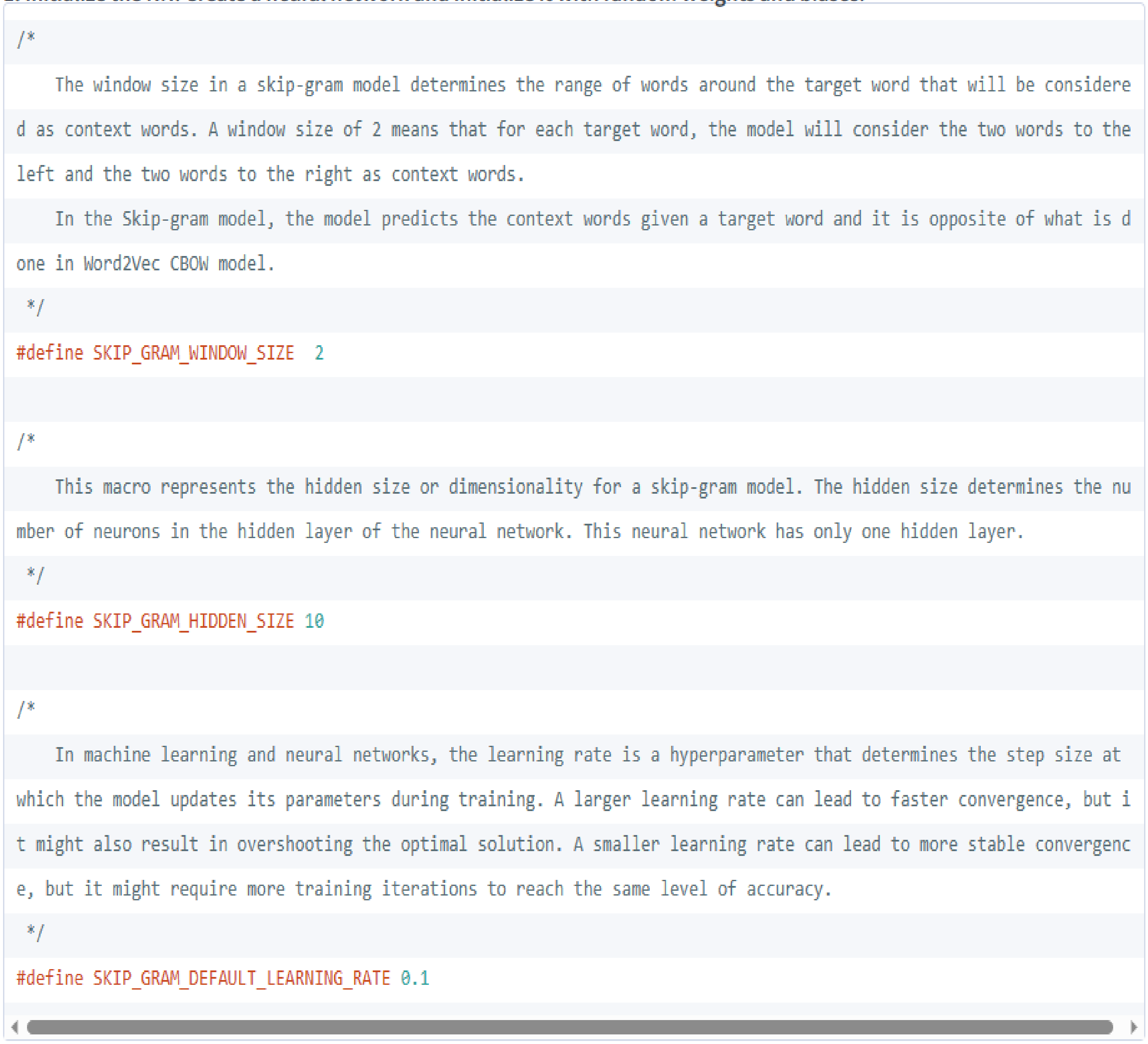
Building an Unsupervised NLP Skip-Gram Word2Vec Model from Scratch in C/C++: A Step-by-Step Implementation, by Q@spysee.pk

"The skip-gram model is a type of neural network architecture used for natural language processing tasks. In natural language processing tasks, the skip-gram model is used to learn word representations (referred to as target words) based on their contexts."

