Building an Unsupervised NLP CBOW Word2Vec Model from Scratch in C/C++: A Step-by-Step Implementation, by Q@spysee.pk

1. Initialize the NN: Create a neural network and initialize it with random weights and biases.

```
/*
   This macro is used to define the size of the window of context words around a target word.
   The CBOW model aims to predict the target word based on its surrounding context words.
   The macro "REPLIKA_WINDOW_SIZE" is set to 2, the context window for each target word will include the two words t
o the left and the two words to the right of the target word.
 */
#define REPLIKA_WINDOW_SIZE  2
/*
    Number of neurons in the hidden layer and this represents the size of the hidden layer in the neural network.
    10 neurons is small size, suitable for small vocabulary.
    However, for larger vocabularies and more complex tasks, a larger hidden layer size may be required to capture m
ore intricate relationships between the input and output.
    The choice of hidden layer size is a hyperparameter(that requires tuning and experimentation). Hyperparameters a
re set by the data scientist or the model developer before training the model. These settings determine the behavior
of the learning algorithm and influence how the model learns from the data.
    Increasing the hidden layer size could potentially lead to better model performance, but it may also increase th
e computational cost and training time. The goal is to strike a balance between underfitting and overfitting by find
ing hyperparameters that allow the model to generalize well to unseen data.
    Another example of hyperparameter is "learning rate".
 */
#define REPLIKA_HIDDEN_SIZE 10
```

1. Initialize the NN: Create a neural network and initialize it with random weights and biases(slide 2).

```
 /*
    The preprocessor macro that processes the corpus and generates the training data in the form of context words and target words.
    It creates a list of context words (x_train_list) and a list of target words (y_train_list). x prefix in all names means input words(context words) and y prefix in all names means output words(the target words).
  */
 CORPUS_TO_CONTEXTS_AND_TARTGETS(data_parser, vocab, x_train_list, y_train_list, x_train_dim, y_train_dim, REPLIKA_WINDOW_SIZE, false);

 /*
   Convert the lists into 2D arrays (x_train and y_train) with the appropriate dimensions for training the CBOW model. These arrays will be used during the training process to predict the target word given the context words.
   x_train represents the input data to the CBOW model, consisting of context words(one-hot encoded input words).
   y_train represents the target words.
  */
 unsigned int (*x_train)[REPLIKA_CONTEXT_LINE_SIZE] = reinterpret_cast<unsigned int(*)[REPLIKA_CONTEXT_LINE_SIZE]>(numc_obj.ndarray(x_train_list, x_train_dim));

 unsigned int (*y_train)[REPLIKA_TARGET_LINE_SIZE] = reinterpret_cast<unsigned int(*)[REPLIKA_TARGET_LINE_SIZE]>(numc_obj.ndarray(y_train_list, y_train_dim));
```

1. Initialize the NN: Create a neural network and initialize it with random weights and biases(slide 3).

```
/*
    This code segment is responsible for initializing the weight matrices W1 and W2 for a simple neural network that has only one hidden layer. These weight matrices are used to learn word embeddings through training.
    Word-embedings, W1 and W2 represent the word-embeding vectors.
    W1: This matrix has dimensions len(vocab) x hidden_size, where len(vocab) is the length of the vocabulary (number of unique words in the corpus), and hidden_size is the size of the hidden layer. Each row in W1 represents the dense vector embedding for a specific word in the vocabulary. It connects the input layer (one-hot vectors representing context words) to the hidden layer.
    W2: This matrix has dimensions hidden_size x len(vocab). It connects the hidden layer to the output layer, which represents the predicted one-hot vector for the target word. Each column in W2 corresponds to a specific word in the vocabulary, and the values in the column represent the weights connecting the hidden layer neurons to the output layer neurons for that specific word.
 */
  double (*W1)[REPLIKA_HIDDEN_SIZE] = reinterpret_cast<double (*)[REPLIKA_HIDDEN_SIZE]>(numc_obj.RANDN(/*vocab.len()*/ REPLIKA_VOCABULARY_LENGTH , REPLIKA_HIDDEN_SIZE));

  double (*W2)[REPLIKA_VOCABULARY_LENGTH] = reinterpret_cast<double(*)[REPLIKA_VOCABULARY_LENGTH]>(numc_obj.RANDN(REPLIKA_HIDDEN_SIZE, /*vocab.len()*/ REPLIKA_VOCABULARY_LENGTH));
```

