POE Práctica 1

Sonia Castro Paniello
Eduard Ramon Aliaga

March 2023

1 Improve CBOW model

a) A fixed scalar weight, e.g, (1,2,2,2,2,1) to give more weight to the words that are closer to the predicted central word.

```
class CBOW(nn.Module):
    def _init_(self, num_embeddings, embedding_dim):
        super()._init_()
        self.emb = nn.Embedding(num_embeddings, embedding_dim, padding_idx=0)
        self.lin = nn.Linear(embedding_dim, num_embeddings, bias=False)
        self.register_buffer('position_weight', torch.tensor([1,2,3,3,2,1],
            dtype=torch.float32)).view(1,-1,1)

    # B = Batch size
    # W = Number of context words (left + right)
    # E = embedding_dim
    # V = num_embeddings (number of words)
    def forward(self, input):
        # input shape is (B, W)
        e = self.emb(input)
        weighted_e=e*self.position_weight
        # e shape is (B, W, E)
        u = weighted_e.sum(dim=1)
        # u shape is (B, E)
        z = self.lin(u)
        # z shape is (B, V)
        return z
```

	Tain accuracy	Val. accuracy	Test accuracy	Train loss	Val. loss	Test loss
Epoch 1	29.1	29.9	20.4	4.99	4.81	5.79
Epoch 2	32.5	30.8	21.0	4.52	4.66	5.67
Epoch 3	33.2	31.0	21.0	4.40	4.60	5.61
Epoch 4	33.6	31.5	21.4	4.34	4.56	5.59

b) A trained scalar weight for each position.

```
class CBOW(nn.Module):
    def _init_(self, num_embeddings, embedding_dim):
        super()._init_()
        self.emb = nn.Embedding(num_embeddings, embedding_dim, padding_idx=0)
```

```
        self.lin = nn.Linear(embedding_dim, num_embeddings, bias=False)
        #self.position_weight = nn.Parameter(torch.randn(6), dtype=torch.float32))

    # B = Batch size
    # W = Number of context words (left + right)
    # E = embedding_dim
    # V = num_embeddings (number of words)
    def forward(self, input):
        # input shape is (B, W)
        e = self.emb(input)
        weighted_e=e*self.position_weight
        # e shape is (B, W, E)
        u = weighted_e.sum(dim=1)
        # u shape is (B, E)
        z = self.lin(u)
        # z shape is (B, V)
        return z
```

	Tain accuracy	Val. accuracy	Test accuracy	Train loss	Val. loss	Test loss
Epoch 1	30.9	31.7	22.2	4.76	4.58	5.50
Epoch 2	34.4	32.7	22.5	4.31	4.45	5.41
Epoch 3	35.0	32.9	22.8	4.21	4.40	5.39
Epoch 4	35.4	33.2	22.9	4.17	4.37	5.38

We see how when we train the tensor, the accuracy is improved both in the train, validarion and test. The trained tensor ends with the values [0.0995, 0.2190, 0.5160, 0.5208, 0.2144, 0.1002] so that as expected, it gives more importance to the words close to the central and less to the farthest in a symmetric way.

c) A trained vector weight for each position. Each word vector is element-wise multiplied by the corresponding position-dependent weight and then added with the rest of the weighted word vectors.

```
class CBOW(nn.Module):
    def _init_(self, num_embeddings, embedding_dim):
        super()._init_()
        self.emb = nn.Embedding(num_embeddings, embedding_dim, padding_idx=0)
        self.lin = nn.Linear(embedding_dim, num_embeddings, bias=False)
        self.position_weight = nn.Parameter(torch.randn(6,embedding_dim),
            requires_grad=True)

    # B = Batch size
    # W = Number of context words (left + right)
    # E = embedding_dim
    # V = num_embeddings (number of words)
    def forward(self, input):
        # input shape is (B, W)
        e = self.emb(input)
        weighted_e=e*self.position_weight
        # e shape is (B, W, E)
        u = weighted_e.sum(dim=1)
        # u shape is (B, E)
        z = self.lin(u)
```

```
        # z shape is (B, V)
        return z
```

	Tain accuracy	Val. accuracy	Test accuracy	Train loss	Val. loss	Test loss
Epoch 1	37.6	39.5	29.8	4.15	3.88	4.77
Epoch 2	42.1	40.8	30.6	3.63	3.74	4.65
Epoch 3	43.0	41.2	30.7	3.52	3.69	4.61
Epoch 4	43.4	41.3	30.9	3.48	3.67	4.59

We can see a greater improvement by increasing the number of trained parameters from 6 to 600. We have calculated the average of the values for each word to compare results with the two previous cases [0.0249, 0.0612, 0.0842, 0.0674, 0.0380, 0.0206]. As we can see the model also learns that the closer the words are to the center the most important they are for the prediction. Although in this case we see how the words just before the central have more weight than those immediately after.

d) Hyperparameter optimization: study the performance of the model as a function of one of its parameters: the embedding size, batch size, optimizer, learning rate/scheduler, number of epochs, sharing input/output embeddings.