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

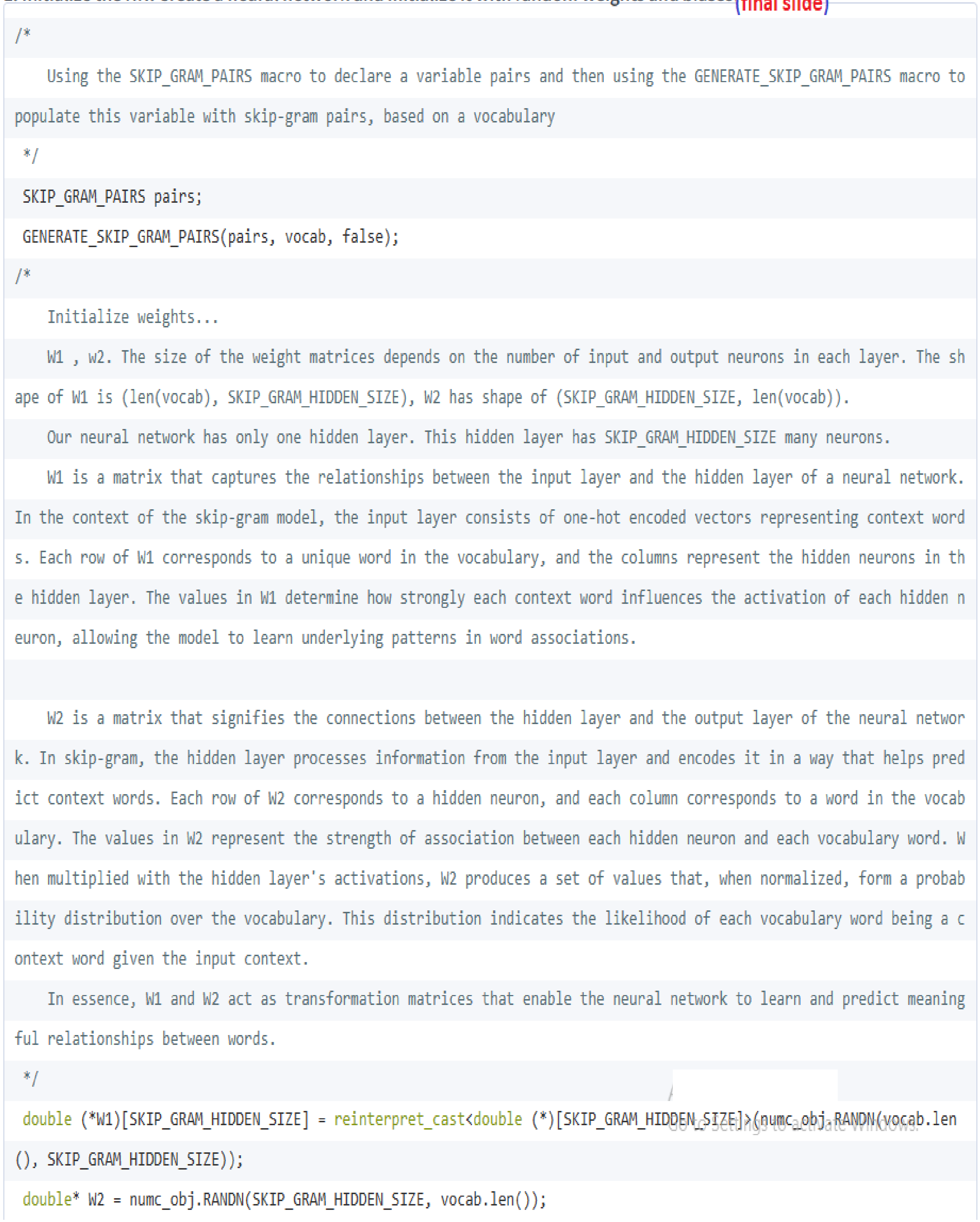


```c
/*
    The window size in a skip-gram model determines the range of words around the target word that will be considere
d as context words. A window size of 2 means that for each target word, the model will consider the two words to the
left and the two words to the right as context words.
    In the Skip-gram model, the model predicts the context words given a target word and it is opposite of what is d
one in Word2Vec CBOW model.
 */
#define SKIP_GRAM_WINDOW_SIZE  2

/*
    This macro represents the hidden size or dimensionality for a skip-gram model. The hidden size determines the nu
mber of neurons in the hidden layer of the neural network. This neural network has only one hidden layer.
 */
#define SKIP_GRAM_HIDDEN_SIZE 10

/*
    In machine learning and neural networks, the learning rate is a hyperparameter that determines the step size at
which the model updates its parameters during training. A larger learning rate can lead to faster convergence, but i
t might also result in overshooting the optimal solution. A smaller learning rate can lead to more stable convergenc
e, but it might require more training iterations to reach the same level of accuracy.
 */
#define SKIP_GRAM_DEFAULT_LEARNING_RATE 0.1
```

