Perfect 👍
Let's go through this **CBOW (Continuous Bag of Words)** PDF code **line by line** — in **simple but technical language**.
By the end, you'll clearly understand what each part does.

🧩1️⃣ Import Libraries

```
import numpy as np
import re
import random
```

- `numpy` → For mathematical operations like dot product, mean, etc.
- `re` → Regular expressions; used here for text cleaning.
- `random` → Used later for selecting random examples.

🧹2️⃣ Data Preparation

```
text = """We are about to study ... People create programs to direct
processes."""
```

A **sample text corpus** — the raw material for your model to learn from.

```
text = re.sub('[^A-Za-z]+', ' ', text)
```

Removes **special characters, numbers, or punctuation**, keeping only words.

```
text = text.lower()
```

Converts everything to lowercase — standardizes words like "The" and "the".

```
corpus = text.split()
```

Splits the text into individual words (a Python list of words).

✅ **Result:**
Now `corpus = ['we', 'are', 'about', 'to', 'study', ...]`

Vocabulary Building

```
vocab = set(corpus)
vocab_size = len(vocab)
```

- `set(corpus)` removes duplicates.
- `vocab_size` counts unique words.

```
word2idx = {word: i for i, word in enumerate(vocab)}
idx2word = {i: word for word, i in word2idx.items()}
```

Creates **lookup tables** to convert:

- Word → Number (`word2idx`)
- Number → Word (`idx2word`)

🧩3️⃣ Generate Training Data

```
data = []
for i in range(2, len(corpus) - 2):
    context = [corpus[i - 2], corpus[i - 1], corpus[i + 1], corpus[i + 2]]
    target = corpus[i]
    data.append((context, target))
```

👉 **Window size = 2** means:

- Take 2 words before and 2 words after the target word as **context**.

🧠 Example:
Sentence part → `['we', 'are', 'about', 'to', 'study']`
For target `"about"`
Context = `[we, are, to, study]`

So, data pair = (`[we, are, to, study]`, `about`)

Each tuple in `data` is a **training sample** for CBOW:

> Context words → predict the target word.

🧱4️⃣ Model Setup (CBOW)

One-hot Encoding

```
def one_hot_encoding(word):
    one_hot = np.zeros(vocab_size)
    one_hot[word2idx[word]] = 1
    return one_hot
```

Creates a vector where:

- Size = `vocab_size`
- All 0's except the index for that word = 1
  🧠 Example: if vocab has 36 words and "study" = index 5 →
  → `[0, 0, 0, 0, 0, 1, 0, 0, ...]`

Softmax Function

```
def softmax(x):
    e_x = np.exp(x - np.max(x))
    return e_x / e_x.sum(axis=0)
```

Converts raw output scores into **probabilities** that sum to 1.
Used to pick which word is most likely the target.

🧱4️⃣ Model Setup — Understanding the Neural Network(Part 2)

Up to now, you cleaned text → split into words → created context–target pairs.
Now you'll build a **tiny neural network** (without TensorFlow or PyTorch!) using just **NumPy**.

🧱 Step 1: Two Weight Matrices

```
embedding_dim = 10
W1 = np.random.rand(vocab_size, embedding_dim)
W2 = np.random.rand(embedding_dim, vocab_size)
```

Let's decode this line by line 👇

🔹 **`embedding_dim = 10`**

This means:

- Each word will be represented by a **10-dimensional vector**.
- You can think of it as "compressing" a word's meaning into 10 numbers.

Example:

```
“apple” → [0.12, 0.58, 0.09, ... , 0.45]
“fruit” → [0.11, 0.61, 0.12, ... , 0.47]
```

So similar words end up with similar vectors after training.

🔹 `W1` — Input to Hidden Layer

`W1` has shape `(vocab_size, embedding_dim)`

Let’s say your vocabulary has 36 words →
then `W1` = 36 × 10 matrix.

Each **row** corresponds to a word,
and each **column** is one number in that word’s 10-dimensional embedding.

