Introduction

In this assignment, you will implement the Continuous Bag of Words (CBOW) variant of the Word2Vec model using Python and `numpy`. This hands-on experience will deepen your understanding of word embeddings and their implementation in Natural Language Processing.

The main goals of this assignment are:

- Understand the architecture of the CBOW model.
- Implement core components of a neural network from scratch.
- Learn about implementation details such as numerical stability.
- Learn how weight optimization works in neural networks by implementing backpropagation.

Throughout this assignment, you will implement these components step by step, gradually building up to a full CBOW model.

Submission Instructions

First, fork the following Gitlab repository URL. **Make sure that your fork is private**. Then, clone your forked repository to your local machine. After completing the assignment, push your changes to your forked repository. Make sure to add the TA (username: `msaad`) as a member of your repository with the "*Maintainer*" role. This will allow them to access your repository and grade your assignment. **Finally, paste the link to your forked repository in Brightspace to submit your assignment.**

Requirements and Remarks

- You must use Python 3.7 or later.
- The only external library you are allowed to use is `numpy`. You will also need to install `matplotlib` for plotting the loss curve at the end.
- To install all dependencies, run `pip3 install -r requirements.txt`.
- You must not use any deep learning libraries like TensorFlow or PyTorch.
- You must not use any pre-built Word2Vec libraries.
- You may add other functions or classes as needed. However, you should NOT modify the existing code, other than the lines of code you are asked to complete. Moreover, you should pay attention to the return types and arguments of the functions you are asked to complete. If a function returns an `ndarray`, you should return an `ndarray`, not a list or any other data type.

- The assignment is easy to implement. Do not let the math here overwhelm you. In all cases, the implementation is direct and simple. The handout provides all the necessary information to implement the assignment. In addition, the code is well documented and provides hints to help you understand the implementation.

Deadlines

This assignment has a soft deadline and a hard deadline. The soft deadline is on **Monday, February 3, 2025, 11:59 PM**. The goal of the soft deadline is to provide structure and encourage you to work on it regularly. The code that you submit by the soft deadline will have no bearing on your grade. Meaning, you can submit an incomplete assignment by the soft deadline. However, you must submit the complete assignment by the hard deadline which is on **Wednesday, Februraury 12, 2025, 11:59 PM**. Late submissions will not be accepted.
If you need to submit an SDA, **you must do so 48 hours prior the hard deadline** by submitting an email to the TA and the instructor.

If you have any questions, please do not hesitate to post them on the course's Teams channel. If you have not joined already, you can do so by using this code: ot45hmp

Good luck!

Brief introduction to Word2Vec

The Continuous Bag of Words (CBOW) is a variant of the Word2Vec model used to learn word embeddings. Word embeddings are dense vector representations of words that capture their meanings and relationships. CBOW predicts a target word based on its surrounding context words. Concretely, it is a feeforward neural network with one hidden layer.

CBOW takes a set of context words as input and predicts a target word. For example, given the sentence *"The quick brown fox jumps over the lazy dog"*, the context words for the target word *"jumps"* could be *"The"*, *"quick"*, *"brown"*, *"fox"*, *"over"*, *"the"*, *"lazy"*, *"dog"*. The model is trained to predict the target word *"jumps"* based on these context words. Here *content length* is 4. Meaning, the model will predict the target word based on the 4 words *before* and *after* the target word.

How CBOW Works

The CBOW model works as follows:

1. **Input**: A set of context words.
2. **Embedding Layer**: Each context word is mapped to its corresponding embedding vector using an embedding matrix $\mathbf{E}$.

