Algorithm for Hopfield Network Simulation

1. Initialize the Environment
   1.1 Import necessary modules.
   1.2 Set up inline plotting for visualizations.

2. Define Network Parameters
   2.1 Set the pattern size (`pattern_size = 5`).
   2.2 Create a Hopfield Network with `pattern_size^2` neurons.

3. Generate Patterns
   3.1 Instantiate a pattern factory.
   3.2 Generate a checkerboard pattern.
   3.3 Store the checkerboard pattern in a list.
   3.4 Generate and add three random patterns (each pixel has a 50% probability of being ON).

4. Visualize Patterns
   4.1 Plot the generated patterns.

5. Analyze Pattern Overlap
   5.1 Compute the overlap matrix between all stored patterns.
   5.2 Plot the overlap matrix.

6. Train the Hopfield Network
   6.1 Store the generated patterns in the Hopfield Network.

7. Introduce Noise to a Pattern
   7.1 Create a noisy version of the checkerboard pattern by flipping 4 random bits.
   7.2 Set the Hopfield Network's initial state to this noisy pattern.

8. Run the Network Dynamics
   8.1 Execute the Hopfield Network for 4 iterations to recall the pattern.
   8.2 Monitor the network's state evolution over time.
   8.3 Reshape state vectors into a 2D pattern format.

9. Visualize Network Recovery Process
   9.1 Plot the evolution of the network's states over time.
   9.2 Display the overlap between the network's evolving state and the stored patterns.

10. End of Algorithm

Algorithm for Training a Word Embedding Model Using Skip-Grams in TensorFlow

---

1. Define Training and Testing Corpus

1.1 Create a **training corpus** with multiple sentences.
1.2 Create a **testing corpus** with similar sentence structures.

---

2. Define Parameters

2.1 Set `embedding_dim` (size of word vectors).
2.2 Set `window_size` (context window for skip-grams).
2.3 Initialize `vocab_size` (will be computed later).

---

3. Preprocess Corpus

Input: List of sentences (corpus).
Process:
3.1 Initialize a tokenizer to convert words into numerical indices.
3.2 Fit the tokenizer on the corpus to create a vocabulary.
3.3 Convert each sentence into a sequence of word indices.
3.4 Create a **word-to-index** dictionary.
3.5 Create an **index-to-word** dictionary.
3.6 Compute **vocab_size** (total number of unique words +1 for padding).
Output: Processed sequences, word-to-index mapping, index-to-word mapping.

---

4. Generate Skip-Grams

Input: Sequences of word indices, vocab_size, window_size.
Process:
4.1 Iterate through each sequence in the dataset.
4.2 Use the **skip-grams function** to generate word pairs and labels.
4.3 Store all generated skip-gram pairs.
Output: Skip-gram word pairs with corresponding labels.

---

5. Preprocess Training Data

5.1 Apply the preprocessing function to the combined training and testing corpus.
5.2 Generate skip-grams for the processed sequences.

---

6. Define the Model Architecture

Input:
6.1 Define two input layers (`input_target` and `input_context`).
6.2 Create an **embedding layer** to convert words into dense vectors.
6.3 Extract embeddings for both the target word and the context word.
6.4 Compute the **dot product** of both embeddings.
6.5 Flatten the output.
6.6 Pass it through a **dense layer with a sigmoid activation** to classify whether the word pair is contextually related.
Output: A compiled **Keras model** with Adam optimizer and binary cross-entropy loss.

---

7. Prepare Data for Training

Input: Skip-gram pairs from Step 5.
7.1 Extract target words, context words, and labels from skip-gram pairs.
7.2 Convert them into NumPy arrays for model training.

---

8. Train the Model

Input: Training data (target words, context words, labels).
8.1 Train the model using **100 epochs** and **batch size = 128**.

---

9. Testing the Model

Input: Testing corpus.
Process:
9.1 Preprocess the testing corpus.
9.2 Generate skip-grams for test data.