🧠 So, if “apple” is at index 5 → row 5 of `W1` gives its current embedding vector.

🔹 `W2` — Hidden to Output Layer

`W2` has shape `(embedding_dim, vocab_size)`
That’s **10 × 36** in our case.

This matrix does the reverse job:

- It converts the 10-dimensional hidden vector back to a probability distribution over all 36 words.
- After multiplication, you get one score for every word → then you apply **softmax** to turn those scores into probabilities.

🧠 Step 2: Forward Pass — Predicting the Target Word

```
def cbow_forward(context_words):
    hidden = np.mean([W1[word2idx[w]] for w in context_words], axis=0)
    output = np.dot(W2.T, hidden)
    prediction = softmax(output)
    return prediction, hidden
```

Let’s go line by line 👇

🔹 `[W1[word2idx[w]] for w in context_words]`

This looks up each **context word**’s embedding from `W1`.

Example:
If context = `[we, are, about, to]`, you get their embeddings → 4 vectors of length 10.

🔹 **`np.mean(..., axis=0)`**

Takes the **average** of all those 4 vectors → gives one combined "context" vector.
That's the **hidden layer representation** — it summarizes the meaning of all nearby words.

🔹 **`np.dot(W2.T, hidden)`**

Now multiply it by `W2.T` (transpose of W2) → this gives a vector of **vocab_size** numbers.
Each number represents the **likelihood** of a particular word being the target.

🔹 **`softmax(output)`**

Converts those raw scores into **probabilities that sum to 1**.
Highest probability → model's predicted target word.

✅ So, this forward function outputs:

- `prediction`: probability for each word in vocab
- `hidden`: the averaged context embedding

🏋️ Step 3: Training — Adjusting Weights

```
for epoch in range(epochs):
    for context, target in data:
        y_pred, h = cbow_forward(context)
        target_one_hot = one_hot_encoding(target)
        error = y_pred - target_one_hot
```

1. Feed a context → get predicted probabilities (`y_pred`)
2. Compare with the actual word (one-hot vector) → get `error`.

⚙️ Step 4: Compute Gradients (Backpropagation)

```
dW2 = np.outer(h, error)
dW1 = np.zeros_like(W1)
for w in context:
    dW1[word2idx[w]] += np.dot(W2, error)
```

- **`dW2`** → how much to adjust output weights.
  It's based on the **outer product** of the hidden layer and the error.
- **`dW1`** → how much to adjust embeddings (W1).
  For every context word, we propagate the error backward to its corresponding word vector.

🔁 Step 5: Update the Weights

```
W1 -= lr * dW1
W2 -= lr * dW2
```

- Subtract the gradient times the **learning rate (lr)**.
- This is **gradient descent** — the model slowly learns word relationships by reducing prediction errors.

📉 Step 6: Calculate Loss

```
loss += -np.sum(target_one_hot * np.log(y_pred + 1e-9))
```

- This is **cross-entropy loss** — measures how far the predicted probabilities are from the true one-hot target.
- The smaller the loss, the better the model is learning.

🗒️ Step 7: Periodic Loss Display

```
if (epoch + 1) % 200 == 0:
    print(f"Epoch {epoch+1}/{epochs}, Loss: {loss:.4f}")
```

Shows progress every 200 epochs.
You can safely reduce training to **200–400 epochs** instead of 2000 — it'll still learn fine for small data.

🧠 Step 8: Prediction

After training:

```
for context, target in random.sample(data, 5):
    prediction, _ = cbow_forward(context)
    predicted_word = idx2word[np.argmax(prediction)]
    print(f"Context: {context} | Predicted: {predicted_word} | Actual: {target}")
```

It randomly picks a few examples and shows:

- The **context words**
- The model's **predicted target**

- The **actual target**

If training went well → both match.

⚙️ Mental Picture of CBOW

```
[Context words] → [Embeddings (W1)] → [Average (Hidden)]
→ [Output (W2)] → [Softmax] → [Predicted target word]
```

Everything — all meaning — is stored in those two weight matrices, **W1** and **W2**.