

correction concours 25_Atef

February 11, 2026

```
[9]: import numpy as np
class Tokenizer:
    def __init__(self, filters='!"$%&()*+,-./:;<=>?
↳ @[\ \ ] ^ _ ` { | } ~ \t \n', lower=True, oov_token=None, split=" "):
        self.filters= filters
        self.lower= lower
        self.oov_token= oov_token
        self.word_index= {oov_token: 0}
        self.split=split
    def tokenize(self, texts):
        L=[]
        for text in texts:
            if self.lower:
                text =text.lower()
            for c in text:
                if c in self.filters:
                    text=text.replace(c, self.split)
            L.append(text.split(self.split))
        return L

    """    def tokenize(self, texts):
        L=[]
        for text in texts:
            if self.lower:
                text =text.lower()
            for c in self.filters:
                text=text.replace(c, self.split)
            L.append(text.split(self.split))
        return L
    """

    def fit_on_texts(self, texts):
        index =len(self.word_index)
        words=self.tokenize(texts)
        for w in words:
            for word in w:
                if word and word not in self.word_index:
                    self.word_index[word]= index
```

```

        index += 1

def to_sequences(self, texts):
    sequences = []
    words = self.tokenize(texts)
    for text in words:
        sequence = []
        for word in text:
            if word :
                if word in self.word_index:
                    sequence.append(self.word_index[word])
                else:
                    sequence.append(0)
            #Token pour les mots inconnus
        sequences.append(sequence)
    return sequences

def generate_pairs(self, texts, window_size= 2):
    tokens = self.to_sequences(texts)
    pairs = list()
    for batch in tokens:
        for i, token in enumerate(batch):
            left_context = batch[max(i-window_size,0):i]
            right_context = batch[i+1:min(i+window_size+1, len(batch))]
            context = left_context + right_context
            for ntoken in context:
                pairs.append((token, ntoken))
    return pairs

"""def generate_pairs(self, texts, window_size=2):
    pairs = []
    texts = self.tokenize(texts)
    for text in texts:
        words = [self.word_index[word] for word in text if word in self.
↪word_index]]
        for i, target in enumerate(words):
            start = max(0, i-window_size)
            end = min(len(words), i + window_size + 1)
            for j in range(start, end):
                if i != j:
                    pairs.append((target, words[j]))
    return pairs"""

def generate_data(self, texts, window_size = 2):
    context_indices = self.generate_pairs(texts, window_size)
    lx = [x for x, y in context_indices]
    ly = [y for x, y in context_indices]
    x = np.array(lx)
    vocab_size = len(self.word_index)

```

```

        y_true = np.zeros((len(ly), vocab_size))
        for idx, vy in enumerate(ly):
            y_true[idx, vy] = 1
        return (x, y_true)

import numpy as np
import random as r
class Layer:
    def __init__(self, input_size, output_size):
        self.input_size=input_size
        self.output_size=output_size
        self.x=None#entree de la couche
        self.y=None#sortiedelacouche
        a =np.sqrt(6 /(self.input_size+ self.output_size))
        #self.w=np.random.uniform(-a,a,size=(self.input_size,self.output_size))
        self.w =np.zeros((self.input_size,self.output_size))
        for i in range(self.input_size):
            for j in range(self.output_size):
                self.w[i,j]=r.uniform(-a,a)

class Embedding(Layer):
    def forward(self,x):
        self.x=x.copy()
        #self.y=self.w[x]
        self.y=np.zeros((len(x),self.output_size))
        for i in range(len(x)):
            self.y[i]=self.w[x[i]]
        return self.y
    """oubien
    self.x= x
    self.y = self.w[self.x]
    return self.y """

    def backward(self,e,learning_rate):
        self.y=np.zeros((len(self.x), self.output_size))
        dX=np.zeros((len(e),self.output_size))
        for i, word_index in enumerate(self.x):
            self.w[word_index]-= learning_rate* e[i]
            dX[i]=e[i]
        return dX

class Dense(Layer):
    def __init__(self, input_size, output_size):
        super().__init__(input_size, output_size)
        self.b=np.zeros(output_size)

