calculate the cycle consistency loss for both cycles.
- The function should return the overall cycle consistency loss, which is the sum of the loss for both cycles.

Answer:

```
import torch

def cycle_consistency(X,Y,F,G):
    # Apply generator F to X
    X_hat = F(X)
    # Calculate the cycle consistency loss for the first cycle
    first_cycle_loss = torch.mean(torch.abs(G(X_hat) - X))
    # Apply generator G to Y
    Y_hat = G(Y)
    # Calculate the cycle consistency loss for the second cycle
    second_cycle_loss = torch.mean(torch.abs(F(Y_hat) - Y))
    # Sum both loss to calculate the overall cycle consistency loss
    overall_loss = first_cycle_loss + second_cycle_loss
    return overall_loss
```

Problem 107. Generative Adversarial Network Architectures

Implement the correction loss function $\mathcal{L}_{\text{cor}}(\mathcal{G}, \mathcal{R})$ in Python.

Answer:

```python
import tensorflow as tf

def correction_loss(G, R, x, y_tilde):
    y = G(x)
    deformation_field = R(y, y_tilde)
    resampled_y = tf.contrib.resampler.resampler(y, deformation_field)
    return tf.reduce_mean(tf.abs(y - resampled_y))
```

Problem 108. Evaluation

- Implement a function `compute_IS(images)` that takes in a list of `images` and returns the Inception Score (IS) of the images. You can use the pre-trained Inception v3 model for classification.
- Implement a function `compute_FID(real_images, generated_images)` that takes in two lists of images, one for real images and one for generated images, and returns the Frechet Inception Distance (FID) of the images. You can use the pre-trained Inception v3 model to extract the feature vectors of the last layer.
- Using the provided functions, evaluate the quality of some synthesized images compared to real images.

Answer:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import models

# Load pre-trained Inception v3 model
inception = models.inception_v3(pretrained=True)

# Freeze all layers
for param in inception.parameters():
    param.requiresGrad = False

# Replace the last layer with a linear layer
num_ftrs = inception.fc.in_features
inception.fc = nn.Linear(num_ftrs, 1000)

# Move model to GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
inception = inception.to(device)

# Compute IS
def compute_IS(images):
    inception.eval()
    with torch.no_grad():
        images = torch.stack(images)
        images = images.to(device)
        logits = inception(images)
        probs = F.softmax(logits, dim=1)
        KL = probs * (torch.log(probs) - torch.tensor(probs.mean(0),
            requires_grad=False))
        KL = KL.sum(1).mean()
    return torch.exp(KL)

# Compute FID
def compute_FID(real_images, generated_images):
    inception.eval()
    with torch.no_grad():
        real_images = torch.stack(real_images)
        generated_images = torch.stack(generated_images)
```

```
    real_images = real_images.to(device)
    generated_images = generated_images.to(device)
    real_features = inception(real_images)
    generated_features = inception(generated_images)
    real_features = real_features.cpu().numpy()
    generated_features = generated_features.cpu().numpy()
    mu_real = real_features.mean(axis=0)
    mu_generated = generated_features.mean(axis=0)
    sigma_real = np.cov(real_features, rowvar=False)
    sigma_generated = np.cov(generated_features, rowvar=False)
    FID = (mu_real - mu_generated) @ (mu_real - mu_generated) +
        np.trace(sigma_real + sigma_generated - 2*np.sqrt(sigma_real
        @ sigma_generated))
return FID
```

Problem 109. Variational Inference

- Define a simple probabilistic model with a single hidden variable z and a single observed variable x. Assume that z is a scalar value and x is a binary value (0 or 1).
- Define the prior density $p(z)$ as a normal distribution with mean 0 and standard deviation 1.
- Define the likelihood function $p(x|z)$ as a Bernoulli distribution with a probability of success equal to `sigmoid(z)`.
- Using the definition of the posterior $p(z|x)$ in the text, compute the posterior for a given value of x.
- Plot the prior density, likelihood function, and posterior density in a single figure for different values of x.

Answer:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
from scipy.special import expit

# Define the probabilistic model
def p(z, x):
    p_z = norm.pdf(z, 0, 1) # prior density
    p_x_given_z = expit(z) if x == 1 else 1 - expit(z) # likelihood
        function
    return p_z * p_x_given_z

# Compute the posterior density
def p_z_given_x(z, x):
    p_x = 0.5 # assume equal probability for x = 0 and x = 1
    return p(z, x) / (p(z, 0) + p(z, 1))

# Plot the prior density, likelihood function, and posterior density
z = np.linspace(-5, 5, 100)
x = 1
plt.plot(z, norm.pdf(z, 0, 1), label='prior')
plt.plot(z, expit(z) if x == 1 else 1 - expit(z), label='likelihood')
plt.plot(z, p_z_given_x(z, x), label='posterior')
plt.legend()
plt.show()
```

Problem 110. Variational Autoencoder

- Implement a single-layer autoencoder in Python
- Train the autoencoder on a dataset of your choice (e.g. MNIST)
- Compute the reconstruction error as the mean squared error between the original data and the reconstruction

Answer:

```
import tensorflow as tf
from tensorflow.keras import layers

# define the model
inputs = tf.keras.Input(shape=(784,))
encoded = layers.Dense(32, activation='relu')(inputs)
decoded = layers.Dense(784, activation='sigmoid')(encoded)
autoencoder = tf.keras.Model(inputs, decoded)

# compile the model
autoencoder.compile(optimizer='adam', loss='mean_squared_error')

# load the data
(x_train, _), (x_test, _) = tf.keras.datasets.mnist.load_data()
x_train = x_train.reshape((60000, 784)) / 255.
x_test = x_test.reshape((10000, 784)) / 255.

# train the model
autoencoder.fit(x_train, x_train, epochs=10, batch_size=256,
    shuffle=True, validation_data=(x_test, x_test))

# evaluate the model
reconstruction_error = tf.keras.losses.MeanSquaredError()(x_test,
    autoencoder.predict(x_test))
print("Reconstruction error:", reconstruction_error)
```

Problem 111. Variational Autoencoder

- Create a simple Variational Autoencoder in Python.
- The autoencoder should have a bottleneck of low-dimensional variable z of size 2.
- The input dataset to the autoencoder should be a 2D dataset of points in $\mathbb{R}^2$.
- Train the autoencoder using the mean squared error loss function.
- Plot the input dataset and the reconstructed dataset after training.

Answer:

```python
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt

# Define the dataset
input_data = torch.randn(100, 2)

# Define the encoder
class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        self.linear1 = nn.Linear(2, 10)
        self.linear2 = nn.Linear(10, 2)

    def forward(self, x):
        x = torch.relu(self.linear1(x))
        x = self.linear2(x)
        return x

# Define the decoder
class Decoder(nn.Module):
    def __init__(self):
        super(Decoder, self).__init__()
        self.linear1 = nn.Linear(2, 10)
        self.linear2 = nn.Linear(10, 2)

    def forward(self, x):
        x = torch.relu(self.linear1(x))
        x = self.linear2(x)
        return x

# Define the autoencoder
encoder = Encoder()
decoder = Decoder()
autoencoder = nn.Sequential(encoder, decoder)

# Define the loss function and optimizer
criterion = nn.MSELoss()
optimizer = optim.Adam(autoencoder.parameters())

# Train the autoencoder
for epoch in range(1000):
    optimizer.zero_grad()
```

```
    encoded = encoder(input_data)
    reconstructed = decoder(encoded)
    loss = criterion(reconstructed, input_data)
    loss.backward()
    optimizer.step()

# Plot the input data and the reconstructed data
plt.scatter(input_data[:, 0], input_data[:, 1], label='input data')
plt.scatter(reconstructed[:, 0], reconstructed[:, 1],
    label='reconstructed data')
plt.legend()
plt.show()
```

Problem 112. Variational Autoencoder

- Implement a simple Variational Autoencoder (VAE) in Python.
- Compute the ELBO for a given data set.
- Using stochastic gradient descent, maximize the ELBO by updating the parameters of the encoder and decoder.