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



```
/*
    Using the SKIP_GRAM_PAIRS macro to declare a variable pairs and then using the GENERATE_SKIP_GRAM_PAIRS macro to
populate this variable with skip-gram pairs, based on a vocabulary
 */
 SKIP_GRAM_PAIRS pairs;
 GENERATE_SKIP_GRAM_PAIRS(pairs, vocab, false);
/*
    Initialize weights...
    W1 , w2. The size of the weight matrices depends on the number of input and output neurons in each layer. The sh
ape of W1 is (len(vocab), SKIP_GRAM_HIDDEN_SIZE), W2 has shape of (SKIP_GRAM_HIDDEN_SIZE, len(vocab)).
    Our neural network has only one hidden layer. This hidden layer has SKIP_GRAM_HIDDEN_SIZE many neurons.
    W1 is a matrix that captures the relationships between the input layer and the hidden layer of a neural network.
In the context of the skip-gram model, the input layer consists of one-hot encoded vectors representing context word
s. Each row of W1 corresponds to a unique word in the vocabulary, and the columns represent the hidden neurons in th
e hidden layer. The values in W1 determine how strongly each context word influences the activation of each hidden n
euron, allowing the model to learn underlying patterns in word associations.

    W2 is a matrix that signifies the connections between the hidden layer and the output layer of the neural networ
k. In skip-gram, the hidden layer processes information from the input layer and encodes it in a way that helps pred
ict context words. Each row of W2 corresponds to a hidden neuron, and each column corresponds to a word in the vocab
ulary. The values in W2 represent the strength of association between each hidden neuron and each vocabulary word. W
hen multiplied with the hidden layer's activations, W2 produces a set of values that, when normalized, form a probab
ility distribution over the vocabulary. This distribution indicates the likelihood of each vocabulary word being a c
ontext word given the input context.
    In essence, W1 and W2 act as transformation matrices that enable the neural network to learn and predict meaning
ful relationships between words.
 */
 double (*W1)[SKIP_GRAM_HIDDEN_SIZE] = reinterpret_cast<double (*)[SKIP_GRAM_HIDDEN_SIZE]>(numc_obj.RANDN(vocab.len
(), SKIP_GRAM_HIDDEN_SIZE));
 double* W2 = numc_obj.RANDN(SKIP_GRAM_HIDDEN_SIZE, vocab.len());
```

2. Feedforward pass: NN predicts an outcome based on input. For example: given the input target word, the network aims to predict the context words that are likely to appear in its vicinity within a given context window and has our NN does that correctly?

This code block represents the feedforward pass of the neural network of the Skip-Gram model.