9.3 Iterate over each skip-gram pair:
    9.3.1 Predict the relationship score using the trained model.
    9.3.2 Convert prediction to binary classification (threshold = 0.5).
    9.3.3 Compare predicted label with actual label.
9.4 Compute the **testing accuracy** based on correct predictions.

---

10. Display Results

10.1 Print the final **vocabulary size**.
10.2 Print the **testing accuracy** in percentage format.

Algorithm for BSB (Brain-State-in-a-Box) Network with Hebbian Learning

---

1. Initialize Stored Patterns

1.1 Define stored patterns as vectors of length 5.
1.2 Store multiple patterns in a matrix.

---

2. Compute Hebbian Weight Matrix

Input: Stored patterns.
Process:
2.1 Compute the weight matrix **W** using the outer product rule:
$W = P^T \times P$, where **P** is the pattern matrix.
2.2 Convert **W** to float type for numerical stability.
2.3 Subtract a scaled identity matrix to ensure stability:
$W = W - I \times \max(W)$, where **I** is the identity matrix.

Output: Stabilized weight matrix **W**.

---

3. Define the BSB Update Function

Input: Initial state vector **x**, weight matrix **W**, number of iterations.
Process:
3.1 Define **tanh activation** function:
Apply non-linearity: **$x' = \tanh(W \times x)$**.
3.2 Define **energy computation** function:
Calculate network energy: **$E = -0.5 \times x^T \times W \times x$**.
3.3 Initialize **state list** with the input vector.
3.4 Compute and store initial energy.
3.5 Iterate **num_iterations** times:
3.5.1 Update **x** using the tanh activation function.
3.5.2 Store updated **x**.
3.5.3 Compute and store new energy.

Output: Sequence of updated states and energy values over time.

---

4. Define Test Input

4.1 Create a slightly **noisy version** of a stored pattern.
4.2 Use this vector as an initial input for BSB processing.

---

5. Perform BSB Iterations

Input: Test input vector, weight matrix **W**.
Process:
5.1 Apply **BSB update function** on the input vector.
5.2 Store the evolving **states** and **energy** at each iteration.

Output:

- Matrix of evolving states per iteration.
- List of computed energy values.

---

6. Convert Results to DataFrames for Visualization

Input: BSB output states and energy values.
6.1 Convert state evolution results into a **DataFrame** with labeled columns.
6.2 Convert energy values into a **DataFrame** for easy analysis.

---

7. Display Computed Results

7.1 Print stored patterns from **Hebbian learning**.
7.2 Print the **BSB state evolution** table.
7.3 Print **energy evolution** values.

---

8. Plot State Evolution Over Time

Input: DataFrame of state values over iterations.
Process:
8.1 Iterate through each node (feature) and plot its trajectory over time.
8.2 Label the axes and title the plot.
8.3 Show the plot with a grid and legend.

---

9. Plot Energy Evolution Over Time

Input: DataFrame of energy values over iterations.
Process:
9.1 Plot energy as a function of iteration steps.
9.2 Label the axes and title the plot.
9.3 Show the plot with gridlines for clarity.

Neural Network Training using Simulated Annealing on the IRIS Dataset

1. Introduction

This document describes the implementation of a **feedforward neural network** trained using **Simulated Annealing (SA)** for classification on the **IRIS dataset**. The model consists of an **input layer, one hidden layer, and an output layer**. The goal is to optimize the network's weights using **SA** instead of traditional gradient-based methods.

---

2. Dataset and Preprocessing

2.1 IRIS Dataset

- The **IRIS dataset** is a well-known dataset for classification problems.
- It consists of **150 samples** belonging to **3 different classes** (Setosa, Versicolor, and Virginica).
- Each sample has **4 features**:
    1. Sepal length
    2. Sepal width
    3. Petal length
    4. Petal width

2.2 Data Preprocessing

1. **One-Hot Encoding**
    - Since the labels are categorical (0, 1, 2), we apply **One-Hot Encoding** to convert them into binary vectors.
    - Example: Class 0 → `[1, 0, 0]`, Class 1 → `[0, 1, 0]`, Class 2 → `[0, 0, 1]`.
2. **Feature Normalization**
    - We apply **Standardization (Z-score normalization)** to scale the features.
    - Formula: **X_scaled = (X - mean) / standard deviation**
3. **Dataset Splitting**
    - Split the dataset into **80% training data** and **20% testing data**.