```

```

def forward(self,x):
    self.x=x.copy()
    z= np.dot(self.x,self.w)+self.b
    exp_z=np.exp(z)
    self.y=exp_z/np.sum(exp_z)
    return self.y

"""ou bien
self.x= x
z =self.x.dot(self.w)+ self.b
#normalisation
z-=z.min()
d =z.max()-z.min()
if d !=0:
    z /=d
#fin normalization
y =np.exp(z)
y =y /np.sum(z, axis=1, keepdims=True)
self.y= y
return self.y"""

def backward(self,e,learning_rate):
    #s=self.Y.flatten()
    #dZ=self.Y*(1-self.Y)-np.dot(self.Y,self.Y.T)
    dZ=e
    d_w =np.dot(self.x.T,dZ)
    d_b =np.zeros(self.output_size)
    for j in range(len(self.x)):
        d_b[j]= np.sum(dZ[:,j])
    self.w-= learning_rate* d_w
    self.b-= learning_rate* d_b
    dX= np.dot(dZ, self.w.T)
    return dX
    """ ou bien
dz =e
dw =self.x.T.dot(dz)
db =dz.sum(axis= 0)
self.w-= learning_rate* dw
self.b-= learning_rate* db
dx =dz.dot(self.w.T)
return dx"""

class NeuralNetwork:
    def __init__(self, learning_rate = 1e-2):
        self.layers = list()
        self.learning_rate= learning_rate

```

```

def add_layer(self,l):
    self.layers.append(l)

def forward(self,x):
    for layer in self.layers:
        x =layer.forward(x)
    return x

def backward(self,e):
    for i in range(len(self.layers)- 1,-1,-1):
        layer = self.layers[i]
        e = layer.backward(e,self.learning_rate)
    return e
"""ou bien
    for layer in reversed(self.layers):
        e = layer.backward(e, self.learning_rate)
    return e"""

def compute_loss(self,y_true, y_pred):
    loss =0
    Batch_size,Vocab_size=y_pred.shape
    for i in range(Batch_size):
        example_loss= 0
        for j in range(Vocab_size):
            example_loss+= y_true[i][j]* np.log(y_pred[i][j])
        loss-= example_loss
    return loss/Batch_size

def compute_accuracy(self,y_true, y_pred):
    batch_size,vocab_size =y_true.shape
    y_pred= y_pred == y_pred.max(axis= 1).reshape(-1,1)
    return 1/batch_size*(y_pred== y_true).sum()
"""def compute_accuracy(self,y_true, y_pred):
    #correct_predictions=np.sum(np.argmax(y_true,axis=1)==np.
    ↪argmax(y_pred,axis=1))
    y=y_pred.copy()
    Batch_size,p=y.shape
    nb=0
    for i in range(Batch_size):
        m=np.max(y[i])
        L=y[i].tolist()
        index=L.index(m)
        y[i]=np.zeros(p)
        y[i][index]=1
        if np.equal(y[i],y_true[i]).all(): nb+=1
    accuracy= nb/Batch_size