1. Initialize the NN: Create a neural network and initialize it with random weights and biases(final slide).

This part of the code is performing the necessary steps to set up the neural network for training the Continuous Bag of Words (CBOW) model. The code accomplishes the following:

- Defines and initializes the weight matrices W_1 and W_2 with random values. These weight matrices are essential components of the neural network and represent the connections between the input, hidden, and output layers.
- Prepares the training data by creating the x_train and y_train arrays. x_train represents the input data to the CBOW model, consisting of context words (one-hot encoded input words), and y_train represents the target words.
- Sets the hidden layer size (REPLIKA_HIDDEN_SIZE) and the window size (REPLIKA_WINDOW_SIZE) to control the number of neurons in the hidden layer and the size of the context window around the target word.

Initializing the neural network with random weights and biases is the first step of the training process. After this step, you would proceed with the forward propagation, loss computation, backpropagation, and weight updates to train the CBOW model to learn word embeddings and improve its performance on your specific NLP task.

2. Feedforward pass: NN predicts an outcome based on input. For example: does the input image have a cat?

The code block represents the feedforward pass of the neural network for the Continuous Bag of Words (CBOW) model.

- forward_propogation: This function is responsible for executing the forward pass of the neural network. It takes as input the context words, weight matrices W_1 and W_2, objects corpus_obj and numc_obj, and it returns the hidden layer representation h and the predicted output vector y_pred.
- w1_subarray: This is a two-dimensional array that represents a subset of the weight matrix W_1 corresponding to the context words. It extracts a portion of W_1 based on the indices of the context words, which are provided as input. This is achieved using the SUBARRAY macro.
- h: The h vector is obtained by taking the mean (average) of the rows of w1_subarray. It represents the hidden layer representation and is used in both the forward and backward passes of the neural network.
- u: The u vector is calculated by performing a dot product between h and W_2. This represents the output before applying the softmax activation function.
- y_pred: The predicted output vector y_pred is obtained by applying the softmax activation function to the u vector. The softmax function converts the raw output scores into probabilities, representing the likelihood of each word in the vocabulary being the target word.

In summary, the forward_propogation function carries out the feedforward pass of the CBOW neural network. It takes the context words, calculates the hidden layer representation h, and then computes the predicted output vector y_pred using the softmax activation function. This forward pass is a crucial step in generating predictions for the CBOW model, and it precedes the backpropagation step, which updates the weights and biases to improve the model's performance during the training process.

2. Feedforward pass: NN predicts an outcome based on input. For example: does the input image have a cat?(final slide)

```
struct forward_propogation<double> forward(unsigned int* context, double (*W1)[REPLIKA_HIDDEN_SIZE], double (*W2)[REPLIKA_VOCABULARY_LENGTH], class corpus& corpus_obj, class numc& numc_obj)
{
    cc_tokenizer::allocator<char> alloc_obj;
    double (*w1_subarray)[REPLIKA_HIDDEN_SIZE];

    SUBARRAY(double, w1_subarray, W1, corpus_obj.len(), REPLIKA_HIDDEN_SIZE, context, corpus_obj.get_size_of_context_line());
    /*
        Returned array is single dimension array, size of returned array for REPLIKA_PK_NUMC_YAXIS is REPLIKA_HIDDEN_SIZE
        Small h is for hidden, h refers to the hidden layer vector obtained by averaging the embeddings of the context words.
        It is used in both the forward and backward passes of the neural network.
        Each context word array has its own h value... h has a shape (1, REPLIKA_HIDDEN_SIZE)
     */
    double* h = numc_obj.mean(reinterpret_cast<double*>(w1_subarray), {REPLIKA_HIDDEN_SIZE, corpus_obj.get_size_of_context_line(), NULL, NULL}, /*REPLIKA_PK_NUMC_MEAN_AXIS::*/REPLIKA_PK_NUMC_YAXIS);

    double* u = numc_obj.dot(h, reinterpret_cast<double*>(W2), {REPLIKA_HIDDEN_SIZE, 1, NULL, NULL}, {corpus_obj.len(), REPLIKA_HIDDEN_SIZE, NULL, NULL}, REPLIKA_PK_NUMC_YAXIS);
    double* y_pred = softmax<double>(u, corpus_obj, numc_obj);

    alloc_obj.deallocate(reinterpret_cast<char*>(u));
    alloc_obj.deallocate(reinterpret_cast<char*>(w1_subarray));

    return {h, y_pred};
}
```