---

3. Neural Network Architecture

3.1 Layer Structure

- **Input Layer**: 4 neurons (corresponding to the 4 features of IRIS data)
- **Hidden Layer**: 7 neurons (fully connected, with Sigmoid activation)
- **Output Layer**: 3 neurons (corresponding to the 3 classes, using Softmax activation)

3.2 Weight and Bias Initialization

- Weights and biases are **randomly initialized** using a normal distribution.

---

4. Activation Functions

4.1 Sigmoid Activation Function *(Used in Hidden Layer)*

- Formula: σ(x)=11+e−x\sigma(x) = \frac{1}{1 + e^{-x}}
- Maps values into a range of (0,1) and introduces non-linearity.

4.2 Softmax Activation Function *(Used in Output Layer)*

- Formula: softmax(xi)=exi∑exj\text{softmax}(x_i) = \frac{e^{x_i}}{\sum e^{x_j}}
- Converts output scores into a probability distribution.

---

5. Forward Propagation

Inputs: Feature matrix `X`, weights `w1, w2`, biases `b1, b2`
Steps:

1. Compute hidden layer activations: H=σ(X×W1+b1)H = \sigma(X \times W_1 + b_1)
2. Compute output layer activations: O=softmax(H×W2+b2)O = \text{softmax}(H \times W_2 + b_2)
3. Return hidden layer and output predictions.

Outputs: Hidden layer activation `H`, Output probabilities `0`.

---

6. Cross-Entropy Loss Function

Formula: L=−1N∑(Ytrue×log(Ypred+10−8))L = -\frac{1}{N} \sum (Y_{true} \times \log(Y_{pred} + 10^{-8})) Where:

- YtrueY_{true} is the actual label.
- YpredY_{pred} is the predicted probability.
- 10−810^{-8} prevents log(0) errors.

---

7. Simulated Annealing for Weight Optimization

7.1 Key Concepts

- **Objective:** Optimize neural network weights using Simulated Annealing (SA).
- **Metropolis Criterion:** Accept new weight configuration if it reduces the loss or with a probability e(Lold−Lnew)/Te^{(L_{old} - L_{new})/T}.
- **Cooling Schedule:** Temperature decreases after each iteration.

7.2 Steps in Simulated Annealing

1. **Initialize temperature (`T_initial`) and cooling rate (`cooling_rate`).**
2. **Compute initial loss using forward propagation.**
3. **Repeat for `max_iter` iterations:**
    - **Generate new candidate weights and biases** with small perturbations.
    - **Perform forward propagation** with new weights.
    - **Compute new loss**.
    - **Accept new weights** if:
        - New loss is lower.
        - Acceptance probability e(Lold−Lnew)/Te^{(L_{old} - L_{new})/T} is greater than a random value.
    - **Reduce temperature**.
4. **Stop when temperature is near zero.**

Outputs: Optimized `w1, b1, w2, b2`.

---

8. Model Evaluation

8.1 Testing the Model

1. Perform **forward propagation** on `X_test` using the optimized weights.
2. Convert **softmax outputs** to class predictions (selecting the highest probability).
3. Compare predicted labels with true labels (`Y_test`).

8.2 Compute Classification Accuracy

Accuracy=Correct PredictionsTotal Predictions×100Accuracy = \frac{\text{Correct Predictions}}{\text{Total Predictions}} \times 100

---

9. Results and Performance

- **Final Classification Accuracy:** Displayed after training.
- Simulated Annealing successfully optimizes weights, demonstrating an alternative to traditional optimization methods.

---

10. Conclusion

This document describes the implementation of a **feedforward neural network trained using Simulated Annealing**. The model is evaluated on the **IRIS dataset**, achieving high classification accuracy. The results demonstrate that **Simulated Annealing can be an effective alternative optimization technique for neural networks**.