We have done small tests to try to improve the performance of the algorithm, guiding us by intuition. Since the executions are very slow and we have limited weekly GPU usage. First we have increased the embedding size to 300 so that the model can learn more detailed semantic relationships between words, leading to better word representations.

The second change has been to use a specific learning rate scheduler OneCycleLR with the parameters: $max_lr = 0.01, steps\_per\_epoch = len(vocab), epochs = params.epochs$, $anneal\_strategy =' cos', cycle\_momentum = False$.

We have not modified the batch size since even if it makes the convergence faster it needs more memory. We have not modified the optimizer either since we consider that Adam already has a good performance. We have also not shared the two embedding matrices since we will reduce the number of parameters and it may lead to a loss of information and reduced model performance.

Finally we have decided to change the number of embeddings to 500, but as the execution time was to extense, the final value for the parameter has stayed as 300. It resulted in 47% train accuracy, 44,7% valid accuracy for wikipedia and 33,4% valid accuracy for 'el Periodico'.

e) (Optional) Implement other methods to obtain word embeddings.

We have decided to try to implement a SkipGram. Intuition has led us to take the CBOW input as the SkipGram target and vice versa. We have created the model in such a way that from each central word we predict the six words that make up the context (6, $vocab\_size$). We have pivoted dimensions in the train to be able to use the same criteria.

This implementation of this model is very slow, therefore we have not finished executing it. In real use cases it is implemented with negative sampling and it is only classified if they belong to the same window or not. Being a binary classification it is much faster.

```
def load_preprocessed_dataset(prefix):
    # Try loading precomputed vocabulary and preprocessed data files
    token_vocab = Vocabulary()
    token_vocab.load(f'{prefix}.vocab')
    data = []
    for part in ['train', 'valid', 'test']:
        with np.load(f'{prefix}.{part}.npz') as set_data:
            #Change of target and input data
            target, idata = set_data['idata'], set_data['target']
            data.append((idata, target))
            print(f'Number of samples ({part}): {len(target)}')
    print("Using precomputed vocabulary and data files")
    print(f'Vocabulary size: {len(token_vocab)}')
    return token_vocab, data


class SkipGram(nn.Module):
    def __init__(self, vocab_size, embed_size):
        super(SkipGram, self).__init__()
        self.embeddings = nn.Embedding(vocab_size, embed_size)
        self.linear = nn.Linear(embed_size, vocab_size, bias = False)

    def forward(self, inputs):
        embeds = self.embeddings(inputs)
        embeds = embeds.unsqueeze(1).repeat(1, 6, 1)
        output = self.linear(embeds)
        return output


def train(model, criterion, optimizer, idata, target, batch_size, device,
    log=False):
    model.train()
    total_loss = 0
    ncorrect = 0
    ntokens = 0
    niterations = 0


    for X, y in batch_generator(idata, target, batch_size, shuffle=True):
        # Get input and target sequences from batch
        X = torch.tensor(X, dtype=torch.long, device=device)
        y = torch.tensor(y, dtype=torch.long, device=device)

        model.zero_grad()
        output = model(X)

        output = output.permute(0, 2, 1)
        loss = criterion(output, y)
        loss.backward()
        optimizer.step()
        # Training statistics
```

```
        total_loss += loss.item()
        ncorrect += (torch.max(output, 1)[1] == y).sum().item()
        ntokens += y.numel()
        niterations += 1
        if niterations == 200 or niterations == 500 or niterations % 1000 == 0:
            print(f'Train: wpb={ntokens//niterations}, num_updates={niterations},
                accuracy={100*ncorrect/ntokens:.1f},
                loss={total_loss/ntokens:.2f}')

    total_loss = total_loss / ntokens
    accuracy = 100 * ncorrect / ntokens
    if log:
        print(f'Train: wpb={ntokens//niterations}, num_updates={niterations},
            accuracy={accuracy:.1f}, loss={total_loss:.2f}')
    return accuracy, total_loss
```

We have executed the first epoch after 24000 updates where it had reached values of: accuracy=7.2, loss=7.05.