3. Calculate Loss: At first the prediction is a random guess. The error between the prediction and reality is called loss. If the network predicts there is no cat in an image of a cat, loss is high and vice-versa.

```
/* Part of the training loop */
for (cc_tokenizer::string_character_traits<char>::size_type epoch = 0; epoch < default_epoch; epoch++)
{
    double epoch_loss = 0;
    for (cc_tokenizer::string_character_traits<char>::size_type i = 0; i < vocab.get_number_of_context_lines_in_voca
bulary(); i++)
    {
        /* The context words and the center(target word based on the context words) word */
        unsigned int (*context)[REPLIKA_CONTEXT_LINE_SIZE] = x_train + i;
        /*
        y_true refers to the target word for which we are trying to predict the context words. In the CBOW model, we
input a sequence of context words and try to predict the target word. y_true represents the target word in the train
ing data, and we use it to compute the loss and gradients during training. */
        unsigned int y_true = *(y_train[i]);
        /* y_pred is a numc array of predicted probabilities of the output word given the input context. In our impl
ementation, it is the output of the forward propagation step. */
        /* In the context of our CBOW model, h refers to the hidden layer vector obtained by averaging the embedding
s of the context words. It is used in both the forward and backward passes of the neural network. */
        forward_propogation fpg = forward(*context, W1, W2, vocab, numc_obj);
        epoch_loss = epoch_loss + (-1*log(fpg.y_pred[y_true]));
    }
        printf("Epoch %d loss = %.16f\n", epoch, (epoch_loss/vocab.get_number_of_context_lines_in_vocabulary()));
}
```

4. Backpropagation: This is where the magic happens. Fundamentally, you are calculating how much each parameter (weight or bias of each node) in the network contributed to the loss or error from step 2.

You move backwards from the last layer using the chain rule of calculus and compute "gradients". Basically, you are calculating the gradient of the loss function with respect to each weight or bias

The code in the next slide is implementing the backward propagation (backpropagation) step for the Continuous Bag of Words (CBOW) model. This step involves computing the gradients of the model's parameters (weights) with respect to the loss function, which is used to update the parameters during training.
Let's break down the code and its functionalities:

- backward_propagation: This function implements the backpropagation step. It takes as input the context words, the forward propagation values (fpg) obtained from the feedforward pass, the true label y_true of the center word, weight matrices W1 and W2, objects corpus_obj and numc_obj, and returns the gradients grad_W1 and grad_W2.
- hot: This is a one-hot encoded vector representing the true target word. It is used to compute the difference between the predicted output vector y_pred and the true distribution, which is part of the loss calculation.
- grad_u: This vector represents the gradient of the loss function with respect to the output vector u. It is computed as the difference between the predicted output vector y_pred (obtained from the forward pass) and the one-hot encoded target vector hot.
- W2_T: This matrix represents the transpose of weight matrix W2. It is used to compute the gradient grad_h by performing a dot product with grad_u.
- grad_h: This vector represents the gradient of the loss function with respect to the hidden layer representation h. It is computed by performing a dot product between grad_u and the transpose of W2.
- grad_W2: This matrix represents the gradient of the loss function with respect to weight matrix W2. It is computed by taking the outer product of the hidden layer representation h and grad_u.
- grad_W1: This matrix represents the gradient of the loss function with respect to weight matrix W1. It is computed by first obtaining the outer product of grad_h and a vector of ones (of the same size as the context words), and then updating the appropriate rows of grad_W1 using the SUBARRAY macro.