Answer:

```
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim

# Define the encoder and decoder as PyTorch nn.Module classes
class Encoder(nn.Module):
    def __init__(self, input_size, hidden_size, latent_size):
        super(Encoder, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, latent_size)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

class Decoder(nn.Module):
    def __init__(self, input_size, hidden_size, latent_size):
        super(Decoder, self).__init__()
        self.fc1 = nn.Linear(latent_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, input_size)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Define the VAE as a combination of the encoder and decoder
class VAE(nn.Module):
    def __init__(self, input_size, hidden_size, latent_size):
        super(VAE, self).__init__()
        self.encoder = Encoder(input_size, hidden_size, latent_size)
        self.decoder = Decoder(input_size, hidden_size, latent_size)

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x

# Define the loss function as the negative ELBO
def loss_fn(recon_x, x, mu, logvar):
    BCE = nn.functional.binary_cross_entropy(recon_x, x, reduction='sum')
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return BCE + KLD
```

```python
# Generate some data
data = torch.randn(100, 10)

# Instantiate the VAE and optimizer
vae = VAE(10, 20, 2)
optimizer = optim.Adam(vae.parameters())

# Training loop
for i in range(1000):
    optimizer.zero_grad()
    recon_x, mu, logvar = vae(data)
    loss = loss_fn(recon_x, data, mu, logvar)
    loss.backward()
    optimizer.step()
```

Problem 113. Generative Flows

- Implement a normalizing flow using a simple density, such as a Gaussian, and an invertible function, such as a Planar Flow or Radial Flow.
- Use this normalizing flow to transform the simple density to a more complex density.
- Compare the log-likelihood of the transformed density to the original simple density and show that the transformed density has a higher log-likelihood.

Answer:

```
import numpy as np
from scipy.stats import multivariate_normal

class NormalizingFlow:
    def __init__(self, flow_type='planar'):
        self.flow_type = flow_type
        self.u = None
        self.w = None
        self.b = None

    def fit(self, x):
        self.x = x
        if self.flow_type == 'planar':
            self.u = np.random.normal(size=x.shape[1])
            self.w = np.random.normal(size=x.shape[1])
            self.b = np.random.normal()
        elif self.flow_type == 'radial':
            self.c = np.random.normal(size=x.shape[1])

    def transform(self, x):
        if self.flow_type == 'planar':
            # Planar flow transformation
            h = np.dot(x, self.w) + self.b
            m = np.dot(self.u, self.w)
            z = x + (np.exp(h) - 1)*self.u/(np.linalg.norm(self.w)**2 +
                1e-10)
            log_det_jacobian = h - m
        elif self.flow_type == 'radial':
            # Radial flow transformation
            z = x - self.c
            r = np.linalg.norm(z, axis=1)
            log_det_jacobian = np.log(r + 1e-10)
            z = z/r[:, None]
        return z, log_det_jacobian

# Generate data from a simple Gaussian distribution
mean = np.zeros(2)
cov = np.eye(2)
data = multivariate_normal.rvs(mean, cov, size=1000)

# Create and fit the normalizing flow
flow = NormalizingFlow()
flow.fit(data)
```

```python
# Transform the data using the flow
transformed_data, log_det_jacobian = flow.transform(data)

# Compare the log-likelihood of the transformed data to the original data
simple_density = multivariate_normal(mean, cov)
transformed_density = multivariate_normal(mean, cov, allow_singular=True)
print("Simple density log-likelihood: ",
    simple_density.logpdf(data).mean())
print("Transformed density log-likelihood: ",
    transformed_density.logpdf(transformed_data) +
    log_det_jacobian.sum())
```

Problem 114. Generative Flows

Implement a planar flow family of transformations in Python using the equations given in the text.

- Define the smooth differentiable non-linear function `h`. (You can use the `activation` function of your choice, such as ReLU, sigmoid, etc.)
- Implement the invertible function `f`.
- Implement the log-det Jacobian $\log\det(\frac{\partial f}{\partial z})$.
- Write a function to generate samples from a Gaussian distribution, and use the planar flow family of transformations to transform the Gaussian samples into samples from a more complex distribution.
- Visualize the transformed samples in a 2D plot, and compare the plot to a 2D plot of the original Gaussian samples.

Answer:

```
import numpy as np
import matplotlib.pyplot as plt

def h(x, activation='relu'):
    if activation == 'relu':
        return np.maximum(0, x)
    elif activation == 'sigmoid':
        return 1 / (1 + np.exp(-x))

def f(z, u, w, b, h):
    return z + u * h(np.dot(w, z) + b)

def log_det_jacobian(z, u, w, b, h):
    psi = h(np.dot(w, z) + b, derivative=True) * w
    return np.log(np.abs(1 + np.dot(u, psi)))

def generate_gaussian_samples(mean, covariance, num_samples):
    samples = np.random.multivariate_normal(mean, covariance, num_samples)
    return samples

def transform_samples(samples, u, w, b, h):
    transformed_samples = np.zeros_like(samples)
    for i, sample in enumerate(samples):
        transformed_samples[i, :] = f(sample, u, w, b, h)
    return transformed_samples

mean = [0, 0]
covariance = [[1, 0], [0, 1]]
num_samples = 1000
samples = generate_gaussian_samples(mean, covariance, num_samples)

u = np.array([1, 0])
w = np.array([[1, 0], [0, 1]])
b = 0

transformed_samples = transform_samples(samples, u, w, b, h)
```

```
plt.scatter(samples[:, 0], samples[:, 1], label='Original samples')
plt.scatter(transformed_samples[:, 0], transformed_samples[:, 1],
    label='Transformed samples')
plt.legend()
plt.show()
```

Problem 115. Denoising Diffusion Probabilistic Model

Implement a Denoising Diffusion Probabilistic Model (DDPM) in Python. You need to implement a class DDPM with the following methods:

- `__init__(self, num_layers: int, learning_rate: float)`: Initialize the class with the number of layers in the model and the learning rate.
- `build_model(self, shape: tuple)`: Build the DDPM model with the given shape of the input.
- `fit(self, X: np.ndarray, epochs: int)`: Train the DDPM model on the input data X for epochs number of epochs.
- `generate(self, X: np.ndarray)`: Generate new signals from the input noise X.

You can use the TensorFlow library to implement the DDPM model.