2 Evaluate the performance of the improved word vectors

a) Implement the WordVector class (Word Vector Analysis notebook) with the most_similar and analogy methods.

```
class WordVectors:
    def __init__(self, vectors, vocabulary):
        self.vectors = vectors
        self.vocabulary = vocabulary
        self.vector_norms = np.linalg.norm(self.vectors, axis=1)

    def most_similar(self, word, topn=10):
        # Get the index and vector of the input word
        word_index = self.vocabulary.get_index(word)
        word_vector = self.vectors[word_index]

        # Calculate the cosine similarity between the input word vector and all
            other vectors
        cosine_similarities = np.dot(self.vectors, word_vector) /
            (self.vector_norms * np.linalg.norm(word_vector))

        # Sort in descending order
        most_similar_indices = np.argsort(cosine_similarities)[::-1]

        # Get the top n most similar words
        topn_similar_words=[]
        for i in most_similar_indices:
            #We don't take into account '<pad>' as a valid word for synonsyms
                since it would always appear in the first place because it's norm
                is 0 and then the cosine similarity 'nan'.
            if i != word_index and i != self.vocabulary.get_index('<pad>'):
                topn_similar_words.append((self.vocabulary.get_token(i),
                    cosine_similarities[i]))
        return topn_similar_words[:topn]

    def analogy(self, x1, x2, y1, topn=5, keep_all=False):
        # Get the indices and vectors of the input words
        x1_index = self.vocabulary.get_index(x1)
        x2_index = self.vocabulary.get_index(x2)
        y1_index = self.vocabulary.get_index(y1)

        x1_vector = self.vectors[x1_index]
        x2_vector = self.vectors[x2_index]
        y1_vector = self.vectors[y1_index]

        # Calculate the vector for the analogy following the formula
        analogy_vector = y1_vector + (x2_vector - x1_vector)

        # Calculate the cosine similarity between the analogy vector and all
            other vectors
        cosine_similarities = np.dot(self.vectors, analogy_vector) /
            (self.vector_norms * np.linalg.norm(analogy_vector))

        # Sort in descending order
        most_similar_indices = np.argsort(cosine_similarities)[::-1]
```

```
        if keep_all:
            # Get all the similar words
            similar_words = [(self.vocabulary.get_token(i),
                cosine_similarities[i]) for i in most_similar_indices if i not in
                (x1_index, x2_index, y1_index)]
        else:
            # Get the topn most similar words
            similar_words = [(self.vocabulary.get_token(i),
                cosine_similarities[i]) for i in most_similar_indices if i not in
                (x1_index, x2_index, y1_index)][:topn]

        return similar_words
```

b) Intrinsic evaluation: perform an informal evaluation finding good and bad examples of closest words and analogies.

We have conducted a series of experiments with this function the results of which are commented here and can be consulted in the annex.

First we check the operation of *most_similar* with homograph words. In the case of the word 'banc', words related to its three meanings have appeared: finance: 'proveïdor', furniture: 'parc' and group of fish: 'buc'. However, with the word 'clau' (crucial/keys) only synonyms of the first meaning appear.The same with the word 'casa' (building/marriage) only synonyms of the first meaning appear.

We see how the model classifies antonymous words as similar, for example if we search for words similar to 'content' the model shows us 'satisfet' and 'familiaritzat' but also 'discontent', 'insatisfet' and 'decebut'.

Testing some examples we have found a certain gender bias in the search for synonyms:

- 'dones': 'noies', 'prostitutes', 'mares', 'nenes', 'llars', 'persones', 'famílies', 'femelles'.
- 'homes': 'soldats', 'guerrers', 'espectadors', 'mariners', 'atletes', 'insurgents', 'tiradors'.
- 'metge': 'cirurgià', 'advocat', 'jurista', 'farmacèutic', 'psiquiatre'.
- 'metgessa' : 'educadora', 'fotògrafa', 'escriptora', 'infermera', 'ballarina'.

In particular 'dones' was one of the words in the top 50 most frequent ones so is suprising that the second most similar word found is 'prostitutes'.

We have also found some synonyms with a certain bias, for example 'trasngènere' comes out as a similar word to 'dolents' and to 'heterosexual' words like 'pecador', 'delinquent' i 'infeliç'. In this last case we assume that it is because in the contexts where the word 'heterosexual' is used it is because the term 'homosexual' has also been mentioned.

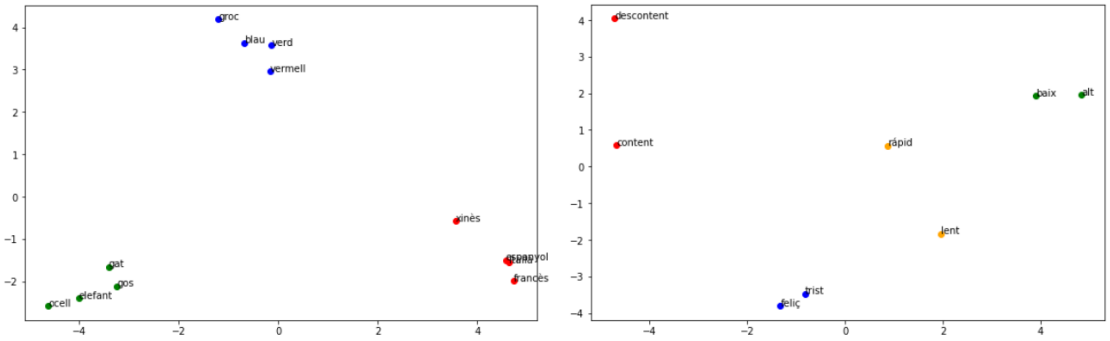
If the searched word is not in the corpus, the answer is quite random as we can see when whe search 'hola' synonyms. Choosing words from the 100 least frequent we see how the results are usually correct although in some cases there are no synonyms but words from the context where it has been used for example 'mascle' appear 'mosquit', 'virus"

or ‘paràsit’. On the other hand, with words among the 50 most frequent the result is excellent, for example ‘empreses’ o ‘ciutat’.

As for the analogies, we find that the operation is more erroneous since, for example, futbol-futbolista esport-esportista does find it correctly but art-artista esport-esportista does not.

c) Visualize word analogies or word clustering properties

Let’s view a couple of examples to check how our algorithm works. On the first plot we see how in this case we can easily divide the words according to their position. On the left we have demonym, on the right the animals and on top colors. In the demonym group we see how Europeans are closer to each other while ’xinès’ is further away. On the second plot we see how the antonyms are close to each other.



(a) *most_similar* with demonym and animals. (b) *most_similar* with antonyms.

Figure 1: *most_similar*

In this case we see how the same analogies between words are represented with different angles and we see why futbol-futbolista esport-esportista does find it correctly but art-artista esport-esportista does not

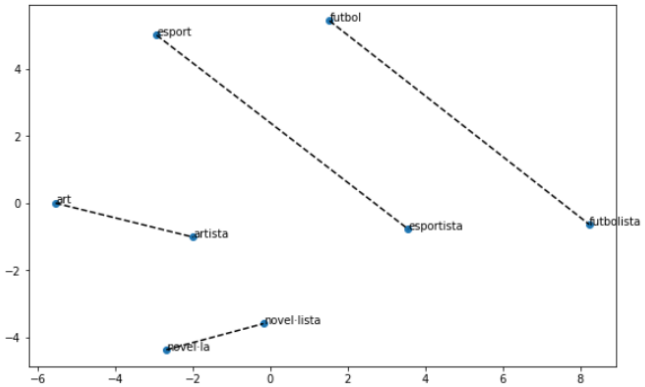


Figure 2: *analogies*

d) Prediction accuracy: compare the accuracy of the implemented CBOW models in the out-of-domain (el Periódico) test set.

The submission that has achieved the best results is the one corresponding to the case explained in the hyperparameters question, where the size of embeding is 300 and we use the sheduler OneCycleLR. The accuracy value achieved is 0.31850.

A Annex

banc:
[('pavelló', 0.6326744),
('consorci', 0.6019502),
('buc', 0.60116994),
('balneari', 0.59840196),
('proveïdor', 0.5937121),
('pilar', 0.5933259),
('casino', 0.5869871),
('balcó', 0.58653957),
('parc', 0.5853937),
('compartiment', 0.58093274)]

clau:
[('fonamental', 0.5915343),
('essencial', 0.584306),
('decisiu', 0.57724375),
('crucial', 0.56790257),
('determinant', 0.5383517),
('imprescindible', 0.508611),
('decoratiu', 0.50403416),
('papereta', 0.5022825),
('importantíssim', 0.4802888),
('crucials', 0.4759672)]

casa:
[('finca', 0.78622985),
('mansió', 0.7588196),
('masia', 0.7164712),
('vil·la', 0.6953218),
('cabana', 0.6952565),
('caseta', 0.69269466),
('granja', 0.66089606),
('capella', 0.65555006),
('Casa', 0.65543234),
('senyora', 0.65359384)]