The backpropagation step allows the model to learn from the errors and update its weights and biases to improve its performance in predicting the target word given the context words. The computed gradients are then used in the optimization step (such as gradient descent) to adjust the weights and biases of the model, moving them in a direction that reduces the loss and enhances the model's accuracy.

```
struct backward_propogation<double> backward(unsigned int* context, struct forward_propogation<double>& fpg, unsigned int y_true, double (*W1)[REPLIKA_HIDDEN_SIZE], double (*W2)[REPLIKA_VOCABULARY_LENGTH], class corpus& corpus_obj, class numc& numc_obj)
{
    cc_tokenizer::allocator<char> alloc_obj;
    double* hot = one_hot<double>(y_true - REPLIKA_PK_INDEX_ORIGINATES_AT, corpus_obj.len(), numc_obj);
    double* grad_u = numc_obj.subtract_matrices<double>(fpg.y_pred, hot, corpus_obj.len());
    double* W2_T = numc_obj.transpose_matrix<double>(reinterpret_cast<double*>(W2), REPLIKA_HIDDEN_SIZE, REPLIKA_VOCABULARY_LENGTH);
    double* grad_h = numc_obj.dot(grad_u, W2_T, {REPLIKA_VOCABULARY_LENGTH, 1, NULL, NULL}, {REPLIKA_HIDDEN_SIZE, REPLIKA_VOCABULARY_LENGTH, NULL, NULL}, REPLIKA_PK_NUMC_YAXIS);
    double* grad_W2 = numc_obj.outer<double>(fpg.h, grad_u, REPLIKA_HIDDEN_SIZE, REPLIKA_VOCABULARY_LENGTH);
    double* grad_W1 = numc_obj.zeros<double>(2, REPLIKA_VOCABULARY_LENGTH, REPLIKA_HIDDEN_SIZE);
    double* ones_in_place_of_context_indexes = numc_obj.ones<double>(1, corpus_obj.get_size_of_context_line());
    double* outer_of_grad_h_and_ones_in_place_of_context_indexes = numc_obj.outer<double>(grad_h, ones_in_place_of_context_indexes, REPLIKA_HIDDEN_SIZE, corpus_obj.get_size_of_context_line());
    double* transpose_of_outer_of_grad_h_and_ones_in_place_of_context_indexes = numc_obj.transpose_matrix<double>(outer_of_grad_h_and_ones_in_place_of_context_indexes, REPLIKA_HIDDEN_SIZE, corpus_obj.get_size_of_context_line());
    double (*grad_W1_subarray)[REPLIKA_HIDDEN_SIZE];
    SUBARRAY(double, grad_W1_subarray, reinterpret_cast<double(*)[REPLIKA_HIDDEN_SIZE]>(grad_W1), corpus_obj.len(), REPLIKA_HIDDEN_SIZE, context, corpus_obj.get_size_of_context_line());
    SUM_OF_TWO_MATRICES_OF_DIFFERENT_NUMBER_OF_ROWS(grad_W1, transpose_of_outer_of_grad_h_and_ones_in_place_of_context_indexes, corpus_obj.len(), REPLIKA_HIDDEN_SIZE, context, corpus_obj.get_size_of_context_line(), double);
    alloc_obj.deallocate(reinterpret_cast<char*>(grad_h));
    alloc_obj.deallocate(reinterpret_cast<char*>(grad_u));
    alloc_obj.deallocate(reinterpret_cast<char*>(grad_W1_subarray));
    alloc_obj.deallocate(reinterpret_cast<char*>(hot));
    alloc_obj.deallocate(reinterpret_cast<char*>(ones_in_place_of_context_indexes));
    alloc_obj.deallocate(reinterpret_cast<char*>(outer_of_grad_h_and_ones_in_place_of_context_indexes));
    return {grad_W1, grad_W2};
}
```