3. **Concatenation**: The embeddings of the context words are concatenated into a single vector:
$$\mathbf{v}_{\text{context}} = [\mathbf{v}_1; \mathbf{v}_2; \ldots; \mathbf{v}_{2C}]$$

4. **Linear Layer**: The concatenated vector is passed through a linear layer to produce logits (scores) for each word in the vocabulary:
$$\mathbf{z} = \mathbf{W} \cdot \mathbf{v}_{\text{context}}$$

5. **Softmax Layer**: The logits are converted into probabilities using the softmax function:
$$p(w_i) = \frac{\exp(z_i)}{\sum_{j=1}^{V} \exp(z_j)}$$

6. **Loss Function**: The model uses the negative log likelihood loss to measure how well it predicts the target word:
$$L = -\log(p(w_{\text{target}}))$$

Backpropagation in CBOW

Backpropagation is used to update the model's parameters (embeddings and weights) by computing gradients of the loss with respect to these parameters. The key steps are:

1. **Forward Pass**: Compute the predicted probabilities for the target word.

2. **Backward Pass**:

   - Compute the gradient of the loss with respect to the logits:
   $$\frac{\partial L}{\partial z_i} = p(w_i) - \mathbb{I}(i = \text{target})$$
   , where $\mathbb{I}(i = \text{target})$ is the indicator function that returns 1 if i is the target word, and 0 otherwise.
   - Propagate the gradients backward through the linear layer to update $\mathbf{W}$.
   - Propagate the gradients further back to update the embeddings of the context words.

3. **Parameter Updates**: Update the parameters using gradient descent:
$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \cdot \frac{\partial L}{\partial \theta}$$
where θ represents the parameters and η is the learning rate.

Key Points to Remember

- CBOW predicts a word from its context.
- Word embeddings capture semantic relationships between words.
- The loss function measures how well the model predicts the target word.
- Backpropagation ensures that gradients are computed and propagated correctly to update the model's parameters.

In this assignment, you will implement these components step by step to build a complete CBOW model.

1 Task 1: Implementing the Parameter Class

In this task, you will implement the `Parameter` class in `model/parameter.py`. This class is fundamental to our neural network implementation, as it will be used to represent trainable parameters , or weights, in our model. In the next tasks, we will use this `Parameter` class to build more complex components of the CBOW model.

1.1 The Parameter Class

The `Parameter` class should have two main attributes:

1. `data`: Stores the actual parameter values (a NumPy array).
2. `grad`: Stores the gradient of the loss with respect to this parameter.

1.2 Your Tasks

1. Complete the `__init__` method:
   - Initialize `self.data` with the input `data`.
   - Initialize `self.grad` with a numpy array of 0s.

1.3 Example Usage

Here's how the `Parameter` class should be used:

```python
# Creating a Parameter instance
param = Parameter(np.random.randn(3, 4))

# Accessing data and grad
print(param.data.shape)  # Should print (3, 4)
print(param.grad)  # Should print the initial value of grad
```

```
# After a backward pass (not implemented here), grad would be populated
# param.grad = np.random.randn(3, 4) # This would be done in the backward pass
# print(param.grad.shape)  # Would print (3, 4)
```

2 Task 2: Implementing Neural Network Layers

In this task, you will implement two fundamental layers for our neural network: the `Linear` layer and the `Embedding` layer. These will be crucial components of the CBOW model.

2.1 The Linear Layer

Before we start, let us first define some terms and their shapes.

- Input: $\mathbf{X}$ (shape: $batch\_size \times in\_features$)
- Weights: $\mathbf{W}$ (shape: $in\_features \times out\_features$)
- Output: $\mathbf{Y} = \mathbf{XW}$ (shape: $batch\_size \times out\_features$)
- Upstream gradient: $\frac{\partial L}{\partial Y}$ (shape: $batch\_size \times out\_features$)

2.1.1 Forward pass

The `Linear` layer performs a linear transformation of the input. Given an input $\mathbf{X}$, it computes:

$$\mathbf{Y} = \mathbf{XW}$$

where $\mathbf{W}$ is the weight matrix. Often, a bias term is also added, but we will omit it for simplicity.