- forward: This function is responsible for executing the forward pass of the neural network. It takes several parameters, including the index of the center or target word in the vocabulary (center_or_target_word_index_in_vocabulary), the weight matrices W_1 and W_2, instances of the corpus class (vocab), and the numc class (numc_obj), numerical computation library.
- h vector has shape (1, SKIP_GRAM_HIDDEN_SIZE), it represents the hidden layer representation. It stores the copy of center or target word at index (center_or_target_word_index_in_vocabulary) in W_1 matrix. The W_1 matrix captures the relationship between input layere and hidden layer. The h is used in both the forward and backward passes of the network.
- u vector has shape (1, vocab.len()), it is used to store the result of the dot product operation between the center or target word vector “h” and the weight matrix W_2. The result stored in u, captures the combined influence of hidden neurons on predicting context words. It provides a numerical representation of how likely each word in the vocabulary is to be a context word for the given target word. The variable u serves as an intermediary step in the forward pass, representing the activations before applying the softmax function to generate the predicted probabilities.
- softmax function is our activation function. It is used to convert the raw scores or activations into a probability distribution, which represents the predicted likelihood of each class (in this case, each word in the vocabulary).
- y_pred vector has shape (1, vocab.len()), each value in the resulting y_pred vector represents the predicted probability of the corresponding vocabulary word being a context word for the given target word. The y_pred vector, obtained through the softmax transformation, forms the basis for evaluating the model’s predictions and training it to improve its accuracy in predicting context words based on input target words.

In summary the forward() function orchestrates the feedforward pass of the skip-gram model, which predicts context words given a target word. It takes the index of the target word in the vocabulary, along with weight matrices W_1 and W_2, and references to the vocabulary and numerical computation objects. The function first allocates memory for the target word vector h and copies it from W_1. It then computes the dot product of h and W_2, generating a vector u that encapsulates the combined influence of hidden neurons on context word prediction. This vector is then transformed using the softmax function to yield a predicted probability distribution, y_pred, over the vocabulary. The function returns a structure containing h (the target word vector) and y_pred (the predicted probabilities), representing the neural network’s estimations of the relationships between the target word and potential context words.

2. Feedforward pass: NN predicts an outcome based on input. For example: given the input target word, the network aims to predict the context words that are likely to appear in its vicinity within a given context window and has our NN does that correctly?(final slide)

```
forward_propogation<double> forward(corpus::corpus_index_type center_or_target_word_index_in_vocabulary, double (*W1)[SKIP_GRAM_HIDDEN_SIZE], double* W2, class corpus& vocab, class numc& numc_obj)
{
    cc_tokenizer::allocator<char> alloc_obj;
    // Center or target word implies that it is a vec from W1
    // Center or target word vetor is what h has shape of (1, SKIP_GRAM_HIDDEN_SIZE)
    double* h = reinterpret_cast<double*>(alloc_obj.allocate(sizeof(double)*SKIP_GRAM_HIDDEN_SIZE));
    memcpy(h, W1[center_or_target_word_index_in_vocabulary], SKIP_GRAM_HIDDEN_SIZE*sizeof(double));
    // Shape of u is (1, vocab.len())
    double* u = numc_obj.dot(h, W2, {SKIP_GRAM_HIDDEN_SIZE, 1, NULL, NULL}, {vocab.len(), SKIP_GRAM_HIDDEN_SIZE}, REPLIKA_PK_NUMC_YAXIS);
    // y_pred has shape of (1, vocab.len())
    double* y_pred = softmax(u, vocab, numc_obj);
    alloc_obj.deallocate(reinterpret_cast<char*>(u));
    return {h /* shape (1, SKIP_GRAM_HIDDEN_SIZE) */, y_pred /* shape (1, vocab.len()) */};
}
```

3. Calculate Loss: Initially, the skip-gram model's predictions are far from accurate. The difference between these predictions and the actual context words that should be associated with a given target word is quantified as the loss. This loss measures the disparity between the predicted and actual context word probabilities. If the model wrongly predicts unlikely context words for a given target, the loss is higher, and conversely, when the predictions align more closely with the true context words, the loss decreases.

```
// Training loop (scaled-down)
#define SKIP_GRAM_TRAINING_LOOP(W1, W2, pairs, numc_obj, vocab, epoch, lr, verbose)  {
    for (unsigned long i = 0; i < epoch; i++)
    {
        double epoch_loss = 0;\
        while ((pair = copy_of_pairs[j + REPLIKA_PK_INDEX_ORIGINATES_AT]) != NULL)
        {
            forward_propogation<double> fp = forward(pair[SKIP_GRAM_PAIR_TARGET_INDEX], W1, W2, vocab, numc_obj);
            epoch_loss = epoch_loss + (-1*log(fp.y_pred[pair[SKIP_GRAM_PAIR_CONTEXT_INDEX]]));
            std::cout<<"Current epoch is = "<<i + 1<<" and epoch loss is "<<epoch_loss/pairs.len()<<std::endl;
        }
    }
```

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(forward propogation).