5: Gradient Descent: We now adjust the weights and biases based on the gradients calculated in the last step. Typically this is done by multiplying the gradient by a small factor called learning rate. The basic idea is to reduce the error or loss.

```
double* grad_W1_multiply_by_learning_rate = NULL;
MULTIPLY_ARRAY_BY_SCALAR(bpg.grad_W1, REPLIKA_LEARNING_RATE, REPLIKA_VOCABULARY_LENGTH*REPLIKA_HIDDEN_SIZE, grad_W1_multiply_by_learning_rate, double);
SUBTRACT_ARRAY_FROM_ARRAY(W1, grad_W1_multiply_by_learning_rate, REPLIKA_VOCABULARY_LENGTH*REPLIKA_HIDDEN_SIZE, double);

double* grad_W2_multiply_by_learning_rate = NULL;
MULTIPLY_ARRAY_BY_SCALAR(bpg.grad_W2, REPLIKA_LEARNING_RATE, REPLIKA_VOCABULARY_LENGTH*REPLIKA_HIDDEN_SIZE, grad_W2_multiply_by_learning_rate, double);
SUBTRACT_ARRAY_FROM_ARRAY(W2, grad_W2_multiply_by_learning_rate, REPLIKA_VOCABULARY_LENGTH*REPLIKA_HIDDEN_SIZE, double);
```

5: Gradient Descent: We now adjust the weights and biases based on the gradients calculated in the last step. Typically this is done by multiplying the gradient by a small factor called learning rate. The basic idea is to reduce the error or loss(final slide).

The previous code block is part of training loop and is implementing the Gradient Descent step for adjusting the weights (W_1 and W_2) based on the gradients calculated during the backpropagation step. The process is as follows:

- grad_W1_multiply_by_learning_rate: This step multiplies the gradient grad_W1 with the learning rate. REPLIKA_LEARNING_RATE. The learning rate is a small factor used to control the step size during weight updates, preventing large changes that could lead to instability. The result is stored in the grad_W1_multiply_by_learning_rate array.
- SUBTRACT_ARRAY_FROM_ARRAY: This step subtracts the grad_W1_multiply_by_learning_rate array from the W_1 array. This is the actual weight update step. By subtracting the product of the gradient and the learning rate from the current weights, the weights are adjusted in the direction that reduces the loss and helps the model to converge towards a better solution.
- grad_W2_multiply_by_learning_rate: Similarly, this step multiplies the gradient grad_W2 with the learning rate REPLIKA_LEARNING_RATE and stores the result in the grad_W2_multiply_by_learning_rate array.
- SUBTRACT_ARRAY_FROM_ARRAY: This step subtracts the grad_W2_multiply_by_learning_rate array from the W_2 array, updating the weights for the second layer based on the gradients and learning rate.

In summary, the code demonstrates the implementation of the Gradient Descent algorithm to update the weights of the neural network based on the gradients calculated during the backpropagation step. This process helps the model iteratively adjust its parameters, gradually reducing the loss and improving its ability to predict the target word given the context words. The learning rate plays a crucial role in determining the step size of these updates and affects how quickly the model converges to an optimal solution.

6. Iteration: We repeat steps 2 to 6 for all the data points in your training set multiple times (epochs). Hence, your NN is likely to be a better fit, if you have more training data points.

```
for (cc_tokenizer::string_character_traits<char>::size_type epoch = 0; epoch < default_epoch; epoch++)
{
    double epoch_loss = 0;
    for (cc_tokenizer::string_character_traits<char>::size_type i = 0; i < vocab.get_number_of_context_lines_in_voca
bulary(); i++)
    {
        unsigned int (*context)[REPLIKA_CONTEXT_LINE_SIZE] = x_train + i;
        unsigned int y_true = *(y_train[i]);
        forward_propogation fpg = forward(*context, W1, W2, vocab, numc_obj);
        backward_propogation bpg = backward(*context, fpg, y_true, W1, W2, vocab, numc_obj);
        double* grad_W1_multiply_by_learning_rate = NULL;
        MULTIPLY_ARRAY_BY_SCALAR(bpg.grad_W1, REPLIKA_LEARNING_RATE, REPLIKA_VOCABULARY_LENGTH*REPLIKA_HIDDEN_SIZE,
grad_W1_multiply_by_learning_rate, double);
        SUBTRACT_ARRAY_FROM_ARRAY(W1, grad_W1_multiply_by_learning_rate, REPLIKA_VOCABULARY_LENGTH*REPLIKA_HIDDEN_SI
ZE, double);
        double* grad_W2_multiply_by_learning_rate = NULL;
        MULTIPLY_ARRAY_BY_SCALAR(bpg.grad_W2, REPLIKA_LEARNING_RATE, REPLIKA_VOCABULARY_LENGTH*REPLIKA_HIDDEN_SIZE,
grad_W2_multiply_by_learning_rate, double);
        SUBTRACT_ARRAY_FROM_ARRAY(W2, grad_W2_multiply_by_learning_rate, REPLIKA_VOCABULARY_LENGTH*REPLIKA_HIDDEN_SI
ZE, double);
        epoch_loss = epoch_loss + (-1*log(fpg.y_pred[y_true]));
        alloc_obj.deallocate(reinterpret_cast<char*>(bpg.grad_W1));
        alloc_obj.deallocate(reinterpret_cast<char*>(bpg.grad_W2));
        alloc_obj.deallocate(reinterpret_cast<char*>(fpg.h));
        alloc_obj.deallocate(reinterpret_cast<char*>(fpg.y_pred));
        alloc_obj.deallocate(reinterpret_cast<char*>(grad_W1_multiply_by_learning_rate));
        alloc_obj.deallocate(reinterpret_cast<char*>(grad_W2_multiply_by_learning_rate));
        bpg.grad_W1 = NULL;
        bpg.grad_W2 = NULL;
        fpg.h = NULL;
        fpg.y_pred = NULL;
```

7. Evaluation: You should always keep aside some data points from your original training set for testing. Here we evaluate how the NN predicts data points, it's never been trained on

```
#define WORDS "put some words here which are part of vocabulary"
data = cc_tokenizer::String<char>(WORDS);
data_parser =  cc_tokenizer::csv_parser<cc_tokenizer::String<char>, char>(data);
double (*word_vectors)[REPLIKA_HIDDEN_SIZE] = NULL;
unsigned int k  = 0;
data_parser.go_to_next_line();
word_vectors = reinterpret_cast<double (*)[REPLIKA_HIDDEN_SIZE]>(alloc_obj.allocate(sizeof(double)*data_parser.get_t
otal_number_of_tokens()*REPLIKA_HIDDEN_SIZE));
while (data_parser.go_to_next_token() != cc_tokenizer::string_character_traits<char>::eof())
{
   unsigned int i = vocab[data_parser.get_current_token()]->index - REPLIKA_PK_INDEX_ORIGINATES_AT;
   for (unsigned int j = 0; j < REPLIKA_HIDDEN_SIZE; j++)
   {
       word_vectors[k][j] = W1[i][j];
   }
   k = k + 1;
}
k = 0;
data_parser.reset(TOKENS);
while (data_parser.go_to_next_token() != cc_tokenizer::string_character_traits<char>::eof())
{
    unsigned int i = vocab[data_parser.get_current_token()]->index - REPLIKA_PK_INDEX_ORIGINATES_AT;
    std::cout<<"--> "<<vocab(i).c_str()<<", vocab_word_index = "<<i<<", word_embedding = ";
    for (int j = 0; j < REPLIKA_HIDDEN_SIZE; j++)
    {
        std::cout<<*(*(word_vectors + k) + j)<<" ";
    }
    k = k + 1;
    std::cout<<std::endl;
}
k = 0;
```

```
data_parser.reset(TOKENS);
while (data_parser.go_to_next_token() != cc_tokenizer::string_character_traits<char>::eof())
{
    for (unsigned int j = k + 1; j < data_parser.get_total_number_of_tokens(); j++)
    {
       double sim = cosine_distance(word_vectors[k], word_vectors[j], REPLIKA_HIDDEN_SIZE);
       printf("Similarity between %s and %s : %f", data_parser.get_token_by_number(k + 1).c_str(), data_parser.get_t
oken_by_number(j + 1).c_str(), sim);
            std::cout<<std::endl;
    }
    k = k + 1;
}
```