2.1.2 Backward pass

For the backward pass, given the gradient of the loss with respect to the output $\frac{\partial L}{\partial \mathbf{Y}}$, we need to compute two gradients:

1. The gradient of the loss with respect to the input $\frac{\partial L}{\partial \mathbf{x}}$.
2. The gradient of the loss with respect to the weight matrix $\frac{\partial L}{\partial \mathbf{W}}$.

The gradient of the loss with respect to the weight matrix is $\frac{\partial L}{\partial W} = X^T \times \frac{\partial L}{\partial Y}$, with shape $(in\_features \times out\_features)$.

The gradient of the loss with respect to the input is $\frac{\partial L}{\partial X} = (\frac{\partial L}{\partial Y}) \times W^T$, with shape $(batch\_size \times in\_features)$.

We have omitted some mathematical details for brevity and simplicity. You can refer to external resources for a more detailed explanation. However, this all you need to know to implement the `Linear` layer.

2.2 The Embedding Layer

As usual, let us first define some terms and their shapes.

- Embedding matrix: $\mathbf{E}$ (shape: *vocab_size* $\times$ *embedding_dim*)
- Input indices: $\mathbf{I}$ (shape: *batch_size* or *batch_size* $\times$ *sequence_length*)
- Output embeddings: $\mathbf{Y}$ (shape: *batch_size* $\times$ *embedding_dim* or *batch_size* $\times$ *sequence_length* $\times$ *embedding_dim*)
- Upstream gradient: $\frac{\partial L}{\partial \mathbf{Y}}$ (shape: *batch_size* $\times$ *embedding_dim* or *batch_size* $\times$ *sequence_length* $\times$ *embedding_dim*)

The `Embedding` layer maps integer indices to dense vectors. For a vocabulary of size V and embedding dimension D, it maintains a matrix $\mathbf{E} \in \mathbb{R}^{V \times D}$.

For the forward pass with input indices $\mathbf{I}$, it returns:

$$\mathbf{y} = \mathbf{E}[\mathbf{I}]$$

For the backward pass, it accumulates gradients for the selected embeddings:

$$\frac{\partial L}{\partial \mathbf{E}[\mathbf{I}]} += \frac{\partial L}{\partial \mathbf{Y}}$$

2.3 Your Tasks

Complete the implementation of both `Linear` and `Embedding` classes in the `layers.py` file. Specifically, you need to implement:

1. For the `Linear` class:
   - `__init__(self, in_features, out_features)`
   - `forward(self, x)`
   - `backward(self, grad_output)`
2. For the `Embedding` class:
   - `__init__(self, vocab_size, embedding_dim)`
   - `forward(self, indices)`
   - `backward(self, grad_output)`

Use the provided mathematical definitions as a guide for your implementations. Remember to use the `Parameter` class you implemented in the previous task for the learnable parameters for `Linear`, `embeddings` for `Embedding`.

2.4 Hints

Although there are no restrictions on what APIs you should use, you may find the following hints helpful:

- For the `Embedding` layer's backward pass, the `np.add.at` method can be helpful to handle repeated indices correctly.
- The implementation of each layer and its functions should be a straightforward mapping of the mathematical definitions provided above to code. If you find yourself writing a lot of code, you may be overcomplicating things. Note that the code has comments to guide you through the implementation. This is the case for all tasks in this assignment as well.
- For sanity check, make sure that the output shapes of the forward pass and the gradient shapes of the backward pass are correct.

3 Task 3: Implementing the Softmax Layer

3.1 Conceptual Overview

The softmax function converts a vector of arbitrary real-valued numbers (also called logits) into a probability distribution. For input logits $\mathbf{z} = (z_1, z_2, ..., z_V)$, it computes:

$$p_i = \frac{\exp(z_i)}{\sum_{j=1}^{V} \exp(z_j)}$$

where V is the vocabulary size. Output values range between 0 and 1, and sum to 1.

3.2 Why is it Needed in CBOW?

In CBOW Word2Vec:

- The model predicts which word is most likely given its context
- The linear layer outputs logits (raw scores) for all vocabulary words
- Softmax converts these scores into probabilities for meaningful prediction
- Required for computing the loss (negative log likelihood)

3.3 Numerical Stability

Direct implementation can cause overflow due to large exponentials. One way to avoid this is to subtract the maximum logit from all logits before computing the softmax:

$$p_i = \frac{\exp(z_i - \max(\mathbf{z}))}{\sum_{j=1}^{V} \exp(z_j - \max(\mathbf{z}))}$$

Subtracting the maximum logit:

- Preserves the probability distribution.
- Prevents numerical instability.
- Avoids NaN/Inf values during computation.

3.4 Your Tasks

You need to complete the implementation of the `SoftmaxLayer` class in `model/activations.py` file. Specifically, you need to complete the following methods:

1. **Forward Pass `forward(self, x)`:**
    - Subtract max logit for numerical stability
    - Compute exponentials of shifted logits
    - Normalize by sum of exponentials

4 Task 4: Implementing the Loss Function

4.1 What Loss Function is Used?

The CBOW model uses the **Negative Log Likelihood (NLL)** loss to measure how well it predicts target words from context. This loss:

- Penalizes bad predictions (low probability for incorrect words)
- Rewards good predictions (high probability for correct words)
- Provides clear gradients for training

4.2 Key Formulae

For a single (context, target) pair:

1. **Probability of Target Word** (from softmax):

$$p_{\text{target}} = \frac{\exp(z_{\text{target}})}{\sum_{j=1}^{V} \exp(z_j)}$$

2. **Single-Example Loss**:

$$L = -\log(p_{\text{target}})$$

3. **Batch Loss** (average over batch size N):

$$L_{\text{batch}} = -\frac{1}{N}\sum_{i=1}^{N} \log(p_{\text{target}}^{(i)})$$

4.3 Gradient Calculation

The gradient of the loss with respect to logits z_j:

$$\frac{\partial L}{\partial z_j} = \begin{cases} p_j - 1 & \text{if } j = \text{target} \\ p_j & \text{otherwise} \end{cases}$$

Or compactly:

$$\frac{\partial L}{\partial \mathbf{z}} = \mathbf{p} - \mathbf{y}$$

where $\mathbf{y}$ is a one-hot vector of the true target. Note this is just for your understanding. The backward method is already implemented.

4.4 Your Tasks

Your task is to finish the implementation of the `NegativeLogLikelihoodLoss` class in `model/loss.py`.

1. Finish the implementation of the `forward(self, probs, target)` method.

5 Task 5: Implementing the CBOW Model

At this stage, you have implemented all the components needed to build the CBOW model. In this task, you will finish its implementation by combining these components.

5.1 Your Tasks

You need to complete the implementation of the following methods of the `CBOWModel` class in the `model/model.py` file:

1. `__init__(self, vocab_size, embedding_dim, context_size)`: The constructor of the `CBOWModel` class. You need to initialize the embedding layer, the linear layer, the softmax layer, and the loss function. Make sure to use the correct dimensions for each layer, when applicable.

2. `forward(self, context_indices)`: You need to implement the forward pass of the CBOW model, which takes the indices of the context words as input and returns the predicted probabilities of the target word.

3. `backward(self, grad_output)`: The backward pass of the CBOW model, which takes the gradient of the loss with respect to the output and returns the gradient of the loss with respect to the input.

4. `compute_loss(self, context_indices, target_indices)`: The method that computes the loss of the CBOW model given the indices of the context words and the target word.

6 Task 6: Implementing the SGD Optimizer

6.1 Conceptual Overview

Stochastic Gradient Descent (SGD) is a simple yet effective optimization algorithm used to train machine learning models. It updates the model parameters in the direction that reduces the loss function, using the gradient of the loss with respect to each parameter.

The basic SGD update rule for each parameter θ is:

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \cdot \frac{\partial L}{\partial \theta} \tag{1}$$

where θ represents the parameters and η is the learning rate.