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.

Description of the backpropagation process for the skip-gram mode as implemented in function `struct backward_propogation<double> backward(corpus::corpus_index_type target_or_center_word_index_in_vocabulary, corpus::corpus_index_type context_word_index_in_vocabulary, double (*W1)[SKIP_GRAM_HIDDEN_SIZE], double* W2, class corpus& vocab, class numc& numc_obj, forward_propogation<double>& fp);`

- It calculates the gradient of the output u (before the softmax) with respect to the loss.
- It calculates the gradient of the weight matrix W_2 using the outer product of the hidden layer output h and the gradient grad_u.
- It calculates the gradient of the hidden layer output h with respect to the loss using the dot product of grad_u and the transposed W_2.
- It initializes a gradient matrix grad_W_1 for the weight matrix W_1.
- It updates the grad_W_1 matrix by adding the calculated gradient grad_h for the specific center/target word.
- The function returns a structure containing the gradients for W_1 and W_2.

In summary this function performs backpropagation step. It computes gradients for adjusting the weight matrices during training to minimize the loss.

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(forward propogation).(final slide)

```
struct backward_propogation<double> backward(corpus::corpus_index_type target_or_center_word_index_in_vocabulary, corpus::corpus_index_type context_word_index_in_vocabulary, double (*W1)[SKIP_GRAM_HIDDEN_SIZE], double* W2, class corpus& vocab, class numc& numc_obj, forward_propogation<double>& fp) {
    cc_tokenizer::allocator<char> alloc_obj;
    // The hot_one_array is row vector, and has shape (1, vocab.len)
    double* hot_one_array = one_hot<double>(context_word_index_in_vocabulary - REPLIKA_PK_INDEX_ORIGINATES_AT, vocab.len(), numc_obj);
    // The shape of grad_u is the same as y_pred which is (1, len(vocab))
    double* grad_u = numc_obj.subtract_matrices<double>(fp.y_pred, hot_one_array, vocab.len());
    // The grad_W2 has shape/dimensions of (REPLIKA_HIDDEN_SIZE, REPLIKA_VOCABULARY_LENGTH)
    double* grad_W2 = numc_obj.outer<double>(fp.h, grad_u, SKIP_GRAM_HIDDEN_SIZE, vocab.len());
    // Shape of W2_T is (REPLIKA_VOCABULARY_LENGTH, REPLIKA_HIDDEN_SIZE), the matrix W2 has shape of (REPLIKA_HIDDEN_SIZE, REPLIKA_VOCABULARY_LENGTH)
    double* W2_T = numc_obj.transpose_matrix<double>(reinterpret_cast<double*>(W2), SKIP_GRAM_HIDDEN_SIZE, vocab.len());
    double* grad_h = numc_obj.dot(grad_u, W2_T, {vocab.len(), 1, NULL, NULL}, {SKIP_GRAM_HIDDEN_SIZE, vocab.len(), NULL, NULL}, REPLIKA_PK_NUMC_YAXIS);
    // The shape/dimensions of grad_W1 is (REPLIKA_VOCABULARY_LENGTH, REPLIKA_HIDDEN_SIZE)
    double* grad_W1 = numc_obj.zeros<double>(2, vocab.len(), SKIP_GRAM_HIDDEN_SIZE);
    SUM_OF_TWO_ARRAYS(reinterpret_cast<double(*)[SKIP_GRAM_HIDDEN_SIZE]>(grad_W1)[target_or_center_word_index_in_vocabulary], grad_h, reinterpret_cast<double(*)[SKIP_GRAM_HIDDEN_SIZE]>(grad_W1)[target_or_center_word_index_in_vocabulary], 1, SKIP_GRAM_HIDDEN_SIZE, double);
    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.

The gradient descent steps adjust the weights based on the calculated gradients to minimize the loss. Let's break down the code(if the next slide) and relate it to the gradient descent process.

- Multiplying by Learning Rate: The gradients bp.grad_W1 and bp.grad_W2 (which represent the changes needed in the weight matrices) are multiplied by a small factor known as the learning rate (lr). The learning rate determines the step size to take in the direction of minimizing the loss.
- Update Values: The resulting products of the learning rate and gradients are stored in product_of_lr_and_grad_W1 and product_of_lr_and_grad_W2.
- Subtraction from Weight Matrices: The values in product_of_lr_and_grad_W1 are subtracted from the original W1 matrix, and similarly, the values in product_of_lr_and_grad_W2 are subtracted from the original W2 matrix. This subtraction adjusts the weights in the direction that reduces the loss.

In essence, this code implements the gradient descent algorithm by updating the weights based on the calculated gradients and the learning rate. The goal is to iteratively adjust the weights in the direction that minimizes the loss, thereby improving the model's accuracy over time. This process is a fundamental step in training neural networks, including the skip-gram model.

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)