Answer:

```
import tensorflow as tf
import numpy as np

class DDPM:
    def __init__(self, num_layers: int, learning_rate: float):
        self.num_layers = num_layers
        self.learning_rate = learning_rate

    def build_model(self, shape: tuple):
        self.input_layer = tf.keras.Input(shape=shape)
        x = self.input_layer

        for i in range(self.num_layers):
            x = tf.keras.layers.Dense(128, activation='relu')(x)

        self.output_layer = tf.keras.layers.Dense(shape[0],
            activation='sigmoid')(x)
        self.model = tf.keras.Model(inputs=self.input_layer,
            outputs=self.output_layer)
        self.model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=self.learning_ra
                       loss=tf.keras.losses.binary_crossentropy)

    def fit(self, X: np.ndarray, epochs: int):
        self.model.fit(X, X, epochs=epochs, verbose=0)

    def generate(self, X: np.ndarray):
        return self.model.predict(X)
```

Problem 116. Denoising Diffusion Probabilistic Model

Implement a DDPM that starts with a given distribution $x_0 \sim q(x_0)$, generates a sequence $(x_1, \ldots, x_T)$ by iteratively adding Gaussian noise, and returns the last element in the sequence x_T.

Input:

- A function `q` that returns random variables from the distribution $x_0 \sim q(x_0)$.
- A positive integer `T` representing the number of iterations.
- A list `betas` of length `T` containing the values of β_t for each iteration t.

Output:

- A variable `xT` representing the last element in the sequence x_T.

Answer:

```
import numpy as np

def ddpm(q, T, betas):
    x0 = q()
    xT = x0
    alpha = 1
    for t in range(T):
        beta = betas[t]
        alpha *= 1 - beta
        xT = np.sqrt(alpha) * xT + np.sqrt(1 - alpha) *
            np.random.normal(size=x0.shape)
    return xT
```

Problem 117. Geometric Variational Inference

Implement the following functions for a Riemannian manifold:

- `exp_map`: Given a point `p0` on the manifold and a tangent vector `theta` $\in Tp_0(M)$, compute the exponential map $\exp_{p_0}(\theta)$.
- `log_map`: Given two points `p0` and `p1` on the manifold, compute the logarithm map $\log_{p_0}(p_1)$.
- `metric`: Given a point `p0` on the manifold and a tangent vector $\theta \in Tp_0(M)$, compute the quadratic form $\sum_{ij} g_{ij}(p)\theta_i\theta_j$.

Answer: Here is a simple example for a 2-dimensional sphere.

```
import numpy as np

def exp_map(p0, theta):
    # Normalize theta
    theta = theta / np.linalg.norm(theta)

    # Compute the exponential map
    p1 = p0 * np.cos(np.linalg.norm(theta)) +
        np.sin(np.linalg.norm(theta)) * theta

    return p1

def log_map(p0, p1):
    # Compute the logarithm map
    theta = np.arccos(np.dot(p0, p1)) * (p1 - np.dot(p0, p1) * p0) /
        np.linalg.norm(p1 - np.dot(p0, p1) * p0)

    return theta

def metric(p0, theta):
    # Compute the quadratic form
    quadratic_form = np.linalg.norm(theta)**2

    return quadratic_form
```

Problem 118. Geometric Variational Inference

Implement a simple VAE model in Python using a Riemannian metric for interpolation in the latent space. The model should be able to generate new samples and perform interpolation.

Answer:

```
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim

# Define the encoder network
class Encoder(nn.Module):
    def __init__(self, input_dim, hidden_dim, latent_dim):
        super(Encoder, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, latent_dim)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Define the decoder network
class Decoder(nn.Module):
    def __init__(self, latent_dim, hidden_dim, output_dim):
        super(Decoder, self).__init__()
        self.fc1 = nn.Linear(latent_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Define the VAE model
class VAE(nn.Module):
    def __init__(self, input_dim, hidden_dim, latent_dim):
        super(VAE, self).__init__()
        self.encoder = Encoder(input_dim, hidden_dim, latent_dim)
        self.decoder = Decoder(latent_dim, hidden_dim, input_dim)

    def forward(self, x):
        z = self.encoder(x)
        x_hat = self.decoder(z)
        return x_hat

# Train the VAE model
def train(model, train_loader, criterion, optimizer, num_epochs):
    for epoch in range(num_epochs):
        for i, data in enumerate(train_loader):
            inputs, labels = data
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, inputs)
```

```
            loss.backward()
            optimizer.step()

        print(f"Epoch [{epoch + 1}/{num_epochs}], Loss: {loss.item()}")

# Generate new samples
def generate_samples(model, num_samples, latent_dim):
    z = torch.randn(num_samples, latent_dim)
    x_hat = model.decoder(z)
    return x_hat

# Interpolate in the latent space
def interpolate(model, num_samples, latent_dim, start, end):
    z = torch.linspace(start, end, num_samples)
    x_hat = model.decoder(z)
    return x_hat
```

11 Reinforcement Learning

Problem 119. Multi-Armed Bandit

- Implement a class `Bandit` that simulates the multi-armed bandit problem with `k` arms. Each arm should have a probability distribution of reward, which is drawn when the arm is pulled.
- Implement another class `Agent` that interacts with the `Bandit` object. The agent should have an action-value function that keeps track of the estimated value of each arm. The agent should have a method `choose_action` that selects an action according to the greedy strategy described in the text. The agent should have another method `update_action_value` that updates the estimated value of the chosen action based on the received reward.
- Run a simulation with 1000 time steps and initialize the agent with an action-value function that assigns equal values to all actions. Plot the average reward per time step for the agent over the simulation.

Answer:

```
import random
import numpy as np
import matplotlib.pyplot as plt

class Bandit:
    def __init__(self, k, distribution_params):
        self.k = k
        self.distribution_params = distribution_params

    def pull(self, action):
        mu, sigma = self.distribution_params[action]
        return np.random.normal(mu, sigma)

class Agent:
    def __init__(self, k):
        self.k = k
        self.action_values = np.zeros(k)
        self.action_counts = np.zeros(k)

    def choose_action(self):
        return np.argmax(self.action_values)

    def update_action_value(self, action, reward):
        self.action_counts[action] += 1
        alpha = 1/self.action_counts[action]
        self.action_values[action] += alpha * (reward -
            self.action_values[action])

if __name__ == "__main__":
    k = 10
    distribution_params = [(np.random.randn(), 1) for _ in range(k)]
    bandit = Bandit(k, distribution_params)
```

```python
agent = Agent(k)

rewards = []
for t in range(1000):
    action = agent.choose_action()
    reward = bandit.pull(action)
    agent.update_action_value(action, reward)
    rewards.append(reward)

plt.plot(np.cumsum(rewards) / (np.arange(1000) + 1))
plt.xlabel("Time step")
plt.ylabel("Average reward")
plt.show()
```

Problem 120. State Machines

Design a state machine that represents the states of a traffic light system. The states should include "red", "yellow", and "green". The inputs should include "switch" which triggers a transition between states. The outputs should indicate the current state of the traffic light system.

Answer:

```
class TrafficLight:
    def __init__(self):
        self.states = ["red", "yellow", "green"]
        self.inputs = ["switch"]
        self.current_state = "red"

    def transition(self, input_value):
        if input_value not in self.inputs:
            return "Invalid input"

        current_index = self.states.index(self.current_state)
        next_index = (current_index + 1) % len(self.states)
        self.current_state = self.states[next_index]

    def output(self):
        return self.current_state

traffic_light = TrafficLight()
print(traffic_light.output()) # Output: red
traffic_light.transition("switch")
print(traffic_light.output()) # Output: green
```

Problem 121. Markov Processes

Create a Markov process simulation in Python. Your simulation should take in two parameters: the number of states `num_states` and the number of steps `num_steps`. The simulation should simulate a Markov process with the following rules:

- The state at time step 0 is randomly initialized.
- The probability of transitioning from one state to another is 0.5 for each possible transition.
- The output of each step is the current state.