```

```

        return accuracy"""

    def fit(self,x,y_true,epochs,batch_size):
        n_samples= x.shape[0]
        loss_list=[]
        accuracy_list=[]
        x_batches= [x[i:i+ batch_size] for i in range(0,n_samples, batch_size)]
        y_batches= [y_true[i:i+ batch_size ]for i in range(0,n_samples,
↪batch_size)]

        for epoch in range(epochs):
            epoch_loss=[]
            epoch_accuracy=[]
            for i in range(len(x_batches)):

                x_batch= x_batches[i]
                y_batch= y_batches[i]
                y_pred= self.forward(x_batch)
                l =self.compute_loss(y_batch,y_pred)
                epoch_loss.append(l)

                a =self.compute_accuracy(y_batch,y_pred)
                epoch_accuracy.append(a)
                error= y_pred-y_batch
                self.backward(error)

            loss_list.append(sum(epoch_loss)/len(epoch_loss))
            accuracy_list.append(sum(epoch_accuracy)/len(epoch_loss))

        return loss_list,accuracy_list

f=open('MonCorpus.txt','r')
corpus=f.readlines()
#corpus=["Installer un logiciel de développement","Configurer un serveur
↪web","Télécharger et configurer un environnement de développement"]
tk= Tokenizer()
window_size,embedding_dim,learning_rate= 2,50,0.01
tk.fit_on_texts(corpus)
vocab_size=len(tk.word_index)
print(vocab_size)
x,y_true=tk.generate_data(corpus)
print(x,y_true)
skipgram=NeuralNetwork(learning_rate)
e_layer=Embedding(vocab_size,embedding_dim)

```

```

skipgram.add_layer(e_layer)
d_layer=Dense(embedding_dim,vocab_size)
skipgram.add_layer(d_layer)
epochs=100
batch_size=10
z,t=skipgram.fit(x,y_true,epochs,batch_size)
import matplotlib.pyplot as plt
l1=np.arange(epochs)
plt.plot(l1,z)
plt.title("Loss and Accuracy over Epochs")
plt.xlabel("Epoch")
plt.ylabel("Value")
plt.legend(["Loss","Accuracy"])
plt.show()
plt.plot(l1,t)
plt.show()