content:
[('satisfet', 0.6494254),
('familiaritzat', 0.6034094),
('contenta', 0.57684803),
('descontent', 0.57007015),
('insatisfet', 0.559078),
('satisfeta', 0.54509574),
('obsessionada', 0.53585774),
('enfadat', 0.53458244),
('decebut', 0.5312384),
('entusiasmat', 0.52031064)]

dones:
[('noies', 0.64666706),
('prostitutes', 0.5950855),
('mares', 0.5840461),
('nenes', 0.5753293),
('llars', 0.5728575),
('persones', 0.5542778),
('famílies', 0.5476472),
('femelles', 0.54419535),
('universitats', 0.54261416),
('religions', 0.5353677)]

homes:
[('soldats', 0.6548615),
('guerrers', 0.6048175),
('espectadors', 0.5730584),
('mariners', 0.5694886),
('individus', 0.5659739),
('atletes', 0.561046),
('insurgents', 0.5549963),
('nois', 0.5534073),
('tiradors', 0.54939175),
('zulus', 0.54246175)]

metge:
[('cirurgià', 0.81231344),
('advocat', 0.72833973),
('jurista', 0.70732856),
('farmacèutic', 0.70717967),
('psiquiatre', 0.70165205),
('psicòleg', 0.69942343),
('capellà', 0.6990914),
('clergue', 0.6941131),
('músic', 0.69400644),
('fotògraf', 0.68808836)]

metgessa:
[('educadora', 0.74193805),
('fotògrafa', 0.69388306),
('escriptora', 0.665247),
('infermera', 0.6649151),
('ballarina', 0.6422394),
('historiadora', 0.63926727),
('psicòloga', 0.6368253),
('professora', 0.63537747),
('filòsofa', 0.62866455),
('advocada', 0.6277417)]

transgènere:
[('LGBT', 0.597367),
('LGTB', 0.56813854),
('gai', 0.5679294),
('queer', 0.5517653),
('lesbiana', 0.5409099),
('autistes', 0.5337209),
('Holloway', 0.52025187),
('transsexuals', 0.50433195),
('dolents', 0.5003812),
('homosexuals', 0.48831928)]

heterosexual:
[('valenta', 0.5223121),
('governanta', 0.5197408),
('homosexual', 0.5066982),
('pecador', 0.5036504),
('delinqüent', 0.49930012),
('reconeixible', 0.49633896),
('infeliç', 0.49523786),
('patriarcal', 0.49112114),
('cordial', 0.48332974),
('innat', 0.4828797)]

hola:
[('polzes', 0.44390225),
('Dur', 0.43741447),
('cels', 0.43103963),
('sagraments', 0.43098953),
('monogràfic', 0.4218921),
('presagis', 0.42111096),
('Annales', 0.42111075),
('estatuts', 0.41717497),
('subtítols', 0.40919),
('Fasti', 0.4073453)]

arquitectes:
[('escultors', 0.75521106),
('artistes', 0.74337727),
('escriptors', 0.69368905),
('actors', 0.6814487),
('fotògrafs', 0.6645472),
('dissenyadors', 0.65984565),
('directors', 0.65853345),
('economistes', 0.6523975),
('enginyers', 0.6477856),
('compositors', 0.64518946)]

mascle:
[('penis', 0.6054708),
('crani', 0.5711948),
('mascles', 0.561419),
('verí', 0.5532419),
('mosquit', 0.53735995),
('virus', 0.5306436),
('paràsit', 0.522567),
('capoll', 0.5223037),
('gat', 0.5220443),
('brau', 0.5205809)]

empreses:
[('entitats', 0.5781319),
('operadores', 0.56472206),
('Agències', 0.56001997),
('companyies', 0.55600035),
('firmes', 0.55493575),
('agències', 0.54649043),
('indústries', 0.53276175),
('auditories', 0.5184171),
('emissores', 0.5157316),
('implementacions', 0.50573266)]

ciutat
[('vila', 0.84498715),
('ciutadella', 0.74688023),
('fortalesa', 0.740033),
('capital', 0.7283462),
('zona', 0.70296097),
('província', 0.69707227),
('municipalitat', 0.6937457),
('colònia', 0.6900644),
('rodalia', 0.6837004),
('localitat', 0.6825671)]

('futbol', 'futbolista', 'esport')
[('esportista', 0.93419534),
('atleta', 0.92807376),
('exfutbolista', 0.9246578),
('lluitador', 0.9208719),
('economista', 0.9206862)]

('art', 'artista', 'esport')
[('organisme', 0.9325395),
('atleta', 0.92631614),
('escultor', 0.9230982),
('heroi', 0.92248344),
('individu', 0.92170626)]