```
// Training loop (scaled-down)
#define SKIP_GRAM_TRAINING_LOOP(W1, W2, pairs, numc_obj, vocab, epoch, lr, verbose)  {
    for (unsigned long i = 0; i < epoch; i++) {
        while ((pair = copy_of_pairs[j + REPLIKA_PK_INDEX_ORIGINATES_AT]) != NULL) {
            forward_propogation<double> fp = forward(pair[SKIP_GRAM_PAIR_TARGET_INDEX], W1, W2, vocab, numc_obj);
            backward_propogation<double> bp = backward(pair[SKIP_GRAM_PAIR_TARGET_INDEX], pair[SKIP_GRAM_PAIR_CONTEX
T_INDEX], W1, W2, vocab, numc_obj, fp);

            MULTIPLY_ARRAY_BY_SCALAR(bp.grad_W1, lr, vocab.len()*SKIP_GRAM_HIDDEN_SIZE, product_of_lr_and_grad_W1, d
ouble);
            MULTIPLY_ARRAY_BY_SCALAR(bp.grad_W2, lr, vocab.len()*SKIP_GRAM_HIDDEN_SIZE, product_of_lr_and_grad_W2, d
ouble);
            SUBTRACT_ARRAY_FROM_ARRAY(W1, product_of_lr_and_grad_W1, vocab.len()*SKIP_GRAM_HIDDEN_SIZE, double);
            SUBTRACT_ARRAY_FROM_ARRAY(W2, product_of_lr_and_grad_W2, vocab.len()*SKIP_GRAM_HIDDEN_SIZE, double);
            epoch_loss = epoch_loss + (-1*log(fp.y_pred[pair[SKIP_GRAM_PAIR_CONTEXT_INDEX]]));
        }
    }
}
```

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.