Answer:

```python
import random

def markov_process(num_states, num_steps):
    states = range(num_states)
    current_state = random.choice(states)
    output = [current_state]
    for step in range(1, num_steps):
        next_state = random.choice(states) if random.random() < 0.5 else
            current_state
        current_state = next_state
        output.append(current_state)
    return output
```

Problem 122. Markov Processes

- Implement a class `MarkovProcess` that contains the following methods:
  - `__init__(self, states, actions, transition_model)`: Initializes the `states`, `actions`, and `transition_model` of the Markov process.
  - `get_transition_prob(self, state, action, next_state)`: Returns the transition probability $p(s_{t+1} = next\_state | s_t = state, a_t = action)$.
  - `get_next_states(self, state, action)`: Returns a list of states that can be reached from the state `state` with the action `action`.
- Test your implementation with the following code:

```
states = {'s1', 's2', 's3'}
actions = {'a1', 'a2'}
transition_model = {
    ('s1', 'a1'): {'s1': 0.3, 's2': 0.7},
    ('s1', 'a2'): {'s2': 1.0},
    ('s2', 'a1'): {'s1': 0.4, 's3': 0.6},
    ('s2', 'a2'): {'s3': 1.0},
    ('s3', 'a1'): {'s1': 0.5, 's2': 0.5},
    ('s3', 'a2'): {'s1': 1.0}
}

mp = MarkovProcess(states, actions, transition_model)
print(mp.get_transition_prob('s1', 'a1', 's2')) # Output: 0.7
print(mp.get_next_states('s2', 'a2')) # Output: ['s3']
```

Answer:

```
class MarkovProcess:
    def __init__(self, states, actions, transition_model):
        self.states = states
        self.actions = actions
        self.transition_model = transition_model

    def get_transition_prob(self, state, action, next_state):
        return self.transition_model.get((state, action),
            {}).get(next_state, 0)

    def get_next_states(self, state, action):
        return list(self.transition_model.get((state, action), {}).keys())
```

Problem 123. Markov Decision Processes

Create a simple MDP where the agent can be in one of two states (`S1, S2`) and can take one of two actions (`A1, A2`). The transition model should have a 0.8 probability of staying in the current state and a 0.2 probability of transitioning to the other state. The reward function should return a reward of 10 if the agent is in `S1` and takes action `A1`, a reward of 5 if the agent is in `S1` and takes action `A2`, a reward of 2 if the agent is in `S2` and takes action `A1`, and a reward of 1 if the agent is in `S2` and takes action `A2`. Use a discount factor of 0.9.

Answer:

```python
import numpy as np

class MDP:
    def __init__(self):
        self.states = ['S1', 'S2']
        self.actions = ['A1', 'A2']
        self.transition_model = np.array([[0.8, 0.2], [0.2, 0.8]])
        self.reward_function = np.array([[10, 5], [2, 1]])
        self.discount_factor = 0.9

    def step(self, current_state, action):
        next_state = np.random.choice(self.states,
            p=self.transition_model[self.states.index(current_state), :])
        reward = self.reward_function[self.states.index(current_state),
            self.actions.index(action)]
        return next_state, reward

    def value_iteration(self):
        # Initialize value function
        value_function = np.zeros(len(self.states))

        # Loop until convergence
        while True:
            new_value_function = np.zeros(len(self.states))
            for state in self.states:
                for action in self.actions:
                    next_state, reward = self.step(state, action)
                    new_value_function[self.states.index(state)] +=
                        self.transition_model[self.states.index(state),
                        self.states.index(next_state)]*(reward +
                        self.discount_factor*value_function[self.states.index(next_state)])
            if np.sum(np.abs(new_value_function - value_function)) < 1e-5:
                break
            value_function = new_value_function
        return value_function

mdp = MDP()
print(mdp.value_iteration())
```

Problem 124. Definitions

- Implement a class called `Policy` that takes in a dictionary `policy_dict` of states and their corresponding possible actions and probabilities as input. The class should have a method called `choose_action` which takes in a `state` as input and returns a randomly chosen action based on the probabilities in the input dictionary.
- Create an instance of the `Policy` class using the following dictionary:

```
{
"state1": {"action1": 0.3, "action2": 0.6, "action3": 0.1},
"state2": {"action1": 0.5, "action2": 0.3, "action3": 0.2},
"state3": {"action1": 0.7, "action2": 0.2, "action3": 0.1},
}
```

- Use the `choose_action` method from the instance of the `Policy` class to randomly select and print an action for each of the states in the dictionary.

Answer:

```
import random

class Policy:
    def __init__(self, policy_dict):
        self.policy_dict = policy_dict

    def choose_action(self, state):
        actions = self.policy_dict[state]
        action_probs = [actions[action] for action in actions]
        chosen_action = random.choices(list(actions.keys()),
            action_probs)[0]
        return chosen_action

policy_dict = {
    "state1": {"action1": 0.3, "action2": 0.6, "action3": 0.1},
    "state2": {"action1": 0.5, "action2": 0.3, "action3": 0.2},
    "state3": {"action1": 0.7, "action2": 0.2, "action3": 0.1},
}

policy = Policy(policy_dict)

for state in policy_dict.keys():
    print(f"For state {state}, action chosen:
        {policy.choose_action(state)}")
```

Problem 125. Definitions

- Define a function `state_value_function(h, s, pi, R, T)` that takes in 5 arguments:
  - `h`: an integer representing the horizon
  - `s`: an integer representing the current state
  - `pi`: a function that takes in an integer state and returns an action
  - `R`: a dictionary that maps a tuple of (state, action) to a float representing the reward
  - `T`: a dictionary that maps a tuple of (state, action, next_state) to a float representing the transition probability
- The function should return the state value function $V_\pi^h(s)$ as defined in the text above using the provided `R` and `T`.
- Use recursion to compute the state value function.

Example:

```
R = {(0, 'a'): 1, (1, 'b'): 2}
T = {(0, 'a', 1): 0.5, (1, 'b', 0): 0.8}
pi = lambda s: 'a' if s == 0 else 'b'

assert state_value_function(2, 0, pi, R, T) == 2.5
```

Answer:

```
def state_value_function(h, s, pi, R, T):
    if h == 0:
        return 0
    if h == 1:
        return R[(s, pi(s))] + 0
    return R[(s, pi(s))] + sum(T[(s, pi(s), s_prime)] *
        state_value_function(h-1, s_prime, pi, R, T) for s_prime in
        T.keys())
```

Problem 126. Definitions

- Implement a function `q_value_iteration(T, R, h, s, a)` that takes in a transition probability matrix `T`, a reward matrix `R`, a horizon `h`, a current state `s`, and an action `a` and returns the action value function $Q_\pi^h(s,a)$ as defined in the text.
- Implement a function `q_value_iteration_all(T, R, h)` that takes in a transition probability matrix `T`, a reward matrix `R`, and a horizon `h`, and returns a matrix of size $n \times m \times h$, where `n` is the number of states, `m` is the number of actions, and the element at position (i, j, k) is the action value function $Q_\pi^k(i,j)$.

Answer:

```
def q_value_iteration(T, R, h, s, a):
    if h == 0:
        return 0
    elif h == 1:
        return R[s][a]
    else:
        next_states = T[s][a]
        next_values = [max(R[next_s]) + q_value_iteration(T, R, h-1,
            next_s, next_a) for next_s, next_a in next_states.items()]
        return R[s][a] + sum(next_values)

def q_value_iteration_all(T, R, h):
    n = len(R)
    m = len(R[0])
    q_values = [[[q_value_iteration(T, R, k, i, j) for k in range(h+1)]
        for j in range(m)] for i in range(n)]
    return q_values
```

Problem 127. Definitions

- Create a function `model_based_RL(policy, observations)` that takes in a initial `policy` and a list of `observations`.
- The function should use the `observations` to update the `policy` and return the updated policy.

Answer:

```
def model_based_RL(policy, observations):
    # update policy based on observations
    # For example, updating the policy with a rule such as:
    # if observation[0] == 'A' and observation[1] == 'B':
    #     policy[0] = 'C'
    return updated_policy

# test the function
initial_policy = ['A', 'B', 'C']
observations = [['A', 'B'], ['B', 'C'], ['A', 'C']]
updated_policy = model_based_RL(initial_policy, observations)
print(updated_policy) # Output: ['C', 'B', 'C']
```

Problem 128. Definitions

- Create a simple model-free agent that learns a policy.
- The agent will interact with an environment that has two states and two actions.
- The agent will start with a random policy.
- The agent will update its policy after each interaction with the environment.
- The agent should learn the optimal policy after a certain number of interactions.

Answer:

```
import numpy as np

# Initialize the agent's policy
policy = np.random.rand(2, 2)
policy /= np.sum(policy, axis=1, keepdims=True)
print("Initial policy:")
print(policy)

# Define the transition function
transition_function = np.array([[0.8, 0.2], [0.1, 0.9]])
# Define the reward function
reward_function = np.array([[1, 0], [0, 1]])

# Interact with the environment for a certain number of steps
num_steps = 100
for i in range(num_steps):
    # Choose a state
    state = np.random.randint(2)
    # Choose an action according to the current policy
    action = np.random.choice(2, p=policy[state])
    # Observe the next state and reward
    next_state = np.random.choice(2, p=transition_function[state, action])
    reward = reward_function[state, action]
    # Update the policy
    policy[state] *= 0.9
    policy[state, next_state] += 0.1
    policy /= np.sum(policy, axis=1, keepdims=True)

# The agent should have learned the optimal policy
print("Final policy:")
print(policy)
```

Problem 129. Definitions

- Create a function called `bellman_equation` that takes in the following parameters:
  - `V_pi`: a dictionary representing the value of each state under policy `pi`
  - `pi`: a dictionary representing the policy for each state
  - `p`: a dictionary representing the probability of transitioning to a new state and receiving a reward given a current state and action
  - `gamma`: a float representing the discount factor
  - `s`: the current state
- The function should return the expected return starting from state `s` and following policy `pi` according to the Bellman equation provided in the text.
- Use the `bellman_equation` function to calculate the value of state `s` under policy `pi` with a `gamma` of 0.9, given the following dictionaries:

```
V_pi = {'s1': 0, 's2': 1, 's3': 2}
pi = {'s1': 0.2, 's2': 0.5, 's3': 0.3}
p = {('s1', 's2', 0.5), ('s2', 's3', 0.2), ('s3', 's1', 0.3)}
```

Answer:

```
def bellman_equation(V_pi, pi, p, gamma, s):
    expected_return = 0
    for a in pi.keys():
        for s_prime, r in p:
            if s_prime == s:
                expected_return += pi[a] * p[(s, a, s_prime, r)] * (r +
                    gamma * V_pi[s_prime])
    return expected_return

V_pi = {'s1': 0, 's2': 1, 's3': 2}
pi = {'s1': 0.2, 's2': 0.5, 's3': 0.3}
p = {('s1', 's2', 0.5), ('s2', 's3', 0.2), ('s3', 's1', 0.3)}
s = 's1'
gamma = 0.9

print(bellman_equation(V_pi, pi, p, gamma, s))
```

Problem 130. Definitions

Create a program that calculates the value of a given state using the Bellman expectation equation. The program should take as input the `state`, the `policy`, the probability of transitioning to the next state `transition_prob`, and the expected `reward` for each action. The program should also take as input a discount factor, `gamma`. The program should output the value of the given state.

Answer:

```
import numpy as np

def calculate_value(state, policy, transition_prob, reward, gamma):
    value = 0
    for a in range(len(policy)):
        for s_prime in range(len(transition_prob)):
            for r in range(len(reward)):
                value += policy[a][state] *
                    transition_prob[s_prime][r][state][a] *
                    (reward[r][state][a] + gamma *
                    calculate_value(s_prime, policy, transition_prob,
                    reward, gamma))
    return value

# Example usage
policy = [[0.2, 0.3, 0.5], [0.1, 0.7, 0.2], [0.4, 0.1, 0.5]]
transition_prob = [[[0.2, 0.4, 0.4], [0.3, 0.3, 0.4], [0.1, 0.5, 0.4]],
    [[0.5, 0.3, 0.2], [0.1, 0.6, 0.3], [0.2, 0.2, 0.6]]]
reward = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
gamma = 0.8

print(calculate_value(0, policy, transition_prob, reward, gamma))
# Output: 3.836
```

Problem 131. Definitions

Given a list of `states`, `actions`, and `rewards`, implement a function `bellman_expectation_qpi(states, actions, rewards, gamma)` that calculates the Q-value of each state-action pair using the Bellman expectation equation. The function should return a dictionary where the keys are state-action pairs and the values are the corresponding Q-values.

Example:

```
states = [0, 1, 2, 3]
actions = [0, 1, 2]
rewards = {(0,0): 1, (0,1): 2, (1,0): 3, (1,1): 4, (2,2): 5, (3,1): 6}

print(bellman_expectation_qpi(states, actions, rewards, 0.9))
```

Output:

```
{(0,0): 2.299999999999997, (0,1): 3.4, (0,2): 0, (1,0): 4.5, (1,1): 5.6,
    (1,2): 0, (2,0): 0, (2,1): 0, (2,2): 6.5, (3,0): 0, (3,1): 7.6,
    (3,2): 0}
```

Answer:

```
def bellman_expectation_qpi(states, actions, rewards, gamma):
    q_values = {}
    for s in states:
        for a in actions:
            q_values[(s,a)] = sum(rewards.get((s_, a_), 0) + gamma *
                max(q_values.get((s_, a_), 0) for a_ in actions) for s_,
                _, p in transition_model(s,a))
    return q_values

def transition_model(s, a):
    # define the transition model
    return [(s_, r, p) for (s_, r, p) in if (s,a) == (s_, r, p)]
```

Problem 132. Optimal Policy

Given two policies `pi` and `pi_prime` represented as dictionaries mapping states to action probabilities and a list of `states`, write a function `is_policy_better_than(pi, pi_prime, states)` that returns True if policy `pi` is better than or equal to `pi_prime` for all `states`, False otherwise.