6.2 Your Tasks

Your task is to implement the `SGD` class found in the `training/optimizer.py`, which will serve as the optimizer for our CBOW model. You need to implement the following methods:

1. `step(self)`: Perform a single optimization step, updating all parameters. Specifically, it iterates over all parameters and updates them using the SGD update rule specified in Equation 1.

7 Putting It All Together

Now that you have implemented all the necessary components, you can run `python3 main.py`. If everything is implemented correctly, you should get the following output on your terminal.

```
    Vocabulary size: 3599
    Epoch 1/50: 100%|████████████████████████████████████████████████████████████████
838/838 [00:03<00:00, 266.39batch/s]
    Epoch 1/50, Loss: 8.1027
    [OUTPUT TRUNCATED FOR BREVITY]
    Epoch 49/50, Loss: 0.1807
    Epoch 50/50: 100%|███████████████████████████████████████████████████████████████
838/838 [00:03<00:00, 260.20batch/s]
    Epoch 50/50, Loss: 0.1692
```

And you will see a plot showing the loss curve over the epochs. This plot will help you visualize how the loss decreases over time as the model learns to predict the target word from its context.

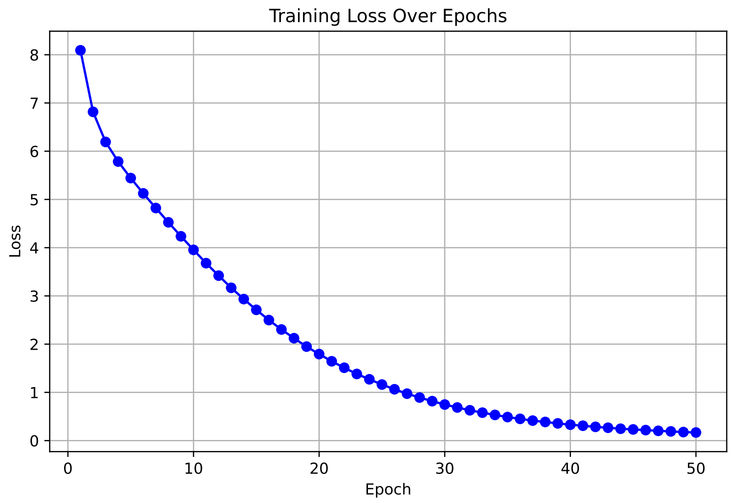

Figure 1: Training Loss Curve

Note that the training time depends on the machine. On a Macbook Pro wih an M2 chip, training for 50 epochs with a context length of 2, an embedding dimension of 50 took around 2 minutes and a half. It is fine if you use other hyperparameters if your machine is slower.

8 Rubric

Note that if the `main.py` file does not execute, a 20% penalty will be applied to the final grade. The rubric is as follows:

1. **Task 1:** Paramter class: 10 point ($\times 2$ 5pt for correctly initializing both attributes).
2. **Task 2:** Linear and Embedding layers: 20 points
   - Forward layer: Correct initialization method implementation: 1 point
   - Forward layer: Correct forward pass implementation: 4 points
   - Backward layer: Correct backward pass implementation: 5 points
   - Backward layer: Correct initialization method implementation: 1 point
   - Backward layer: Correct forward pass implementation: 4 points
   - Backward layer: Correct backward pass implementation: 5 points
3. **Task 3:** Softmax layer: 10 points
4. - Substracting the maximum value: 3 points.
   - Computing the softmax: 7 points.
5. **Task 4:** Loss funtion: 10 points.
6. - Getting the batch size: 2 points.
   - Getting the predicted probabilities: 3 points.
   - Computing the loss: 5 points.
7. **Task 5:** CBOW model implementation: 38 points
8. - Correct initialization of the netowrk's architecture: 6 points.
   - Impelementation of the forward pass: 16 points.
   - Impelementation of the backward pass: 8 points.
   - Implementation of loss computation method: 8 point
9. **Task 6:** Optimizer implementation: correct implementation of the weight update rule 12 points.

   Total: 100 points.