```
#define SKIP_GRAM_TRAINING_LOOP(W1, W2, pairs, numc_obj, vocab, epoch, lr, verbose)  {
 cc_tokenizer::allocator<char> alloc_obj;SKIP_GRAM_PAIRS copy_of_pairs = pairs;double* product_of_lr_and_grad_W1 = N
ULL;double* product_of_lr_and_grad_W2 = NULL; for (unsigned long i = 0; i < epoch; i++) { double epoch_loss = 0; cop
y_of_pairs.shuffle(); skip_gram_pairs::number_of_pairs_type j = 0; corpus::corpus_index_type* pair = NULL; while ((p
air = copy_of_pairs[j + REPLIKA_PK_INDEX_ORIGINATES_AT]) != NULL) { forward_propogation<double> fp = forward(pair[SK
IP_GRAM_PAIR_TARGET_INDEX], W1, W2, vocab, numc_obj); backward_propogation<double> bp = backward(pair[SKIP_GRAM_PAIR
_TARGET_INDEX], pair[SKIP_GRAM_PAIR_CONTEXT_INDEX], W1, W2, vocab, numc_obj, fp); MULTIPLY_ARRAY_BY_SCALAR(bp.grad_W
1, lr, vocab.len()*SKIP_GRAM_HIDDEN_SIZE, product_of_lr_and_grad_W1, double); MULTIPLY_ARRAY_BY_SCALAR(bp.grad_W2, l
r, vocab.len()*SKIP_GRAM_HIDDEN_SIZE, product_of_lr_and_grad_W2, double); SUBTRACT_ARRAY_FROM_ARRAY(W1, product_of_l
r_and_grad_W1, vocab.len()*SKIP_GRAM_HIDDEN_SIZE, double); SUBTRACT_ARRAY_FROM_ARRAY(W2, product_of_lr_and_grad_W2,
vocab.len()*SKIP_GRAM_HIDDEN_SIZE, double); epoch_loss = epoch_loss + (-1*log(fp.y_pred[pair[SKIP_GRAM_PAIR_CONTEXT_
INDEX]])); alloc_obj.deallocate(reinterpret_cast<char*>(fp.y_pred)); alloc_obj.deallocate(reinterpret_cast<char*>(f
p.h)); alloc_obj.deallocate(reinterpret_cast<char*>(bp.grad_W1)); alloc_obj.deallocate(reinterpret_cast<char*>(bp.gr
ad_W2)); alloc_obj.deallocate(reinterpret_cast<char*>(pair)); alloc_obj.deallocate(reinterpret_cast<char*>(product_o
f_lr_and_grad_W1)); alloc_obj.deallocate(reinterpret_cast<char*>(product_of_lr_and_grad_W2)); pair = NULL; fp.y_pred
= NULL; fp.h = NULL; bp.grad_W1 = NULL; bp.grad_W2 = NULL;product_of_lr_and_grad_W1 = NULL; product_of_lr_and_grad_W
2 = NULL; j = j + 1; }
 std::cout<<"Current epoch is = "<<i + 1<<" and epoch loss is "<<epoch_loss/pairs.len()<<std::endl;}}
```

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. If your error on the test set is low, fundamentally you have learnt the weights and biases of a model that predict the output given specific inputs and you can now use it to predict on unseen data.

```
/* ************************************************************************************************************** */
                                          WE HAVE TRAINED SKIP-GRAM MODEL
/* ************************************************************************************************************** */
    #define WORDS "Artificial Intelligence AI has emerged as a transformative technology with the potential to revolutionize numerous fields and industries"
    data = cc_tokenizer::String<char>(WORDS);
    data_parser = cc_tokenizer::csv_parser<cc_tokenizer::String<char>, char>(data);
    cc_tokenizer::string_character_traits<char>::size_type j = 0;
    data_parser.go_to_next_line();
    double (*word_vectors)[SKIP_GRAM_HIDDEN_SIZE] = reinterpret_cast<double (*)[SKIP_GRAM_HIDDEN_SIZE]>(alloc_obj.allocate(sizeof(double)*data_parser.get_total_number_of_tokens()*SKIP_GRAM_HIDDEN_SIZE));
    while (data_parser.go_to_next_token() != cc_tokenizer::string_character_traits<char>::eof())
    {
        for (unsigned long i = 0; i < SKIP_GRAM_HIDDEN_SIZE; i++)
        {
            CORPUS_PTR word = vocab[data_parser.get_current_token()];
            if (word == NULL)
            {
                std::cout<<"NULL -> "<<data_parser.get_current_token().c_str()<<std::endl;
                // TODO, deallocate and stop the program
            }
            word_vectors[j][i] = *(W1[vocab[data_parser.get_current_token()]->index - REPLIKA_PK_INDEX_ORIGINATES_A
T] + i);
        }
        j++;
    }
    data_parser.reset(TOKENS);
    for (cc_tokenizer::string_character_traits<char>::size_type i = 0; i < data_parser.get_total_number_of_tokens();
i++)
    {
        for (j = i + 1; j < data_parser.get_total_number_of_tokens(); j++)
        {
            double sim = numc_obj.cosine_distance(word_vectors[i], word_vectors[j], SKIP_GRAM_HIDDEN_SIZE);
            printf("Similarity between %s and %s : %f", data_parser.get_token_by_number(i + 1).c_str(), data_parser.
get_token_by_number(j + 1).c_str(), sim);
            std::cout<<std::endl;
        }
    }
    alloc_obj.deallocate(reinterpret_cast<char*>(W1));
    alloc_obj.deallocate(reinterpret_cast<char*>(W2));
    alloc_obj.deallocate(reinterpret_cast<char*>(word_vectors));
```