Answer:

```
def is_policy_better_than(pi, pi_prime, states):
    for state in states:
        if pi[state] < pi_prime[state]:
            return False
    return True

pi = {'s1': 0.2, 's2': 0.3, 's3': 0.5}
pi_prime = {'s1': 0.1, 's2': 0.4, 's3': 0.5}
states = ['s1', 's2', 's3']
print(is_policy_better_than(pi, pi_prime, states)) # False
```

Problem 133. Optimal Policy

Given a finite horizon MDP, write a function `find_optimal_policy(Q_h)` that takes in the action value function `Q_h` and returns the optimal finite horizon policy, `pi_h_star`.

Answer:

```
def find_optimal_policy(Q_h):
    pi_h_star = {}
    for s in Q_h.keys():
        pi_h_star[s] = max(Q_h[s], key=Q_h[s].get)
    return pi_h_star
```

Problem 134. Optimal Policy

- Create a Python class MDP that represents a Markov Decision Process with the following attributes:
  - states: a list of all states in the MDP
  - actions: a list of all actions in the MDP
  - transition_prob: a dictionary that represents the transition probabilities of the MDP, where the keys are tuples (s, a) and the values are dictionaries that represent the probability of going to each state s' when taking action a in state s.
  - rewards: a dictionary that represents the rewards of the MDP, where the keys are tuples (s, a, s') and the values are the rewards for transitioning from state s to state s' when taking action a.
  - discount_factor: a float between 0 and 1 representing the discount factor of the MDP.
- Create a method bellman_optimality_vstar() that computes the optimal state value function $V_\star$ for the MDP using the Bellman optimality equation.
- Create a method policy_from_vstar() that computes the optimal policy $\pi^\star$ for the MDP using the optimal state value function vstar.

Answer:

```python
class MDP:
    def __init__(self, states, actions, transition_prob, rewards,
        discount_factor):
        self.states = states
        self.actions = actions
        self.transition_prob = transition_prob
        self.rewards = rewards
        self.discount_factor = discount_factor

    def bellman_optimality_vstar(self):
        vstar = {s: 0 for s in self.states}
        while True:
            vstar_prev = vstar.copy()
            for s in self.states:
                vstar[s] = max(
                    sum(self.transition_prob[s, a][s_] * (self.rewards[s,
                        a, s_] + self.discount_factor * vstar_prev[s_])
                        for s_ in self.states) for a in self.actions)
            if all(abs(vstar[s] - vstar_prev[s]) < 1e-10 for s in
                self.states):
                break
        return vstar

    def policy_from_vstar(self, vstar):
        policy = {s: max(self.actions, key=lambda a:
            sum(self.transition_prob[s, a][s_] * (self.rewards[s, a, s_]
            + self.discount_factor * vstar[s_]) for s_ in self.states))
            for s in self.states}
        return policy
```

Problem 135. Optimal Policy

Implement an algorithm to find the optimal action-value function $Q_\star(s, a)$ for a given MDP using the Bellman optimality equation provided in the text.

Answer:

```
def find_optimal_q(mdp, discount_factor):
    """
    Find the optimal action-value function Q* for a given MDP using the
        Bellman optimality equation.

    Parameters:
        - mdp: an instance of the MDP class
        - discount_factor: the discount factor of the MDP

    Returns:
        - Q: a dictionary where keys are states and actions and values
            are the corresponding optimal action-value
    """
    Q = {(s,a): 0 for s in mdp.states for a in mdp.actions(s)}

    # Set a stopping criteria, e.g maximum number of iterations
    max_iter = 1000
    for i in range(max_iter):
        Q_prev = Q.copy()
        for s in mdp.states:
            for a in mdp.actions(s):
                Q[s,a] = mdp.expected_reward(s,a) + discount_factor *
                    max(mdp.expected_next_state_rewards(s,a,Q_prev))

    return Q
```

Problem 136. Planning by Dynamic Programming with a Known MDP

Given the MDP represented by the tuple `(S, A, T, R, gamma)`, where `S` is the set of states, `A` is the set of actions, $T(s,a,s') = P(s'|s,a)$ is the state transition function, $R(s,a)$ is the reward function and `gamma` is the discount factor, and an initial estimate of the action-value function $Q(s,a)$, implement the Q-Learning algorithm to compute the optimal action-value function $Q_\star(s,a)$.

Answer:

```
def q_learning(S, A, T, R, gamma, Q, n_iter):
    for i in range(n_iter):
        for s in S:
            for a in A:
                Q[s,a] = R[s,a] + gamma * sum(T[s,a,s_] * max(Q[s_,a_] for
                    a_ in A) for s_ in S)
    return Q
```

Problem 137. Reinforcement Learning

- Create a function `mean_update(prev_mean, new_sample, sample_count)` that calculates the next mean given the previous mean `prev_mean`, a new sample `new_sample`, and the current sample count `sample_count`.
- Create a function `monte_carlo_mean(samples)` that takes in a list of `samples` and returns the final mean using the `mean_update` function and the incremental computation of the mean as shown in the equation provided in the text.

Answer:

```python
def mean_update(prev_mean, new_sample, sample_count):
    return prev_mean + (1 / sample_count) * (new_sample - prev_mean)

def monte_carlo_mean(samples):
    sample_count = 0
    mean = 0
    for sample in samples:
        sample_count += 1
        mean = mean_update(mean, sample, sample_count)
    return mean

samples = [1, 2, 3, 4, 5]
print(monte_carlo_mean(samples)) # Output: 3.0
```

Problem 138. Maximum Entropy Reinforcement Learning

- Implement a function called `maximum_entropy_policy(states, actions, rewards, policy)` that takes in four parameters:
  - `states`: a list of states
  - `actions`: a list of actions
  - `rewards`: a list of rewards
  - `policy`: a probability distribution over actions given `states`

  The function should return the optimal policy that maximizes the return and conditional action entropy.
- Implement a function called `maximum_minimum_entropy_policy(states, actions, rewards, policy)` that takes in the same four parameters as the previous function. This function should return the optimal policy that maximizes return while visiting `states` with low entropy and maximizing their entropy for improving exploration.

Answer:

```
import numpy as np

def maximum_entropy_policy(states, actions, rewards, policy):
    returns = np.sum(rewards)
    action_entropy = -np.sum(policy * np.log(policy))
    return returns + action_entropy

def maximum_minimum_entropy_policy(states, actions, rewards, policy):
    returns = np.sum(rewards)
    action_entropy = -np.sum(policy * np.log(policy))
    state_entropy = -np.sum(policy * np.log(policy))
    return returns + action_entropy - state_entropy
```

Problem 139. Function Approximation

- Implement a neural network with a single hidden layer of 8 neurons and an output layer of 1 neuron to approximate the value function $V_\theta(s)$. The input layer should have the same number of neurons as the state space.
- Define a function `calculate_mse(value_function, value_function_approximation)` that takes in the true value function $V_\pi(s)$ `value_function` and the neural network approximation $V_\theta(s)$ `value_function_approximation` and returns the mean squared error (MSE) as defined in the equation provided in the text.
- Define a function `train(states, value_function, policy, learning_rate)` that takes in a list of `states`, the true value function $V_\pi(s)$ `value_function`, a `policy` π, and a learning rate `learning_rate`. The function should use the `policy` to compute the state visitation frequencies $\mu(s)$ and use them to compute the MSE using the `calculate_mse` function. It should then use gradient descent to update the neural network parameters θ to minimize the MSE.

Answer:

```
import numpy as np
import tensorflow as tf

# Neural network implementation
model = tf.keras.Sequential([
    tf.keras.layers.Dense(8, input_shape=(state_space_size,),
        activation='relu'),
    tf.keras.layers.Dense(1)
])

# MSE calculation function
def calculate_mse(value_function, value_function_approximation):
    return np.mean((value_function - value_function_approximation)**2)

# Training function
def train(states, value_function, policy, learning_rate):
    # Compute state visitation frequencies
    state_visitation_frequencies = np.array([policy(state) for state in
        states])
    state_visitation_frequencies /= np.sum(state_visitation_frequencies)

    with tf.GradientTape() as tape:
        # Compute MSE
        value_function_approximation = model(states)
        mse = tf.reduce_mean(tf.square(value_function -
            value_function_approximation))

    # Compute gradients
    gradients = tape.gradient(mse, model.trainable_variables)

    # Update model parameters
    optimizer = tf.optimizers.Adam(learning_rate)
```

```
optimizer.apply_gradients(zip(gradients, model.trainable_variables))
```

Problem 140. Value-Based Methods

Implement a simple experience replay buffer for a Q-Learning agent. The replay buffer should store the `state`, `action`, `reward`, and next state `next_state` for each timestep. It should also have a method for randomly sampling a batch of experience tuples for use in updating the Q-values.

Answer:

```
import random

class ReplayBuffer:
    def __init__(self, buffer_size):
        self.buffer_size = buffer_size
        self.buffer = []

    def add(self, state, action, reward, next_state):
        experience = (state, action, reward, next_state)
        if len(self.buffer) >= self.buffer_size:
            self.buffer.pop(0)
        self.buffer.append(experience)

    def sample(self, batch_size):
        return random.sample(self.buffer, batch_size)

replay_buffer = ReplayBuffer(buffer_size = 100)

# Add experiences to replay buffer
replay_buffer.add(state, action, reward, next_state)

# Sample a batch of experiences for training
batch = replay_buffer.sample(batch_size = 32)
```

Problem 141. Value-Based Methods

- Create a class `QNetwork` that implements a simple neural network with two fully connected layers, where the first layer has 32 units and the second layer has the number of actions as the number of units.
- Create a function `train_q_network` that takes as input the QNetwork `q_network`, a dataset of `experiences` (s, a, r, s') and the number of training iterations `num_iterations`. The function should update the QNetwork parameters using the Q-learning algorithm described in the text.
- Create a function `evaluate_q_network` that takes as input the QNetwork `q_network` and a dataset of `experiences` (s, a, r, s') and returns the mean squared error between the QNetwork predictions and the $Q*$ values.

Answer:

```python
import torch
import torch.nn as nn
import torch.optim as optim

class QNetwork(nn.Module):
    def __init__(self, num_actions):
        super(QNetwork, self).__init__()
        self.fc1 = nn.Linear(in_features=num_features, out_features=32)
        self.fc2 = nn.Linear(in_features=32, out_features=num_actions)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

def train_q_network(q_network, experiences, num_iterations, alpha,
     gamma):
    optimizer = optim.Adam(q_network.parameters())
    for i in range(num_iterations):
        s, a, r, s_prime = experiences[i % len(experiences)]
        q_star = r + gamma * torch.max(q_network(s_prime))
        loss = (q_star - q_network(s)[a]) ** 2
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

def evaluate_q_network(q_network, experiences):
    mse = 0
    for s, a, r, s_prime in experiences:
        q_star = r + gamma * torch.max(q_network(s_prime))
        mse += (q_star - q_network(s)[a]) ** 2
    return mse / len(experiences)
```

Problem 142. Policy-Based Methods

- Create a function called `policy_gradient_ascent` that takes in a policy function `pi(s, theta)` and the environment `env`.
- Initialize the policy parameters `theta` and the learning rate `alpha`.
- Create a for loop to run for a specified number of `episodes`.
- In each episode, generate a trajectory `tau` by following the policy `pi(s, theta)` and interacting with the environment `env`.
- Compute the return of the trajectory `g_tau` using the formula $\sum_t \gamma^t r_t$.
- Compute the gradient of the policy `nabla_theta` $J(pi_\theta)$ using the formula $\nabla_\theta J(\pi_\theta) = \mathbb{E}\tau \sim \pi\theta \left[\sum_t \nabla_\theta \log \pi_\theta(a_t|s_t) g(\tau)\right]$.
- Update the policy parameters `theta` by adding the product of the gradient and the learning rate `alpha` to `theta`.
- Return the updated policy parameters `theta`.

Answer:

```
def policy_gradient_ascent(pi, env, theta, alpha, episodes):
    for episode in range(episodes):
        tau = []
        s = env.reset()
        done = False
        while not done:
            a = pi(s, theta)
            s_, r, done, _ = env.step(a)
            tau.append((s, a, r))
            s = s_
        g_tau = sum([r*(gamma**t) for t, (_, _, r) in enumerate(tau)])
        nabla_theta = sum([nabla_theta_log_pi(s, a, theta)*g_tau for s,
             a, _ in tau])
        theta += alpha*nabla_theta
    return theta
```

Problem 143. Policy-Based Methods

- Create a function `expected_return(theta, num_samples)` that takes in a parameter `theta` and a number of samples `num_samples` and returns the expected return of the trajectory using the equations provided in the text. The function should use the policy parameterized by `theta` to generate `num_samples` number of trajectories and calculate the expected return.
- Create a function `policy_gradient(theta, num_samples)` that takes in a parameter `theta` and a number of samples `num_samples` and returns the gradient of the expected return with respect to the parameters `theta`. The function should use the `expected_return` function to calculate the expected return and its gradient using finite differences.

Answer:

```python
import numpy as np

def expected_return(theta, num_samples):
    """
    Calculates the expected return of a trajectory using the provided
        equations.
    """
    # sample num_samples number of trajectories
    samples = []
    for _ in range(num_samples):
        sample = []
        s = initial_state() # function to generate initial state
        for t in range(num_steps):
            a = pi_theta(s, theta) # function to generate action given
                state and theta
            r = reward(s, a) # function to calculate reward given state
                and action
            sample.append((s, a, r))
            s = next_state(s, a) # function to generate next state given
                current state and action
        samples.append(sample)

    # calculate expected return
    expected_return = 0
    for sample in samples:
        g = 0
        for t, (s, a, r) in enumerate(sample):
            g += (gamma ** t) * r
        expected_return += p(sample, theta) * g
    expected_return /= num_samples
    return expected_return

def policy_gradient(theta, num_samples):
    """
    Calculates the gradient of the expected return with respect to the
        parameters theta.
    """
    grad = np.zeros(theta.shape)
    eps = 1e-4
```

```python
for i in range(len(theta)):
    theta_plus = theta.copy()
    theta_plus[i] += eps
    theta_minus = theta.copy()
    theta_minus[i] -= eps
    grad[i] = (expected_return(theta_plus, num_samples) -
        expected_return(theta_minus, num_samples)) / (2 * eps)
return grad
```

Problem 144. Actor–Critic Methods

- Create a class called `ActorCritic` that takes in two neural networks, one for the actor `actor_net` and one for the critic `critic_net`. The actor network should take in a `state` and output action probabilities, while the critic network should take in a `state` and output a value.
- Implement the `learn` method that takes in a `state`, an `action`, a `reward`, and the next state `next_state`. This method should update the actor and critic networks using the actor-critic algorithm.
- Implement the `act` method that takes in a `state` and returns an action according to the current policy of the actor network.

Answer:

```
import torch
import torch.nn as nn

class ActorCritic(nn.Module):
    def __init__(self, actor_net, critic_net):
        super(ActorCritic, self).__init__()
        self.actor_net = actor_net
        self.critic_net = critic_net

    def learn(self, state, action, reward, next_state):
        # compute the target value
        target_value = reward + discount_factor *
            self.critic_net(next_state)
        # compute the current value
        current_value = self.critic_net(state)
        # compute the value loss
        value_loss = nn.MSELoss()(current_value, target_value)
        # compute the actor loss
        actor_loss = -self.actor_net(state).log_prob(action) *
            (target_value - current_value)
        # update the actor and critic networks
        self.actor_net.optimizer.zero_grad()
        actor_loss.backward()
        self.actor_net.optimizer.step()
        self.critic_net.optimizer.zero_grad()
        value_loss.backward()
        self.critic_net.optimizer.step()

    def act(self, state):
        return self.actor_net(state).sample()
```

Problem 145. Model-Based Reinforcement Learning

- Implement the MCTS algorithm in Python.
- The algorithm should take in the following inputs: start state `start_state`, action value function `Q`, exploration constant `c`, number of simulations `num_simulations`, and learning rate `alpha`.
- The algorithm should return the updated policy parameters.

Answer:

```
import numpy as np

def MCTS(start_state, Q, c, num_simulations, alpha):
    N_s = np.zeros(shape=(1,))
    N_sa = np.zeros(shape=(1,))
    for sim in range(num_simulations):
        # Sample trajectory following pi
        pass
    theta = theta + alpha * np.gradient(J(pi_theta))
    return theta
```

Problem 146. Model-Based Reinforcement Learning

Write a function `mcts_search` that implements AlphaZero's Monte Carlo Tree Search using the UCB update rule of the action value function.

```
def mcts_search(s, c, Q, N, P, V):
    """
    Monte Carlo Tree Search using UCB update rule

    Parameters:
        s (int): the current state
        c (float): the constant that determines the amount of exploration
        Q (dict): a dictionary mapping (s, a) tuples to expected rewards
        N (dict): a dictionary mapping (s, a) tuples to the number of
            times action a was taken from state s
        P (dict): a dictionary mapping (s, a) tuples to the estimate of
            the neural network for the probability of taking action a
            from state s
        V (dict): a dictionary mapping states to the estimate of the
            neural network for the value of state s

    Returns:
        int: the best action for the current state
    """
    pass
```

Answer:

```
import math

def mcts_search(s, c, Q, N, P, V):
    """
    Monte Carlo Tree Search using UCB update rule

    Parameters:
        s (int): the current state
        c (float): the constant that determines the amount of exploration
        Q (dict): a dictionary mapping (s, a) tuples to expected rewards
        N (dict): a dictionary mapping (s, a) tuples to the number of
            times action a was taken from state s
        P (dict): a dictionary mapping (s, a) tuples to the estimate of
            the neural network for the probability of taking action a
            from state s
        V (dict): a dictionary mapping states to the estimate of the
            neural network for the value of state s

    Returns:
        int: the best action for the current state
    """
    actions = []
    for a in range(len(P[s])):
        if (s, a) in N:
            ucb = Q[s, a] + c * P[s][a] * math.sqrt(N[s]) / (1 + N[s, a])
        else:
            ucb = float('inf')
        actions.append((ucb, a))
    _, best_action = max(actions)
    return best_action
```

Problem 147. Imitation Learning

Write a simple implementation of behavioral cloning using supervised learning in Python. The implementation should take a set of expert demonstrations as input and use them to train a neural network that outputs a policy $\pi_\theta(a|s)$.

Answer:

```
import numpy as np
import tensorflow as tf

# Define the neural network architecture
model = tf.keras.Sequential([
    tf.keras.layers.Dense(64, activation='relu',
        input_shape=(input_dim,)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(output_dim, activation='softmax')
])

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer='adam',
    metrics=['accuracy'])

# Train the model on expert demonstrations
history = model.fit(X, y, batch_size=32, epochs=10, validation_split=0.2)
```

Problem 148. Imitation Learning

Implement a function that takes as input a set of expert demonstrations $\mathcal{D}$ and returns the best parameters $\theta_\star$ that maximize the log-likelihood.

```
def inverse_reinforcement_learning(expert_demonstrations):
    # Your implementation here
    return theta_star
```

Answer:

```
import numpy as np
from scipy.optimize import minimize

def reward_function(theta, expert_demonstrations):
    reward = 0
    for demonstration in expert_demonstrations:
        reward += np.exp(np.dot(theta, demonstration))
    return -np.log(reward)

def inverse_reinforcement_learning(expert_demonstrations):
    theta_0 = np.zeros(len(expert_demonstrations[0]))
    theta_star = minimize(reward_function, theta_0,
        args=(expert_demonstrations,), method='BFGS').x
    return theta_star
```

Problem 149. Exploration

Implement the modified transition function and reward as described in the text.

```python
def modified_transition(s, a, s_prime, N_sa, N_sas, S, k, R_max):
    pass

def modified_reward(s, a, N_sa, r_sa, R_max):
    pass
```

Input:

- s: A tuple representing the current state.
- a: A tuple representing the current action.
- s_prime: A tuple representing the next state.
- N_sa: A dictionary where the keys are (s, a) tuples and the values are integers representing the count of times the action a has been taken in state s.
- N_sas: A dictionary where the keys are (s, a, s') tuples and the values are integers representing the count of times the transition (s, a, s') has occurred.
- S: A list of tuples representing all the states in state space.
- k: An integer representing the threshold for high exploration.
- R_max: A float representing the maximum reward.

Output:

- A tuple of the modified next state s_prime and the modified reward R.

Example:

```python
>>> N_sa = {((0, 0), (0, 1)): 5, ((0, 0), (1, 0)): 4}
>>> N_sas = {((0, 0), (0, 1), (0, 1)): 5, ((0, 0), (1, 0), (1, 0)): 4}
>>> S = [(0, 0), (0, 1), (1, 0), (1, 1)]
>>> k = 5
>>> R_max = 1.0
>>> modified_transition((0, 0), (0, 1), (0, 1), N_sa, N_sas, S, k, R_max)
((0, 1), 0.5)
>>> modified_reward((0, 0), (0, 1), N_sa, N_sas, R_max)
0.5
```

Answer:

```python
def modified_transition(s, a, s_prime, N_sa, N_sas, S, k, R_max):
    if N_sa[(s, a)] < k:
        return (s, R_max)
    else:
        return (s_prime, (N_sas[(s, a, s_prime)] + 1) / (N_sa[(s, a)] +
            len(S)))

def modified_reward(s, a, N_sa, r_sa, R_max):
    if N_sa[(s, a)] < k:
        return R_max
    else:
        return sum(r_sa[(s, a)]) / N_sa[(s, a)]
```