```

```

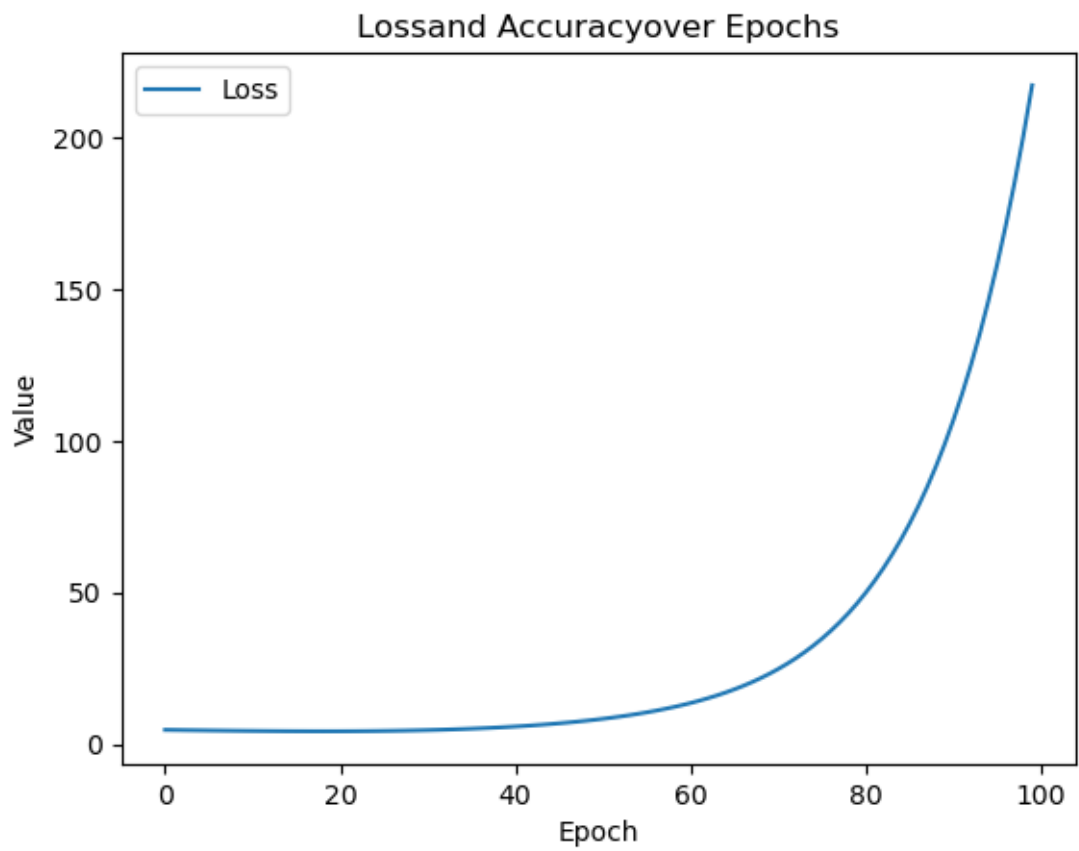
12
[ 1  1  2  2  2  3  3  3  3  4  4  4  5  5  6  6  2  2  2  7  7  7  8  8
  9  9 10 10 10  6  6  6  6  2  2  2  2 11 11 11 11  4  4  4  5  5] [[0. 0. 1.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

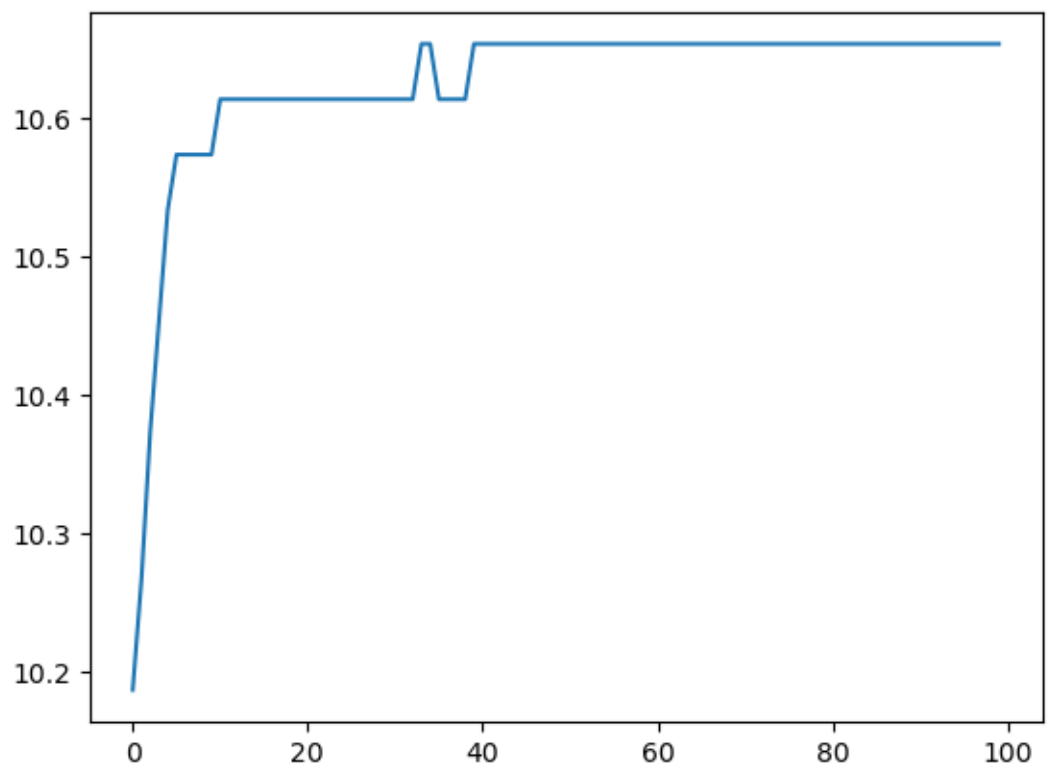
```

```

[0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
[0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
[0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
[0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
[0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
[0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
[0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
[0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
[0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
[0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
[0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
[0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]

```





[]:

[]:

[]:

[]: