

Presentación

En este jupyter notebook va a encontrar una implementación del algoritmo Online Dictionary Learning descrito en el paper [Online dictionary learning for sparse coding](#). Y cuya interpretación y un análisis resumido se puede encontrar en el documento de la primer entrega. Para esta implementación se utilizó como fuente de inspiración el código del siguiente repositorio [repositorio](#) y scripts que se encuentran en el [repositorio del curso] (https://gitlab.fing.edu.uy/tao/datos/-/tree/main?ref_type=heads), principalmente para el uso del dataset del proyecto LUISA.

Además de la implementación, se muestran los resultados para distintos parámetros y mejoras en el algoritmo.

```
In [ ]: # Inspirado en los repositorios
# https://gitlab.fing.edu.uy/tao/datos
# https://github.com/MehdiAbbanaBennani/online-dictionary-learning-for-sparse-coding

# %%
import sys, os
from itertools import tee

import numpy as np
from sklearn.linear_model import Lasso, LassoLars
from tqdm import tqdm
import matplotlib.pyplot as plt
import imageio

sys.path.append(os.path.abspath("/home/pancho/Documents/PhD/FING/TAO/entregable"))

import datos
import util
```

```
In [ ]: data = datos.get_char_luisa()
lambda_reg = 0.001 # Parametro de regularizacion
k = 250 # Cantidad de átomos en el diccionario
n_obs = len(data)
dim_obs = len(data[0])
log_step = 40 # Cada cuantas iteraciones almacenar los datos
test_batch_size = 1000 # Cantidad de muestras para test
losses = []
regret = []
offline_loss = []
objective = []

alphas = []
observed = []
cumulative_losses = []

np.random.seed(14) # semilla para hacer pruebas comparables
```

Aprendizaje de diccionarios

En las siguientes secciones se mostrarán los resultados que se obtienen al variar las inicializaciones de las matrices A, B y el diccionario D.

También se verá el impacto de cambiar entre los algoritmos de optimización Lasso y Lasso-LARS, para los cuales se muestran resultados utilizando distintos valores del parámetro de regularización lambda. En cuanto al tamaño del batch, se utilizó dos valores, se utilizaron batches de un único elemento para seguir el modo más "puro" del método y con batches de tamaño 200 elementos.

Sobre la cantidad de átomos en el diccionario

En la mayoría de los ejemplos que se muestran, la cantidad de átomos del diccionario es 250. Esta cantidad fue seleccionada por capacidad computacional para poder realizar en un tiempo razonable las distintas pruebas. Se buscó la cantidad de átomos fuera amplia y a la vez que fuera analizable al mirar la imagen del diccionario aprendido. Esta condición a la vez que la cantidad de átomos sea mayor a la cantidad de letras del alfabeto, más la cantidad de números y de símbolos. Se especula que hay mayor cantidad de caracteres debido a que en los datos hay distintas "fuentes", caracteres combinados, ejemplo aquellos que tienen tildes, etc. Dado que en estos ejemplos la cantidad de átomos no fue superior a la cantidad características, hay un último ejemplo con un diccionario de 1000 elementos.

Descripción de las inicializaciones

Inicialización del diccionario

Para el diccionario se implementaron dos estrategias. La inicialización "1" es inicializar el diccionario D con valores aleatorios. Es decir, cada átomo es un vector de valores aleatorios y norma unitaria. La otra estrategia (o estrategia "0") es inicializar el diccionario con elementos aleatorios que pertenecen al dataset. De esta forma, cada átomo inicial del diccionario es exactamente un elemento del conjunto de datos.

Inicialización de la matriz de acumulación A

La configuración "1" de inicialización de la matriz de acumulación A consta de inicializar esta matriz con valores aleatorios y que cada columna tenga norma unitaria. La configuración "2" es inicializar esta matriz con el valor constante 0.001. Mientras que la otra configuración, la inicialización "0" es inicializar esta matriz como una matriz estrictamente diagonal con valor 0.001 en su diagonal.

Inicialización de la matriz de acumulación B

Para inicializar esta matriz hay implementadas 2 alternativas. La primer configuración para inicializar esta matriz, la configuración "1", es, al igual que en los dos casos anteriores, cargar la matriz con valores aleatorios y que su columnas tengan norma unitaria. La configuración "2" consisten en cargala con el valor constante 0.001, mientras que la otra configuración, la configuración "0", es cargar la matriz nula.

Procesamiento en Batches

De acuerdo al paper en cuestión, se implementó la posibilidad de realizar procesamiento por batches y no únicamente por un único elemento por iteración. De esta forma se logró reducir drásticamente la cantidad de iteraciones del algoritmo que se requieren. Esto repercute en una reducción del tiempo de entrenamiento. Utilizando el procesamiento por batches, se realizaron una buena cantidad de pruebas que permitió observar como afecta a la eficacia del algoritmo las distintas

Descripción de la Loss

Loss Offline

La loss que se presenta está calculada sobre un conjunto de test que está por fuera del conjunto. Para el calculo se toma promedio del error de reconstruir cada vector del conjunto de test.

Esta Loss no se toma en cuenta para tomar acciones durante el entrenamiento como podría ser realizar un early stopping si se detecta que la Loss deja de decrecer de forma significativa.

Resultados

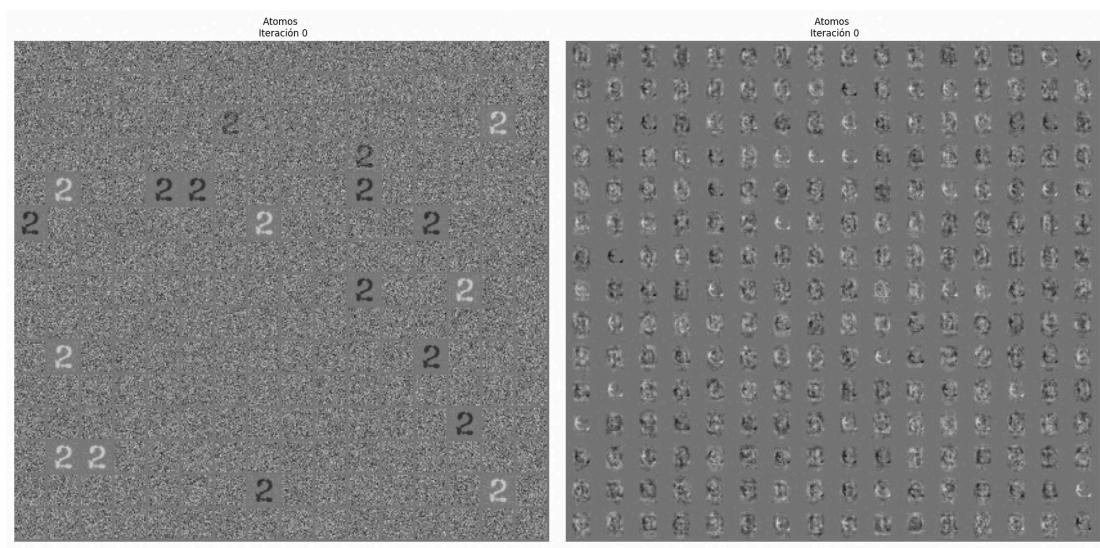
Tabla comparativa de tiempos

Configuración	Init D	Init A	Init B	Optimizador	Lambda	Tamaño del batch	Cantidad de elementos	Tiempo/
1 - base	1	1	0	Lasso-LARS	0.001	1	40000	104ms
2	0	1	0	Lasso-LARS	0.001	1	40000	289ms
3	1	1	0	Lasso-LARS	0.001	200	40000	1140ms
4	1	1	0	Lasso	0.001	200	40000	915ms

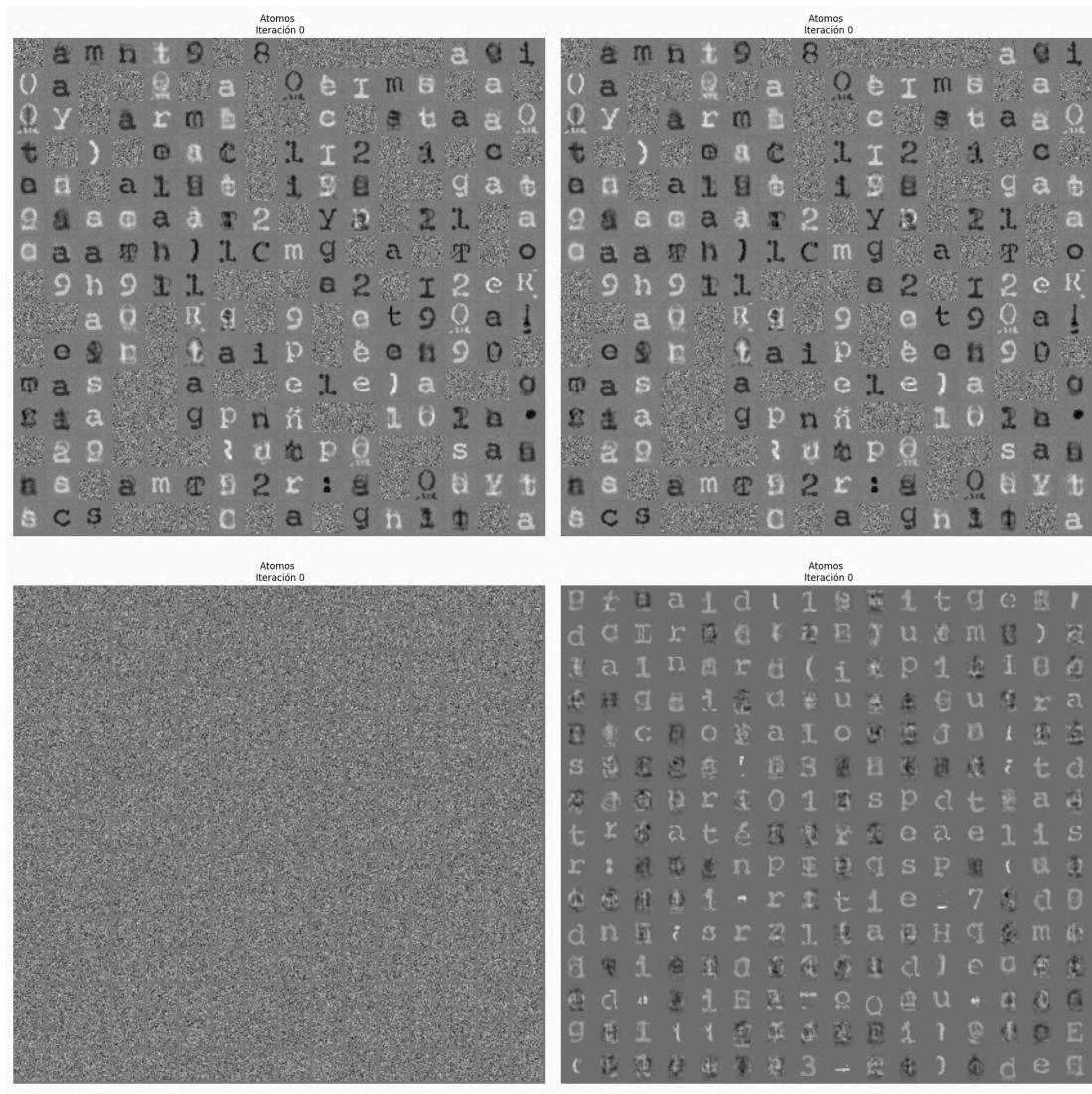
Configuración	Init D	Init A	Init B	Optimizador	Lambda	Tamaño del batch	Cantidad de elementos	Tiempo/
5	0	1	0	Lasso-LARS	0.01	200	40000	790ms
6	1	1	0	Lasso-LARS	0.01	200	40000	705ms
7 k=1000	1	1	0	Lasso-LARS	0.001	200	40000	5330ms

Comparación visual del proceso y graficas de losses.

Proceso de aprendizaje con batches unitarios - 40000 iteraciones

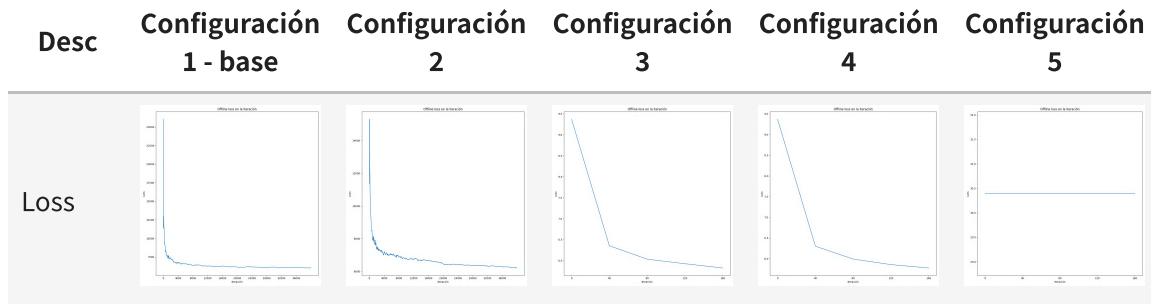


Proceso de aprendizaje con batches de 200 elementos - 200 iteraciones



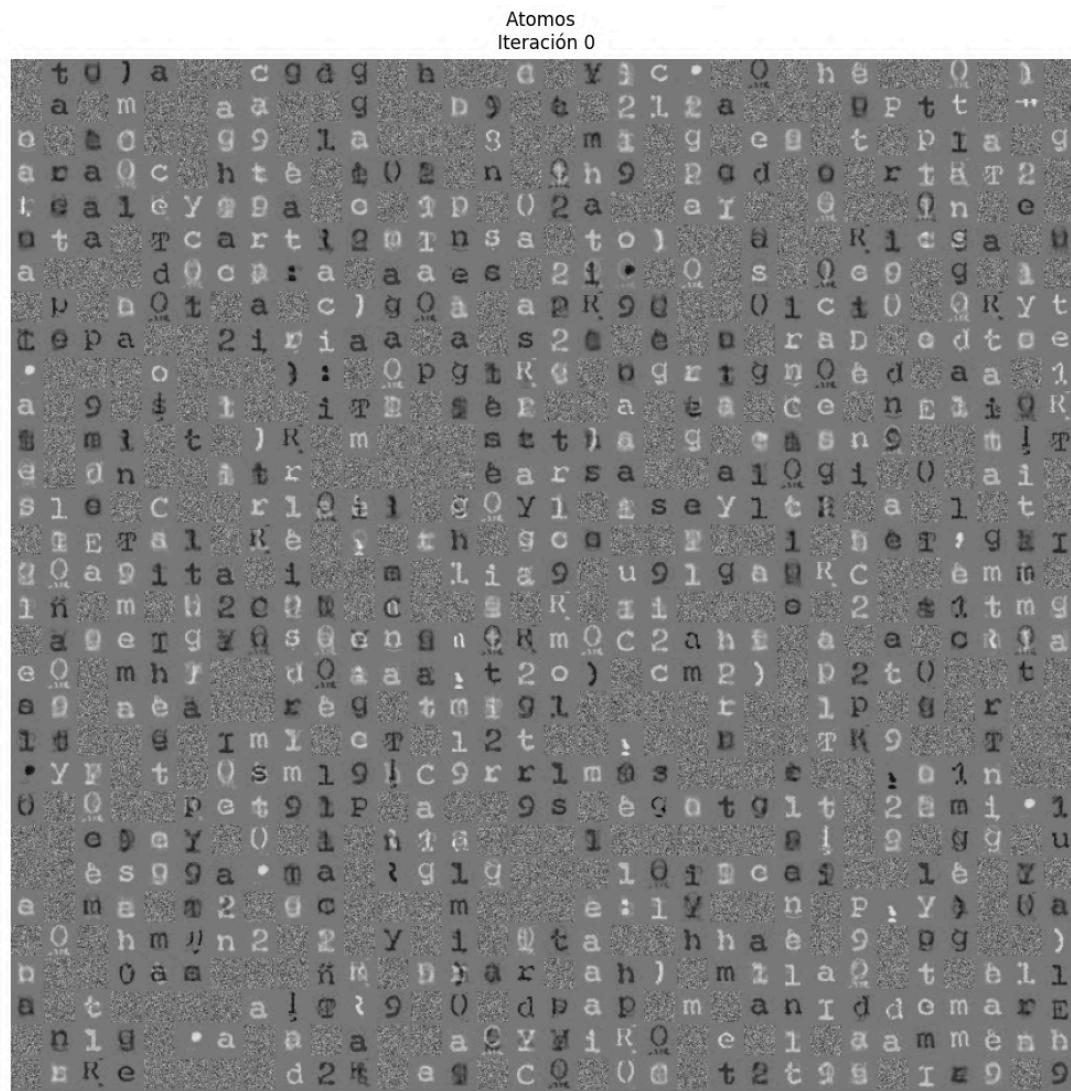
Comparación de frames iniciales, finales y losses

Desc	Configuración 1 - base	Configuración 2	Configuración 3	Configuración 4	Configuración 5
Tiempo dato	104ms	289ms	6ms	5ms	4ms
Frame 1					
Frame final					



Bonus Track - Configuración 7 entrenamiento de un diccionario de 1000 elementos

Diccionario de 1000 elementos obtenido con 200 iteraciones y 200 datos por batch. En este diccionario se puede observar que aún muchos de sus átomos son ruido. Esto indica que es necesario un entrenamiento más prolongado. A la vez que se observa una buena cantidad de caracteres nítidos.



Conclusiones

Se logró implementar un algoritmo para el aprendizaje Online de diccionarios y se logró implementar distintas mejoras que se proponen en la misma publicación. La implementación permitió comprobar de manera empírica los efectos de introducir las mejoras mencionadas en el documento. Si bien hay desarrollos que quedaron como trabajos futuros, la implementación lograda es un buen punto inicial que permite el desarrollo posterior para implementar una nueva tanda de mejoras.

La implementación de algoritmo propuesto, así como las mejoras implementadas, fueron desafiantes y que requirieron de agudeza en el desarrollo.

Análisis de las pruebas realizadas

Las pruebas realizadas dejan de manifiesto que utilizar Lasso como optimizador es más rápido en cada iteración que utilizar Lasso-LARS, sin embargo el resultado final luego de la misma cantidad de iteraciones es superior en el caso de Lasso-LARS. Se aprecia que la correcta elección del parámetro de regularización es relevante para que el algoritmo logre aprender, como se observa en los casos 5 y 6, donde al utilizar un valor de λ grande no se logra un aprendizaje significativo luego de haber utilizado 40000 datos. Cantidad de datos con la cual sí se observa un buen aprendizaje si se utiliza $\lambda = 0.001$ como ocurre en los casos 1,2,3,4 y 7. A su vez, los casos 5 y 6 dejan de manifiesto que inicializar el diccionario con valores de los datos favorece el aprendizaje, pues en el caso 5 no se observa aprendizaje pero si se observa una pequeño aprendizaje en el caso 6. Esto se observa tanto al observar los frames iniciales y finales, así como observando las gráficas de la loss.

Sin embargo la implementación de mejora que se lleva todas las miradas es el procesamiento en batches. La implementación de esta mejora fue ardua, más su resultado permitió la realización de una mayor cantidad de experimentos e incluso realizar pruebas con diccionarios de mayor tamaño. En cuanto a esta mejora es notorio la reducción de tiempo en el procesamiento de cada dato individual, pasando de un promedio de 104ms o 289ms a estar en el orden 4ms, es decir que el procesamiento por batches es entre 26 y 76 veces más rápido para procesar cada muestra.

Los resultados mostrados no permiten aportan información sobre el efecto de las distintas inicializaciones implementadas para las matrices A y B.

Observando las losses y la visualización temporal de los resultados obtenidos, se aprecia que los algoritmos aún tienen capacidad de mejorar los resultados con mayor entrenamientos más largos y con otras mejoras como puede ser el purgado del diccionario.

Trabajo futuro

Purgado del diccionario

El paper propones utilizar una técnica de purgado de forma de detectar atomos que no están siendo utilizados y reinicializarlos, de forma de forzar a que el diccionario aprendido tenga átomos de mejor calidad y más representativos.

Early Stopping

Esta propuesta no se encuentra concretamente en el paper, pero podría ser bueno implementar una versión de Early Stopping que permita detener el entrenamiento si las actualizaciones del diccionario son pequeñas o no hay mejoras en la loss, de forma de no perder tiempo computacional si el algoritmo ya no está mejorando su aprendizaje, e incluso permitir aventurarse a buscar diccionarios con una gran cantidad de iteraciones.

Warm Start

Evaluar utilizar el algoritmo Lasso con Warm Start y evaluar la mejora de velocidad en la convergencia del algoritmo.

Impaintig

Se desea implementar un algoritmo que permita realizar impainting para mejorar la reconstrucción de los caracteres del dataset.

Implementación del código - Clase de python

```
In [ ]: class OnlineDictionaryLearning:
    """
        Aprendizaje en línea de diccionarios para codificación sparsa.

        Esta clase implementa un algoritmo online para aprender un diccionario
        que pueda representar de forma sparsa un conjunto de datos.
    """

    def __init__(
        self,
        data: np.array,
        log_step: int = 40,
        test_batch_size: int = 1000,
        base_dir: str = ".",
    ):
        """
            Inicializa la clase OnlineDictionaryLearning.

            Args:
                data (np.array): Conjunto de datos para el aprendizaje del diccionario.
                log_step (int, optional): Cada cuantas iteraciones se registran los resultados.
                test_batch_size (int, optional): Tamaño del batch de prueba.
                base_dir (str, optional): Directorio base para guardar resultados.
        """

```

```

        self.data_gen = self.sample(data)
        self.n_obs = len(data)
        self.dim_obs = len(data[0])
        self.m = data.shape[1]

        self.log_step = log_step
        self.test_batch_size = test_batch_size

        self.base_dir = base_dir
        self.losses = []
        self.offline_loss = []
        self.objective = []
        self.cumulative_losses = []
        self.imagenes = []
        self.test_batch = iter(())
        np.random.seed(14) # semilla para hacer pruebas comparables
        if not os.path.exists(base_dir):
            os.makedirs(base_dir)

    def sample(self, data: np.array):
        """
        Crea un generador aleatorio de muestras de datos sobre el cual iterar.

        Args:
            data (np.array): Conjunto de datos para muestreo.

        Yields:
            np.array: Una muestra aleatoria del conjunto de datos.
        """
        while True:
            permutation = list(np.random.permutation(self.n_obs))
            for idx in permutation:
                yield data[idx]

    def initialize_logs(self):
        """
        Inicializa las listas para registrar Loss, imágenes y obtiene un iterador.
        """
        self.losses = []
        self.offline_loss = []
        self.imagenes = []
        self.test_batch = iter(
            [next(self.data_gen) for i in range(self.test_batch_size)])
        )

    @staticmethod
    def compute_alpha(x: np.array, dic: np.array, lam, optimizer: str = "lasso"):
        """
        Calcula los coeficientes de representación esparsa para una observación.
        """
        Args:
            x (np.array): Vector de observación.
            dic (np.array): Matriz del diccionario.
            lam (float): Parámetro de regularización.
            optimizer (str, optional): Método de optimización ('lasso' o 'ridge').
        Returns:
            np.array: Coeficientes de representación esparsa.
        Raises:
            ...
        """

```

```

        Exception: Si NO se especifica un optimizador inválido.
"""

if optimizer == "lasso":
    reg = Lasso(alpha=lam)
elif optimizer == "lars":
    reg = LassoLars(alpha=lam)
else:
    raise Exception(
        "Optimizador incorrecto, solo se aceptan 'lasso' (dafault"
    )
reg.fit(X=dic, y=x)
return reg.coef_


@staticmethod
def compute_dic(A: np.array, B: np.array, D: np.array, k: int):
    """
    Actualiza el diccionario utilizando las matrices acumuladas A y B

    Args:
        A (np.array): Matriz de coeficientes acumulada.
        B (np.array): Matriz acumulada de productos de datos.
        D (np.array): Diccionario actual.
        k (int): Número de átomos en el diccionario.

    Returns:
        np.array: Diccionario actualizado.
    """

tolerancia = 1.0e-7
error = 1
o = 0
D_nuevo = D.copy()
# while not converged :
while error > tolerancia and o < 10:
    for j in range(k):
        u_j = (B[:, j] - np.matmul(D, A[:, j])) / A[j, j] + D[:, j]
        D_nuevo[:, j] = u_j / max(np.linalg.norm(u_j), 1)
    D = D_nuevo / np.linalg.norm(D_nuevo, axis=0)
    error = np.linalg.norm(D_nuevo - D, ord="fro")
    o = o + 1
return D


@staticmethod
def compute_A_B(
    A_prev: np.array,
    B_prev: np.array,
    x_i_batch: np.array,
    alphas: np.array,
    beta: int = 1,
):
    """
    Actualiza las matrices A y B utilizadas en la optimización del di

    Args:
        A_prev (np.array): Matriz A acumulada previa.
        B_prev (np.array): Matriz B acumulada previa.
        x_i (np.array): Observación actual.
        alpha_i (np.array): Coeficientes esparsos para la observació
        beta (int, optional): Parámetro de ponderación. Por defecto 1
    """

```

```

    Returns:
        tuple: Matrices A y B actualizadas.
    """

    A_curr = beta * A_prev + sum(
        [np.outer(alpha_i, alpha_i.T) for alpha_i in alphas]
    )
    B_curr = beta * B_prev + sum(
        [np.outer(x_i, alpha_i.T) for x_i, alpha_i in zip(x_i_batch, )
    )
    A_prev = A_curr
    B_prev = B_curr
    return A_curr, B_curr

def learn(
    self,
    it: int,
    lam: float,
    k: int,
    train_batch_size: int = 1,
    optimizer: str = "lasso",
    init_A_mod: int = 1,
    init_B_mod: int = 1,
    init_D_mod: int = 1,
):
    """
    Ejecuta el aprendizaje del diccionario.

    Args:
        it (int): Cantidad de iteraciones.
        lam (float): Parámetro de regularización.
        k (int): Número de átomos en el diccionario.
        train_batch_size (int, optional): Tamaño del batch de entrenamiento.
        optimizer (str, optional): Método de optimización ('lasso' o 'gradient').
        init_A_mod (int, optional): Método de inicialización para la matriz A.
        init_B_mod (int, optional): Método de inicialización para la matriz B.
        init_D_mod (int, optional): Método de inicialización para el vector D.

    Returns:
        np.array: Diccionario aprendido.
    """
    assert train_batch_size >= 1, "train_batch_size tiene que ser >= 1"
    self.initialize_logs()

    # Init A
    if init_A_mod == 1:
        A_prev = np.random.randn(k, k)
        A_prev /= np.linalg.norm(A_prev)
    elif init_A_mod == 2:
        A_prev = 0.001 * np.ones((k, k))
    else:
        A_prev = 0.001 * np.identity(k)

    # Init B
    if init_B_mod == 1:
        B_prev = 0.001 * np.random.randn(self.m, k)
    if init_B_mod == 2:
        B_prev = 0.001 * np.ones((self.m, k))
    else:
        B_prev = np.zeros((self.m, k))

```

```

# Init D
D_prev = self.initialize_dic(k, self.m, self.data_gen, init_D_mod

for it_curr in tqdm(range(it)):
    x_i_batch = [next(self.data_gen) for i in range(train_batch_size)]
    alphas = [
        self.compute_alpha(x_i, D_prev, lam, optimizer=optimizer)
        for x_i in x_i_batch
    ]
    if train_batch_size == 1:
        beta = 1
    elif train_batch_size > 1:
        if (it_curr + 1) + 1 < train_batch_size:
            theta = (it_curr + 1) * train_batch_size
        else:
            theta = train_batch_size**2 + (it_curr + 1) - train_batch_size
        beta = (theta + 1 - train_batch_size) / (theta + 1)
    A_curr, B_curr = self.compute_A_B(
        A_prev, B_prev, x_i_batch, alphas, beta=beta
    )
    D_curr = self.compute_dic(A=A_curr, B=B_curr, D=D_prev, k=k)
    A_prev = A_curr
    B_prev = B_curr
    D_prev = D_curr

    if it_curr % self.log_step == 0:
        self.log(
            observation=x_i_batch[0],
            dictionary=D_curr,
            it=it_curr,
            lam=lam,
            alpha=alphas[0],
        )

self.compute_objective()

mosaic = util.mosaico(np.array(D_curr.T))
plt.figure(figsize=(10, 10))
plt.imshow(mosaic, cmap="gray")
plt.title(f"Atomos \n Final")
plt.axis("off")
plt.tight_layout()
plt.savefig(
    f"{self.base_dir}{os.sep}temp_frame_final.png"
) # Guarda la imagen temporalmente
plt.show()
plt.close()
self.imagenes.append(
    f"{self.base_dir}{os.sep}temp_frame_final.png"
) # Agrega el nombre a la lista

x = np.arange(0, len(self.losses), max(len(self.losses)) // 10, 1)
xticks_labels = x * self.log_step

plt.figure(figsize=(10, 10))
plt.plot(self.cumulative_losses)
plt.title(f"Loss acumulada")
plt.xticks(x, xticks_labels)
plt.xlabel("Iteración")

```

```

plt.ylabel("Loss")
plt.tight_layout()
plt.savefig(
    f"{self.base_dir}{os.sep}Loss_acumulada.png"
) # Guarda la imagen temporalmente
plt.show()
plt.close()

plt.figure(figsize=(10, 10))
plt.plot(self.losses)
plt.xticks(x, xticks_labels)
plt.title(f"Loss en la iteración")
plt.xlabel("Iteración")
plt.ylabel("Loss")
plt.tight_layout()
plt.savefig(
    f"{self.base_dir}{os.sep}Loss_por_iteracion.png"
) # Guarda la imagen temporalmente
plt.show()
plt.close()

plt.figure(figsize=(10, 10))
plt.plot(self.offline_loss)
plt.xticks(x, xticks_labels)
plt.title(f"Offline loss en la iteración")
plt.xlabel("Iteración")
plt.ylabel("Loss")
plt.tight_layout()
plt.savefig(
    f"{self.base_dir}{os.sep}OfflineLoss_por_iteracion.png"
) # Guarda la imagen temporalmente
plt.show()
plt.close()

with imageio.get_writer(
    f"{self.base_dir}{os.sep}proceso_diccionario_its-{it}_lam-{la
    mode="I",
    duration=0.5,
) as writer:
    for imagen in self.imagenes:
        frame = imageio.imread(imagen)
        writer.append_data(frame)
return D_curr.T

def log(
    self,
    observation: np.array,
    dictionary: np.array,
    it: int,
    lam: float,
    alpha: np.array,
):
    """
    Registra el estado actual, incluyendo losss e imágenes.

    Args:
        observation (np.array): Observación actual.
        dictionary (np.array): Diccionario actual.
        it (int): Iteración actual.
        lam (float): Parámetro de regularización.
    """

```

```

        alpha (np.array): Coeficientes sparsos de la observación.
"""

loss = self.one_loss(observation, dictionary, alpha)
self.losses.append(loss)
self.cumulative_losses.append(self.cumulative_loss())
self.offline_loss.append(self.full_dataset_loss(dictionary, lam))
image_path = f"{self.base_dir}{os.sep}temp_frame_{it}.png"
mosaic = util.mosaico(np.array(dictionary.T))
plt.figure(figsize=(10, 10))
plt.imshow(mosaic, cmap="gray")
plt.axis("off")
plt.title(f"Atomos \n Iteración {it}")
plt.tight_layout()
plt.savefig(image_path) # Guarda la imagen temporalmente
plt.close()
self.imagenes.append(image_path) # Agrega el nombre a la lista

def cumulative_loss(self):
"""
Calcula la loss acumulativa para las muestras observadas.

Args:
    dictionary (np.array): Diccionario actual.

Returns:
    float: Valor de loss acumulada.
"""
n_observed = len(self.losses)

return np.mean([self.losses[i] for i in range(n_observed)])


@staticmethod
def one_loss(x, dictionary: np.array, alpha: np.array):
"""
Calcula la loss de reconstrucción para una observación.

Args:
    x (np.array): Observación.
    dictionary (np.array): Diccionario.
    alpha (np.array): Coeficientes dispersos.

Returns:
    float: loss de reconstrucción.
"""
return np.linalg.norm(x - np.matmul(dictionary, alpha), ord=2) ** 2


@staticmethod
def initialize_dic(k: int, m: int, data_gen, init_D_mod: int = 0):
"""
Inicializa la matriz del diccionario.

Args:
    k (int): Número de átomos en el diccionario.
    m (int): Dimensión de las observaciones.
    data_gen (generator): Generador para muestreo de observaciones.
    init_D_mod (int, optional): Método de inicialización. Por defecto es 0.

Returns:
    np.array: Matriz del diccionario inicializada.
"""

```

```

if init_D_mod == 1:
    D = np.random.randn(m, k)
    return D / np.linalg.norm(D, axis=0)
else:
    return np.array([next(data_gen) for _ in range(k)]).T

def observation_loss(self, x_i: np.array, dictionary: np.array, lam: float):
    """
    Calcula la loss para una observación.

    Args:
        x_i (np.array): Observación.
        dictionary (np.array): Diccionario.
        lam (float): Parámetro de regularización.

    Returns:
        float: loss de la observación.
    """
    alpha = self.compute_alpha(x_i, dictionary, lam)
    return np.linalg.norm(x_i - np.matmul(dictionary, alpha), ord=2)

def full_dataset_loss(self, dictionary: np.array, lam: float):
    """
    Calcula la loss total para el conjunto de datos de prueba.

    Args:
        dictionary (np.array): Diccionario actual.
        lam (float): Parámetro de regularización.

    Returns:
        float: Pérdida total del conjunto de datos.
    """
    self.test_batch, data_gen = tee(self.test_batch)
    return np.mean([
        self.observation_loss(next(data_gen), dictionary, lam)
        for _ in range(self.test_batch_size)
    ])

def compute_objective(self):
    """
    Calcula el valor de la función objetivo a lo largo de las iteraciones.

    La función objetivo es la loss acumulativa promedio en cada iteración.

    Args:
        dictionary (np.array): Diccionario actual.
        lam (float): Parámetro de regularización.

    Returns:
        float: Valor de la función objetivo.
    """
    cumulateD_loss = np.cumsum(self.losses)
    self.objective = [
        cumulateD_loss[i] / (i + 1) for i in range(len(cumulateD_loss))
    ]

```

Ejemplos de procesamiento

Cargar base de datos de caracteres LUISA.

In []: luisa = datos.get_char_luisa()

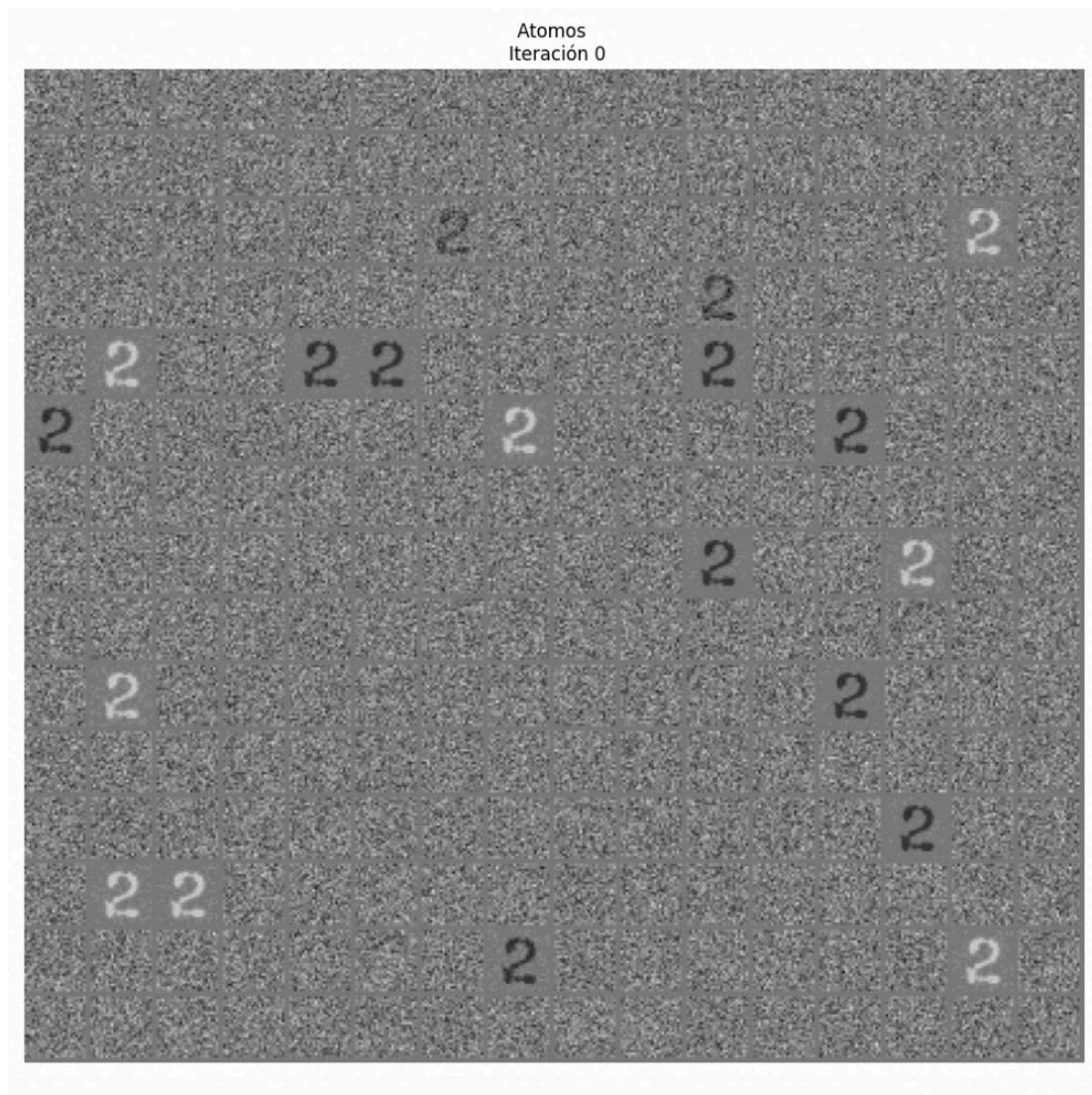
Set configuración general

```
In [ ]: paths = []
dict_size = 250
```

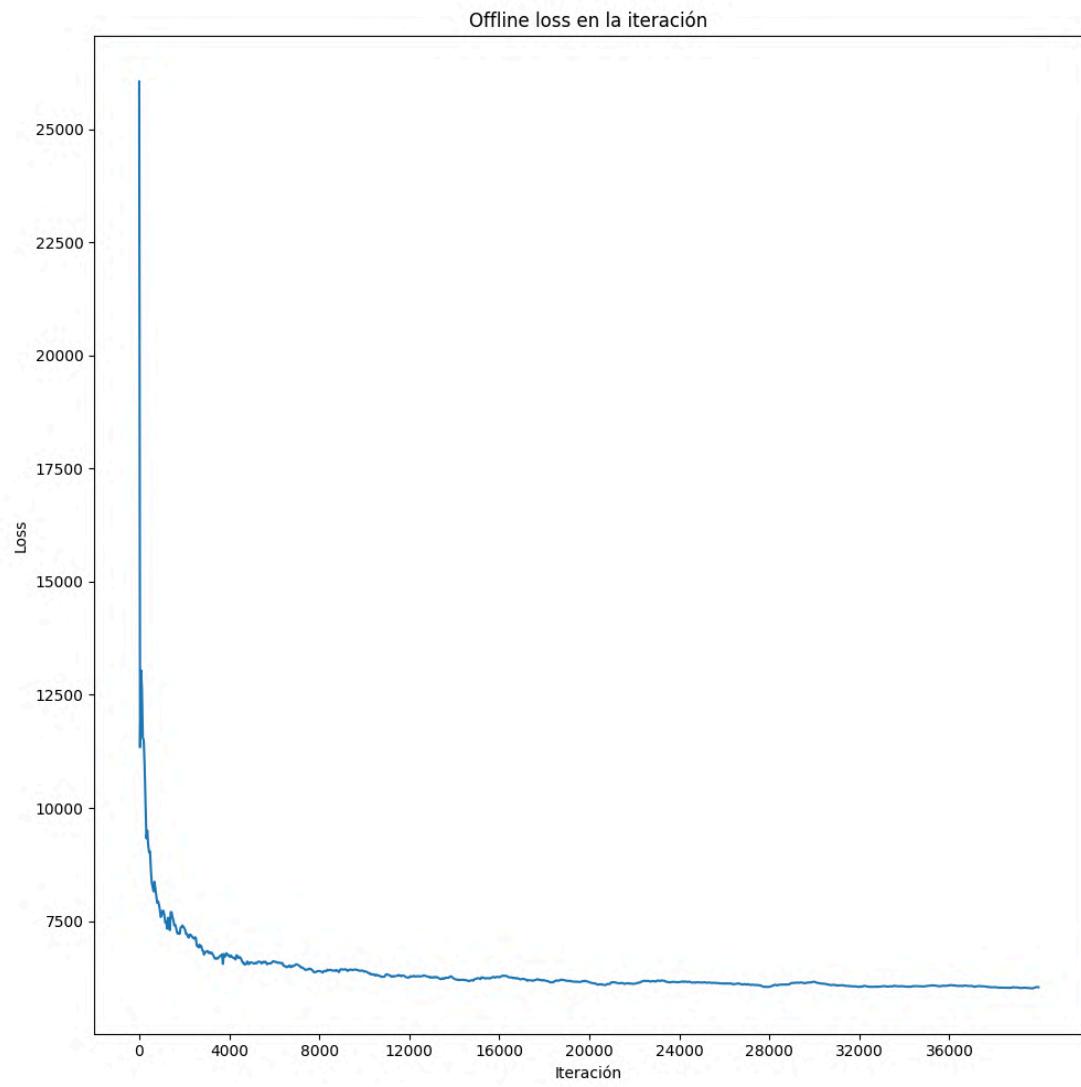
Configuración 1 (base)

Como configuración base se tomó la inicialización "1" para el diccionario D. Es decir con valores aleatorios, como se describió en la sección anterior. La matriz de acumulación A también está inicializada con valores aleatorios con norma 1 en las columnas, con la inicialización "1". Y la matriz B fue inicializada con la inicialización "0", con una matriz nula. En ésta configuración de base se utilizó un batch size de entrenamiento con un único elemento, y como optimizare Lasso-Lars con parámetro de regularización $\lambda = 0.001$.

Gif del entrenamiento

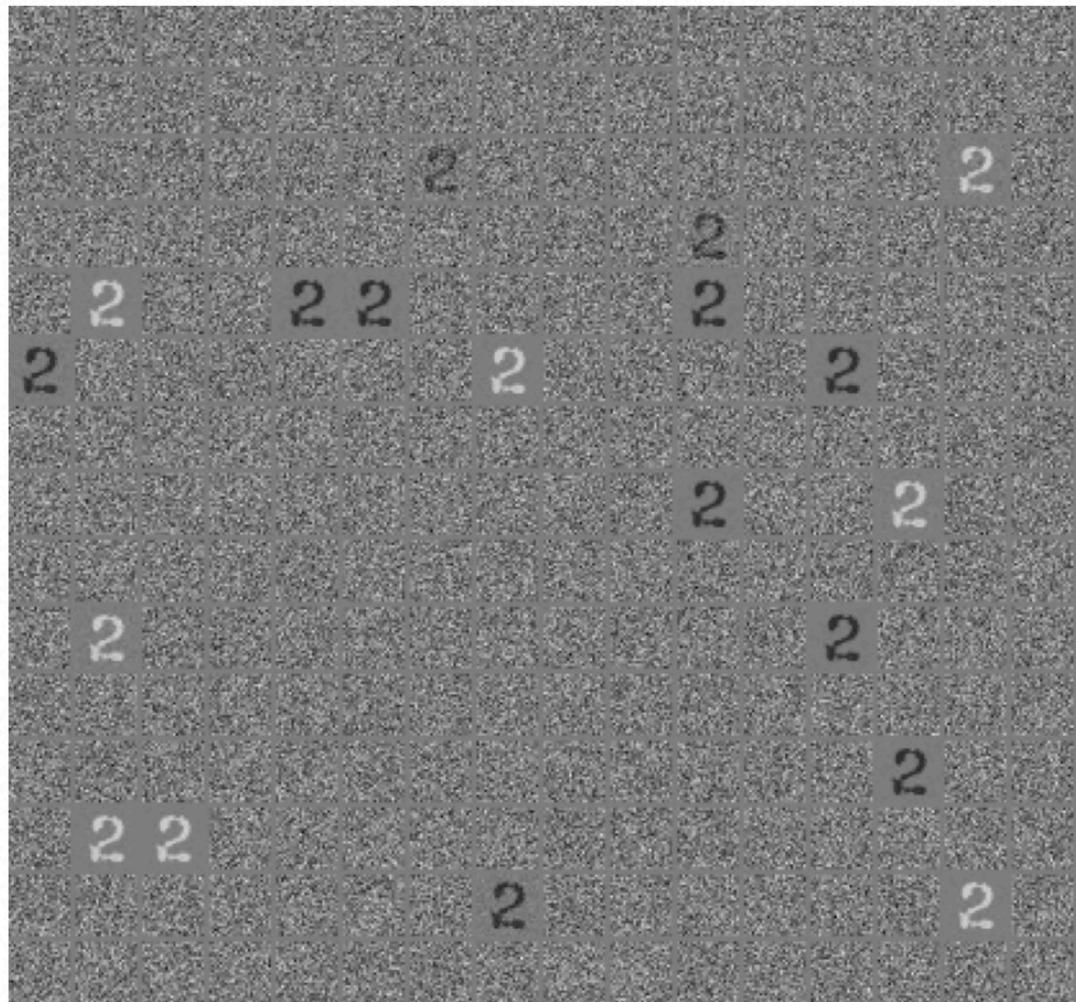


Graficas Loss

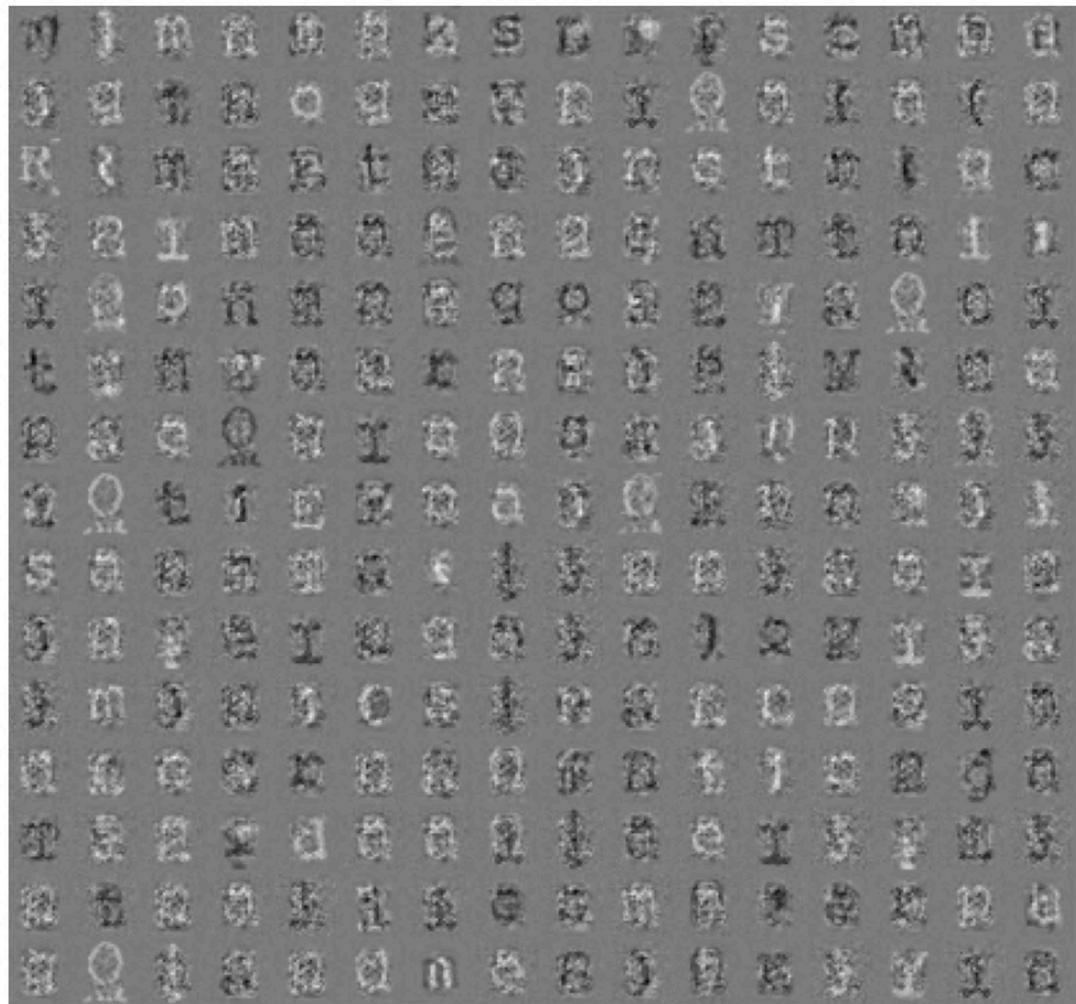


Primer iter, 18k iter, 40k iter

Atomos
Iteración 0



Atomos
Iteración 200



Atomos
Final

J J m b a R 3 s 6 0 / s c R D u
a t S o a n Y p L H a f n T p
E g m x S M o J H c t r e g o
1 9 l x i n u n g a a m t n i i
i d o n T D R g e i 2 d E g o
t N o o u r o : i s i y i S a
G a e t B A u T a v s (n , j 1
n S t i B E u a m a l R z d l l
s a s e v a . i N 9 6 1 u s i a
g d . e t c : o 3 6 1 v i o c
l o) - l d e i s p R E g a I l
d L e e r o m , i s 6 ! o 2 j A
1 z E a d s n A H 4 e o A _ i M
d s c a d n i f e E G 3 , s C p o
g C e s x d n o 8) n a l Z i b

Tiempo de ejecución

Este entrenamiento se completó en 1h 09m 33s. Es decir que consumió 104ms en cada iteración.

Cantidad de datos utilizados

En este entrenamiento se utilizaron 40.000 datos en batches de un único elemento.

Algoritmo para entrenar el diccionario Base

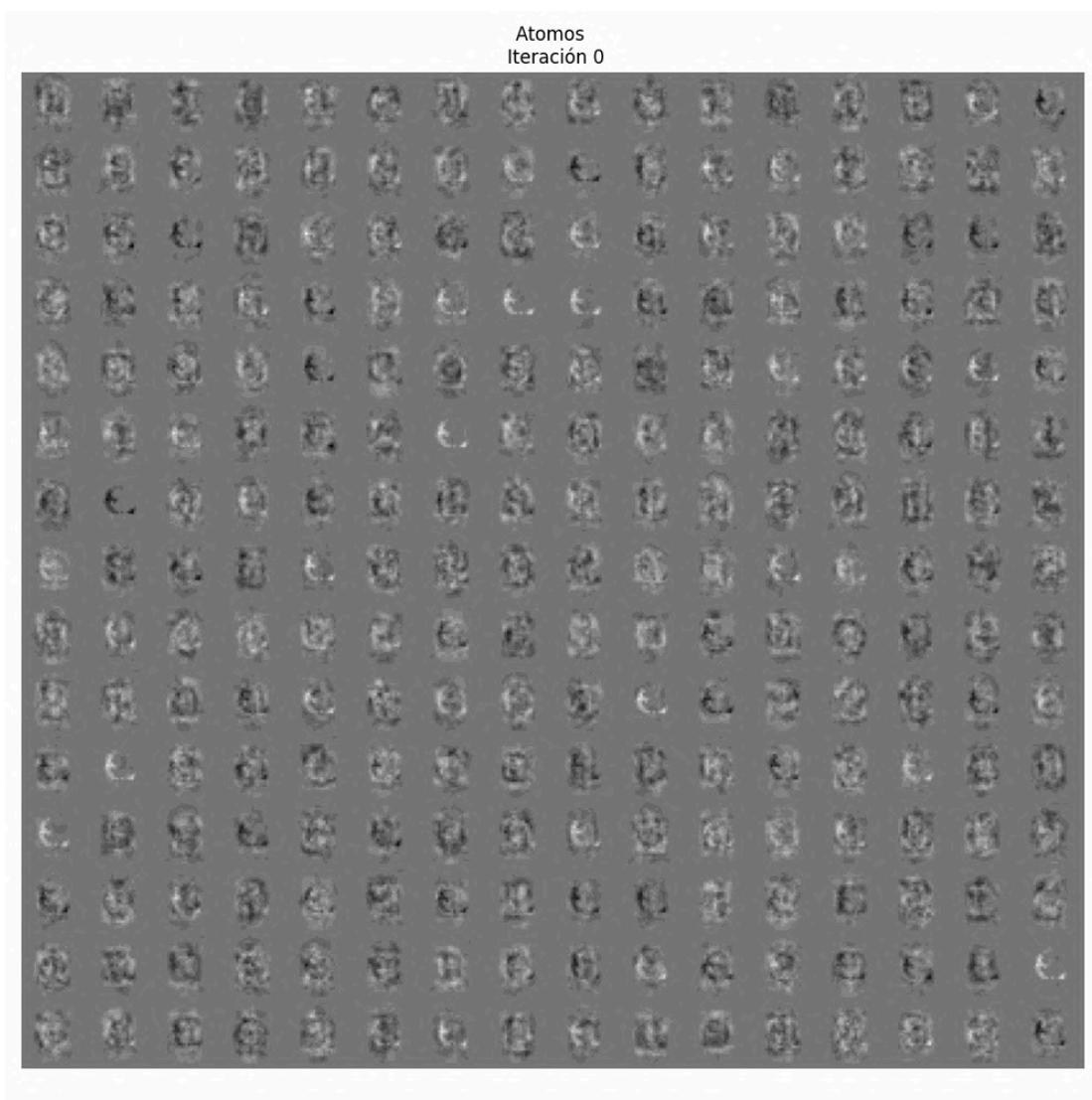
```
In [ ]: conf = {
    "a": 1,
    "b": 0,
    "d": 1,
    "opt": "lars",
    "lamd": 0.001,
    "train_b_s": 1,
    "iteraciones": 40000,
    "dict_size": 250,
}
base_dir = f"dict-{conf['dict_size']}_{conf['iteraciones']}_{conf['a']}_{conf['b']}_{conf['d']}"
ODL = OnlineDictionaryLearning(
```

```
luisa,
test_batch_size=1000,
base_dir=base_dir,
)
D = ODL.learn(
    it=conf["iteraciones"],
    lam=conf["lamd"],
    k=conf["dict_size"],
    optimizer=conf["opt"],
    init_A_mod=conf["a"],
    init_B_mod=conf["b"],
    init_D_mod=conf["d"],
    train_batch_size=conf["train_b_s"],
)
gif_name = f"proceso_diccionario_its-{conf['iteraciones']}_lam-{conf['lam']}_{conf['dict_size']}"
paths.append(f"{base_dir}/{gif_name}")
```

Configuración 2

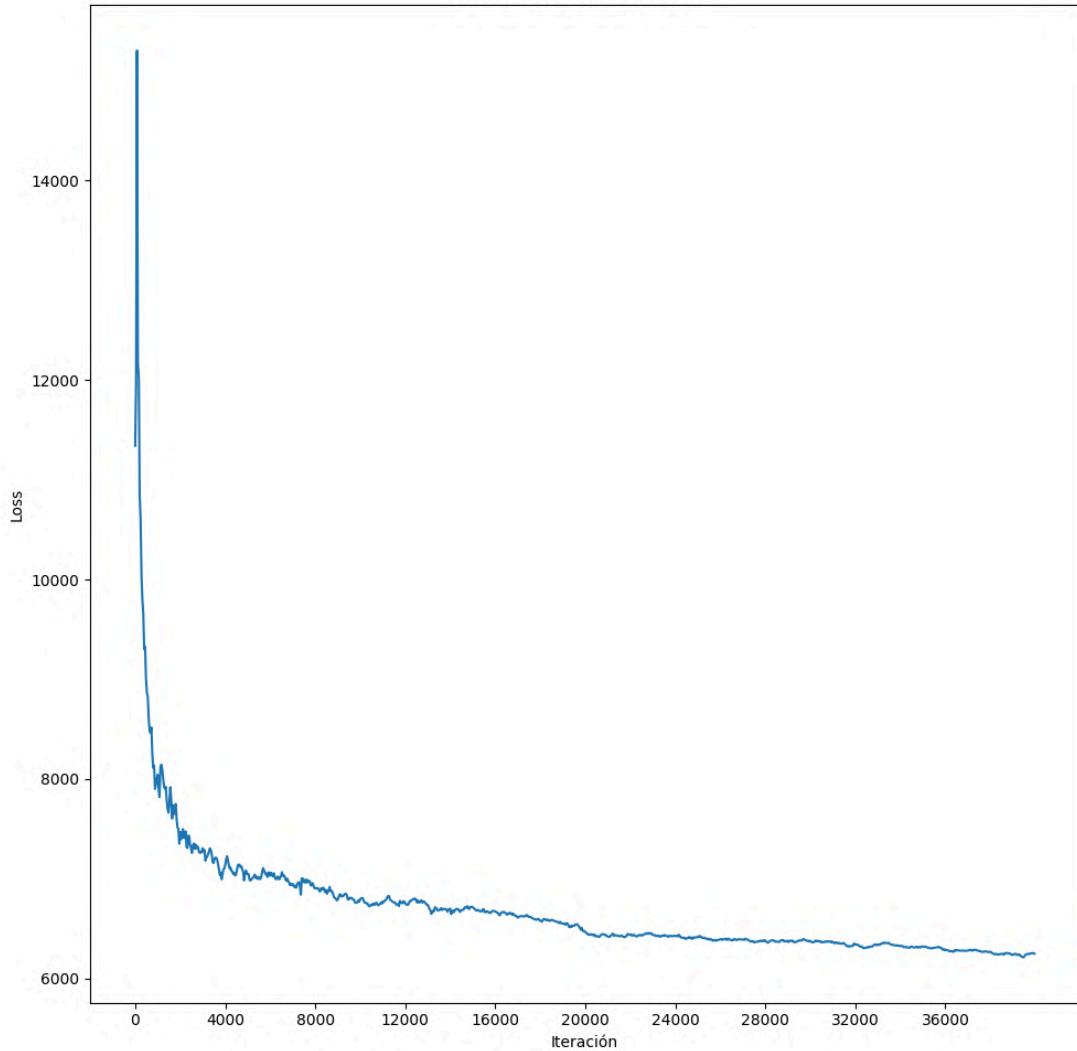
La diferencia con el caso base está en la inicialización de los átomos del diccionario. En esta configuración fue inicializado con la inicialización "0", Es que cada átomo fue inicializado con un elemento al azar del dataset.

Gif del entrenamiento



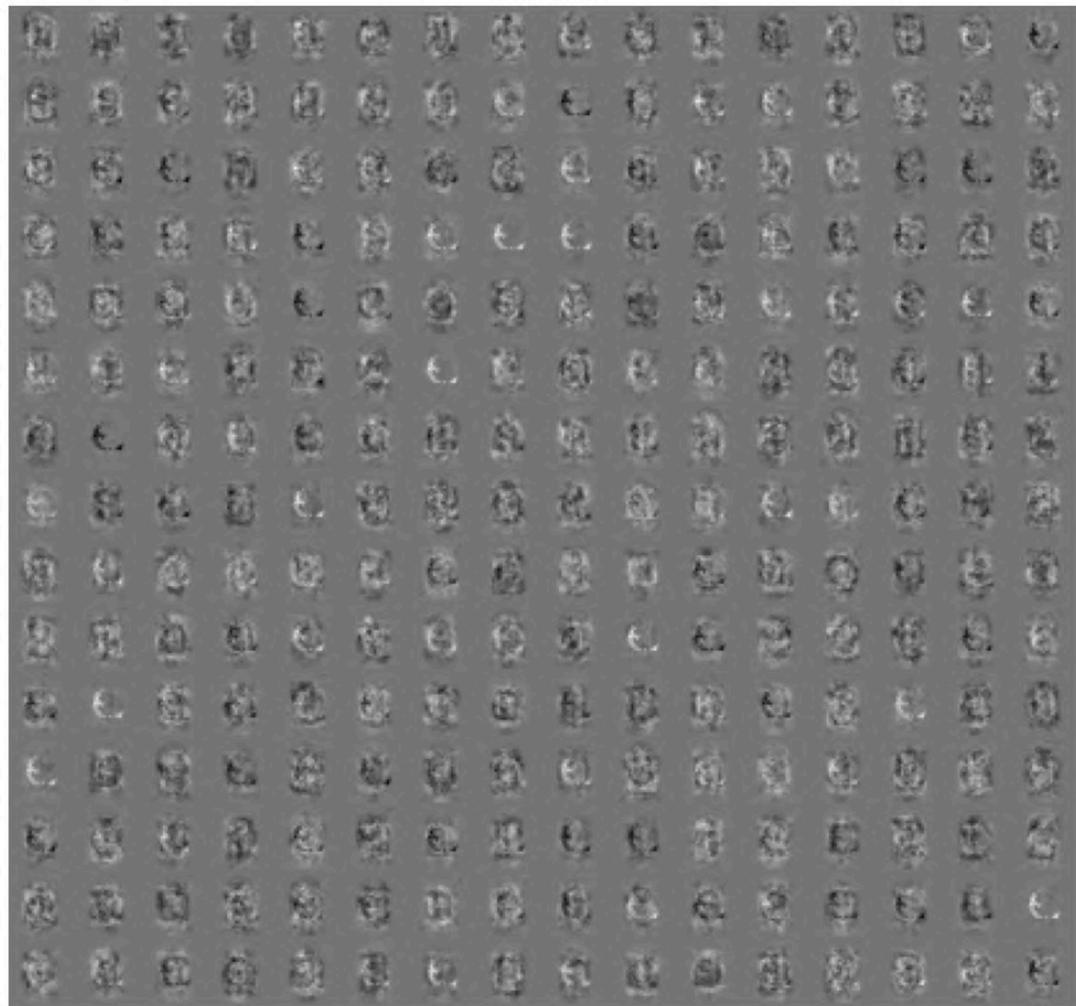
Graficas Loss

Offline loss en la iteración



Primer iter, 18k iter, 40k iter

Atomos
Iteración 0



2-frame5k.png 2-frame18k.png

Atomos
Final



Tiempo de ejecución

Este entrenamiento se completó en 3h 12m 44s. Es decir que consumió 289ms en cada iteración.

Cantidad de datos utilizados

En este entrenamiento se utilizaron 40.000 datos en batches de un único elemento.

Algoritmo para entrenar algoritmo 2

```
In [ ]: conf = {
    "a": 1,
    "b": 0,
    "d": 0,
    "opt": "lars",
    "lamd": 0.001,
    "train_b_s": 1,
    "iteraciones": 40000,
    "dict_size": 250,
}
base_dir = f"dict-{conf['dict_size']}_its-{conf['iteraciones']}_a-{conf['
ODL = OnlineDictionaryLearning(
```

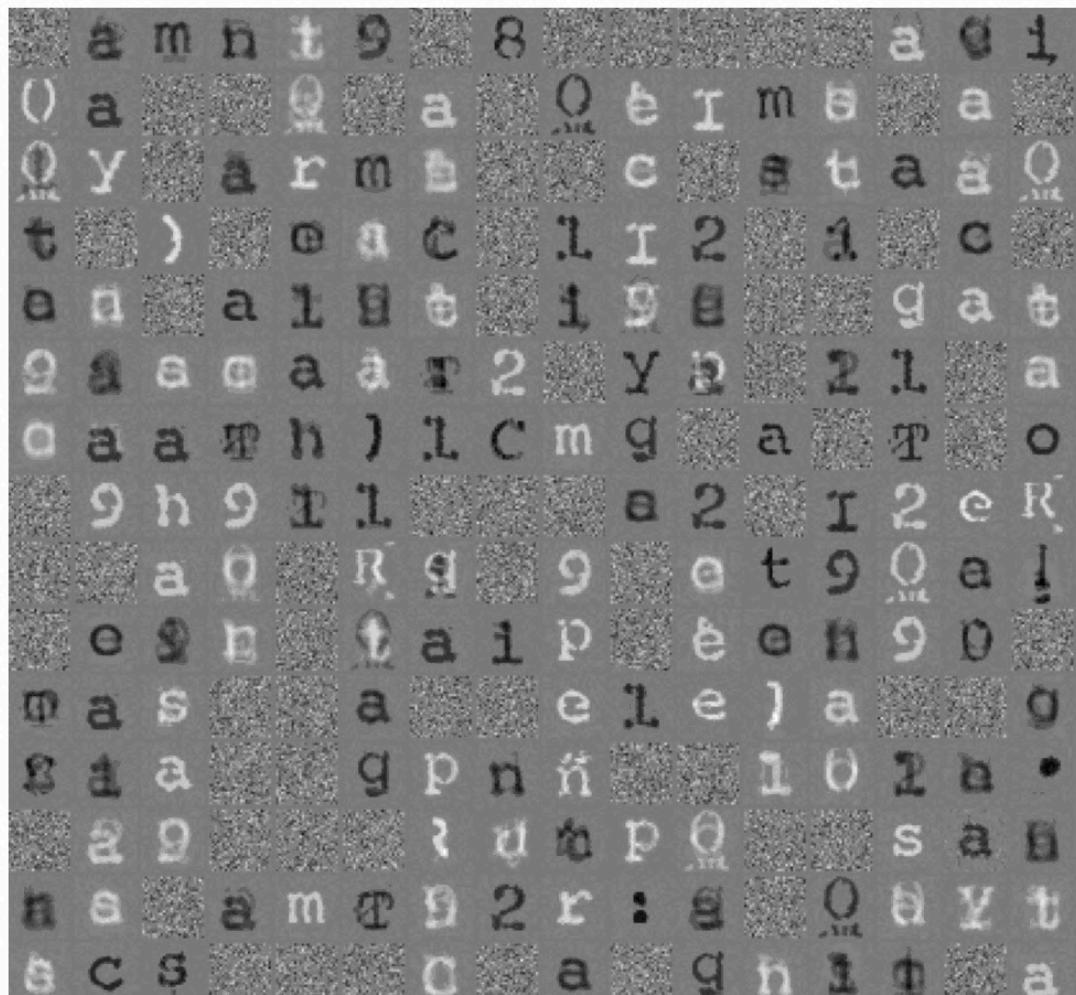
```
luisa,
test_batch_size=1000,
base_dir=base_dir,
log_step=conf["log"],
)
D = ODL.learn(
    it=conf["iteraciones"],
    lam=conf["lamd"],
    k=conf["dict_size"],
    optimizer=conf["opt"],
    init_A_mod=conf["a"],
    init_B_mod=conf["b"],
    init_D_mod=conf["d"],
    train_batch_size=conf["train_b_s"],
)
gif_name = f"proceso_diccionario_its-{conf['iteraciones']}_{lam}-{conf['lam']}"
paths.append(f"{base_dir}/{gif_name}")
```

Configuración 3

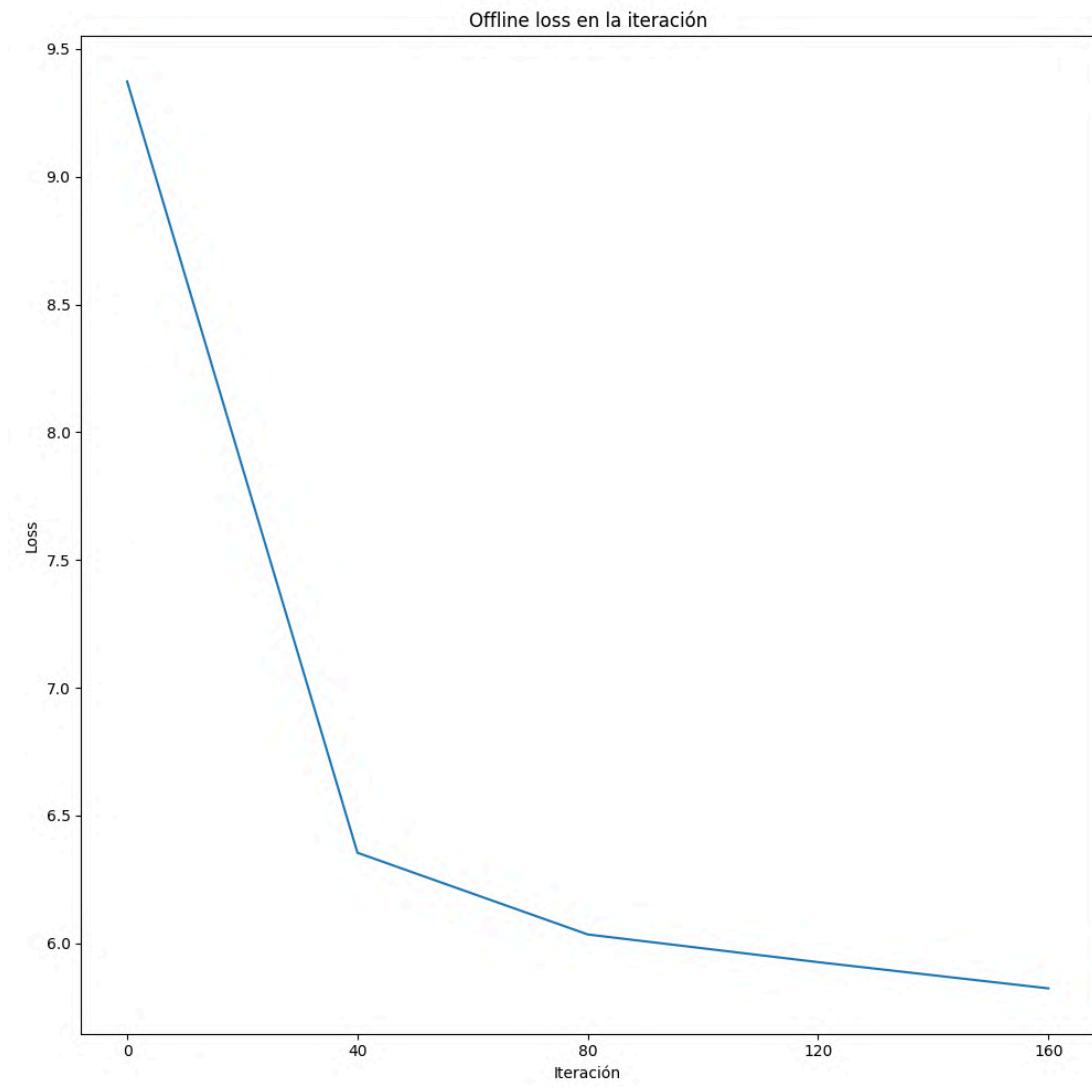
La diferencia con el caso base está en el tamaño del batch y la cantidad de iteraciones. En este caso, se utilizaron 200 iteraciones y 200 muestras en batch. El log se realiza cada 10 iteraciones. En las siguientes configuraciones también se utilizará esta cantidad de iteraciones y el tamaño del batch que se utilizará.

Gif del entrenamiento

Atomos Iteración 0

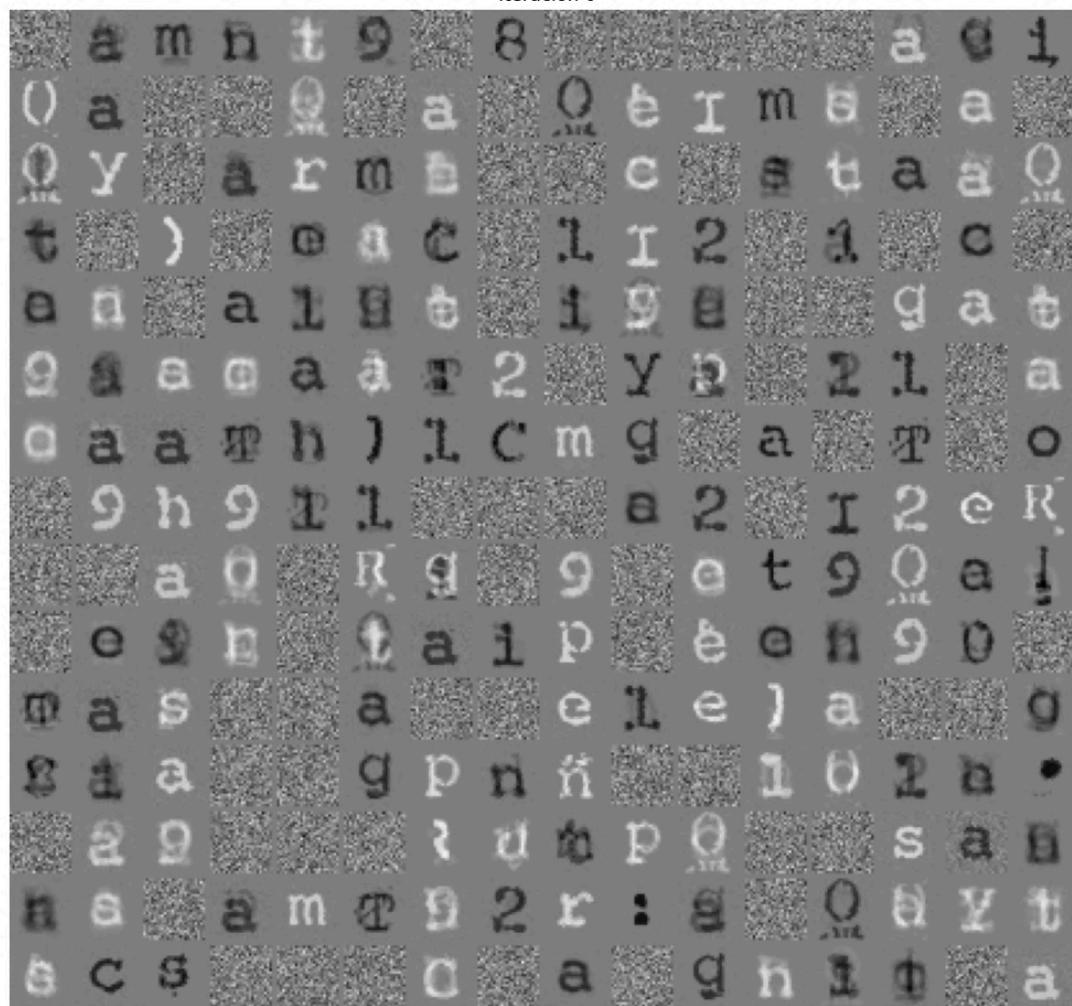


Graficas Loss



Primer iter, 18k iter, 40k iter

Atomos Iteración 0



Atomos
Final



Tiempo de ejecución

Este entrenamiento se completó en 3m 48s. Es decir que consumió en promedio 6ms en procesar cada dato.

Cantidad de datos utilizados

En este entrenamiento se utilizaron 40.000 datos en batches de 200 elementos.

Algoritmo de entrenamiento de la configuración 3

```
In [ ]: conf = {
    "a": 1,
    "b": 0,
    "d": 1,
    "opt": "lars",
    "lamd": 0.001,
    "train_b_s": 200,
    "iteraciones": 200,
    "log": 10,
    "dict_size": 250,
}
```

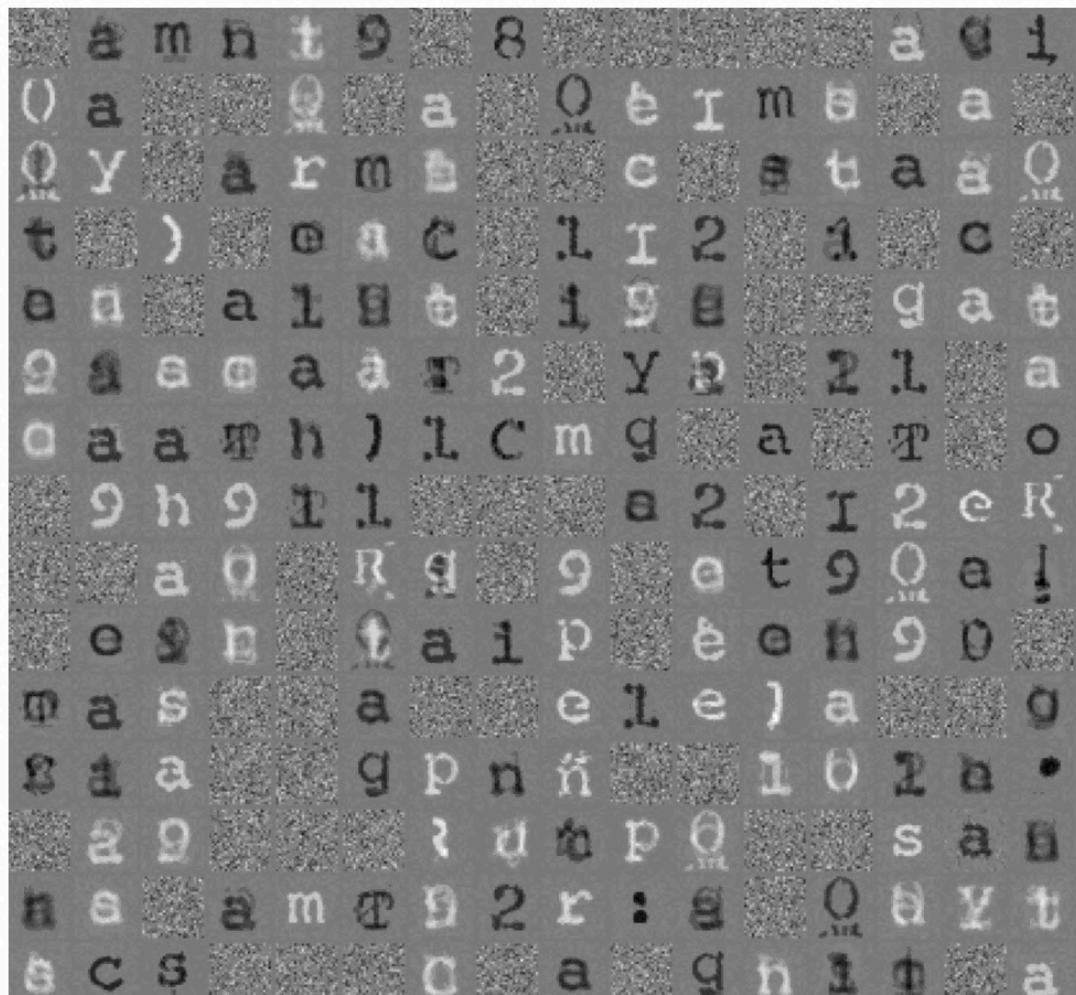
```
base_dir = f"dict-{conf['dict_size']}_{conf['iteraciones']}_{a}-{conf['ODL']}_luisa"
ODL = OnlineDictionaryLearning(
    test_batch_size=1000,
    base_dir=base_dir,
    log_step=conf["log"],
)
D = ODL.learn(
    it=conf["iteraciones"],
    lam=conf["lamd"],
    k=conf["dict_size"],
    optimizer=conf["opt"],
    init_A_mod=conf["a"],
    init_B_mod=conf["b"],
    init_D_mod=conf["d"],
    train_batch_size=conf["train_b_s"],
)
gif_name = f"proceso_diccionario_its-{conf['iteraciones']}_{lam}-{conf['lam']}"
paths.append(f"{base_dir}/{gif_name}")
```

Configuración 4

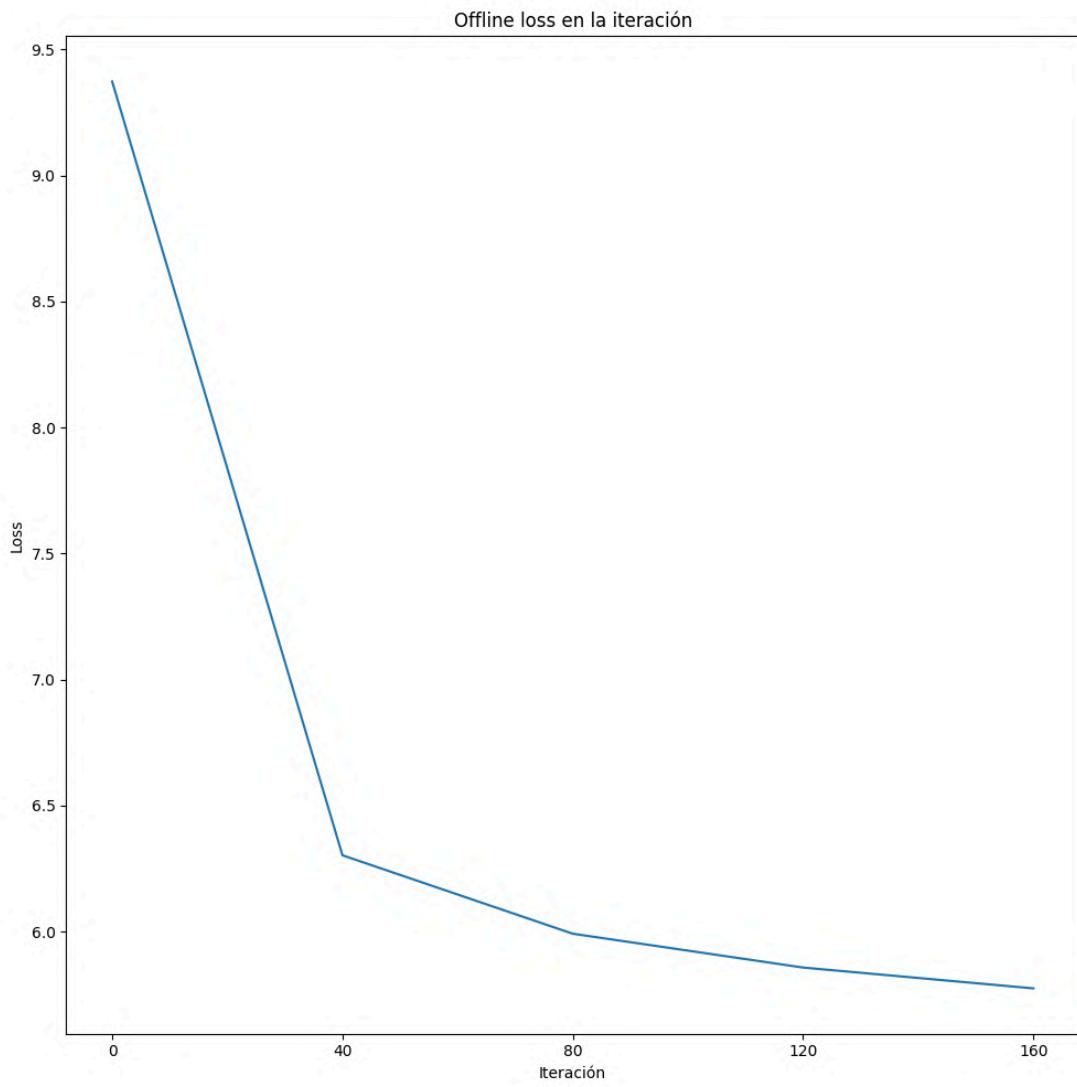
Es el algoritmo de optimización. Se utiliza Lasso en lugar de Lasso-LARS.

Gif del entrenamiento

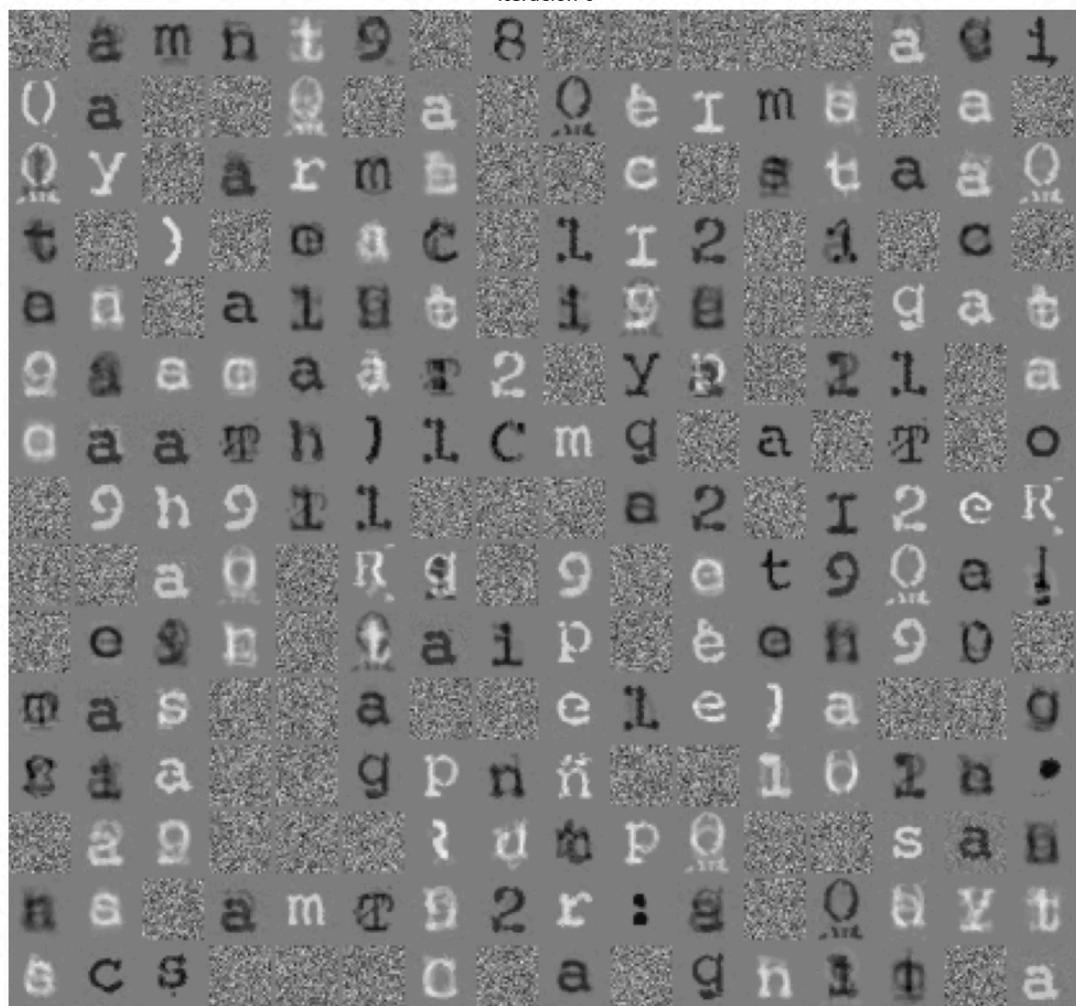
Atomos Iteración 0



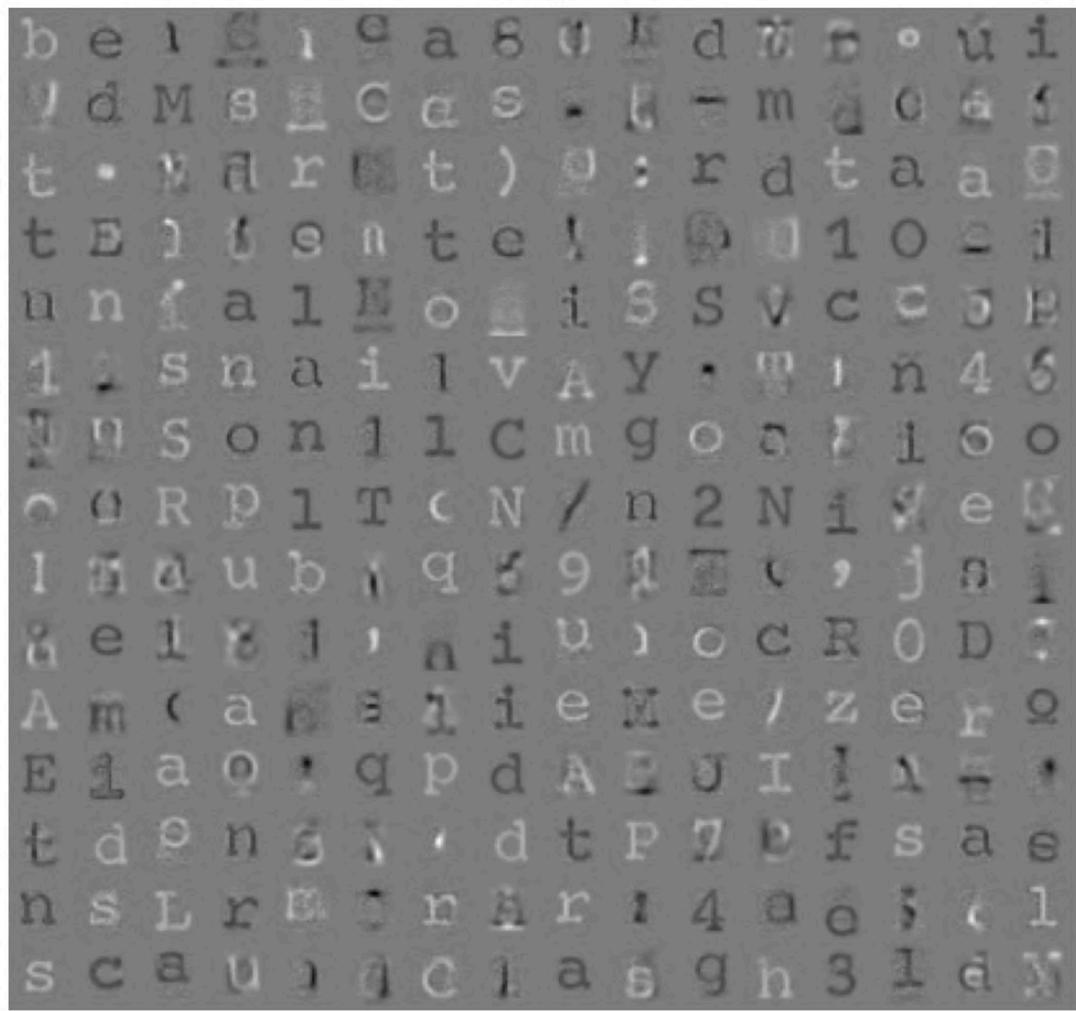
Graficas Loss



Primer iter, 18k iter, 40k iter

Atomas
Iteración 0

Atomos
Final



Tiempo de ejecución

Este entrenamiento se completó en 3m 03s. Es decir que consumió 5ms en procesar cada dato.

Cantidad de datos utilizados

En este entrenamiento se utilizaron 40.000 datos en batches de 200 elementos.

Algoritmo para entrenar configuración 4

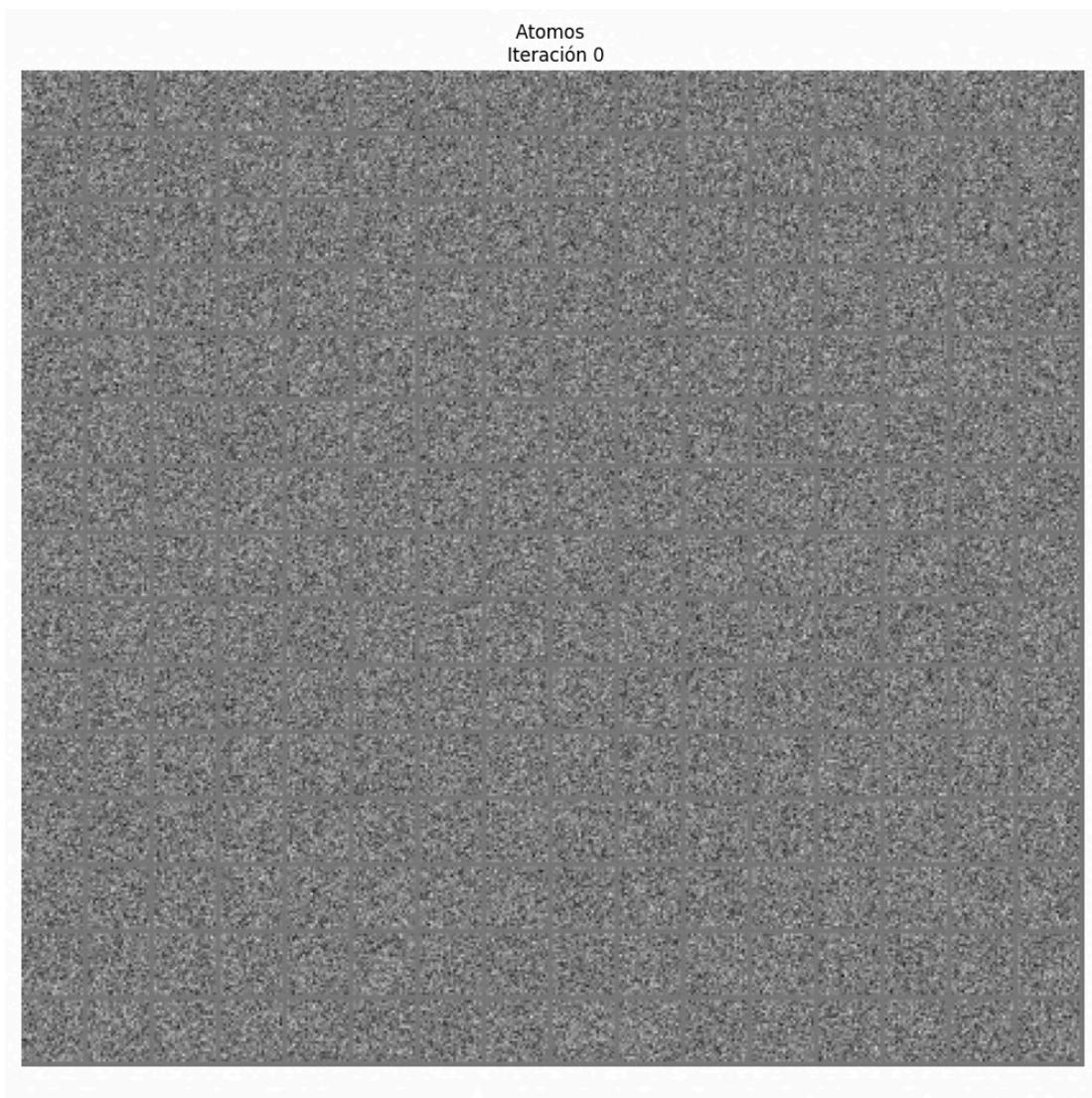
```
In [ ]: conf = (
    {
        "a": 1,
        "b": 0,
        "d": 1,
        "opt": "lasso",
        "lamd": 0.001,
        "train_b_s": 200,
        "iteraciones": 200,
        "log": 10,
        "dict_size": 250,
    },
```

```
)  
  
base_dir = f"dict-{conf['dict_size']}_{its}-{conf['iteraciones']}_{a}-{conf['  
ODL = OnlineDictionaryLearning(  
    luisa,  
    test_batch_size=1000,  
    base_dir=base_dir,  
    log_step=conf["log"],  
)  
D = ODL.learn(  
    it=conf["iteraciones"],  
    lam=conf["lamd"],  
    k=conf["dict_size"],  
    optimizer=conf["opt"],  
    init_A_mod=conf["a"],  
    init_B_mod=conf["b"],  
    init_D_mod=conf["d"],  
    train_batch_size=conf["train_b_s"],  
)  
gif_name = f"proceso_diccionario_its-{conf['iteraciones']}_{lam}-{conf['lam']}  
paths.append(f"{base_dir}/{gif_name}")
```

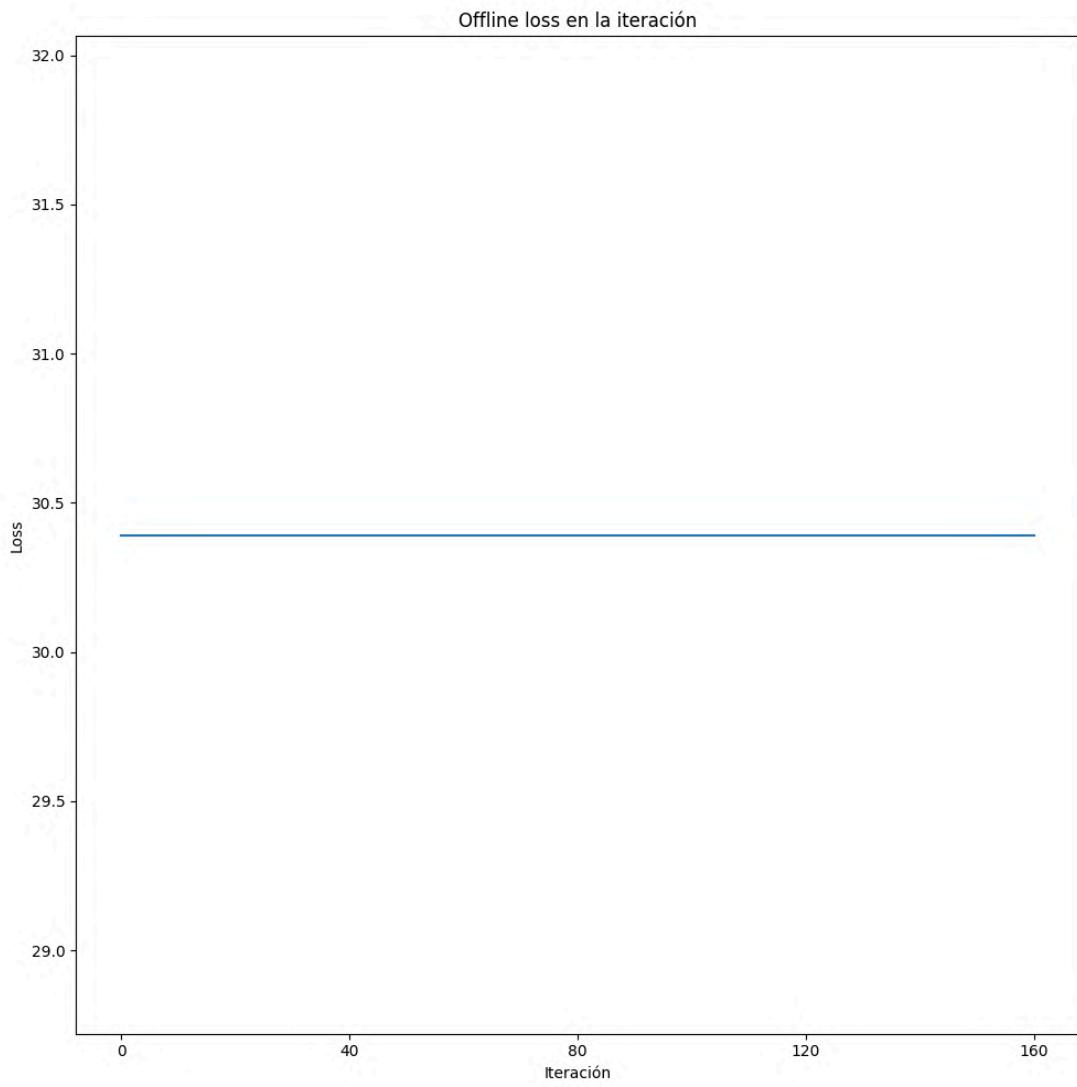
Configuración 5

La diferencia con el caso base está en el parámetro de regularización. Se utiliza $\lambda = 0.01$

Gif del entrenamiento

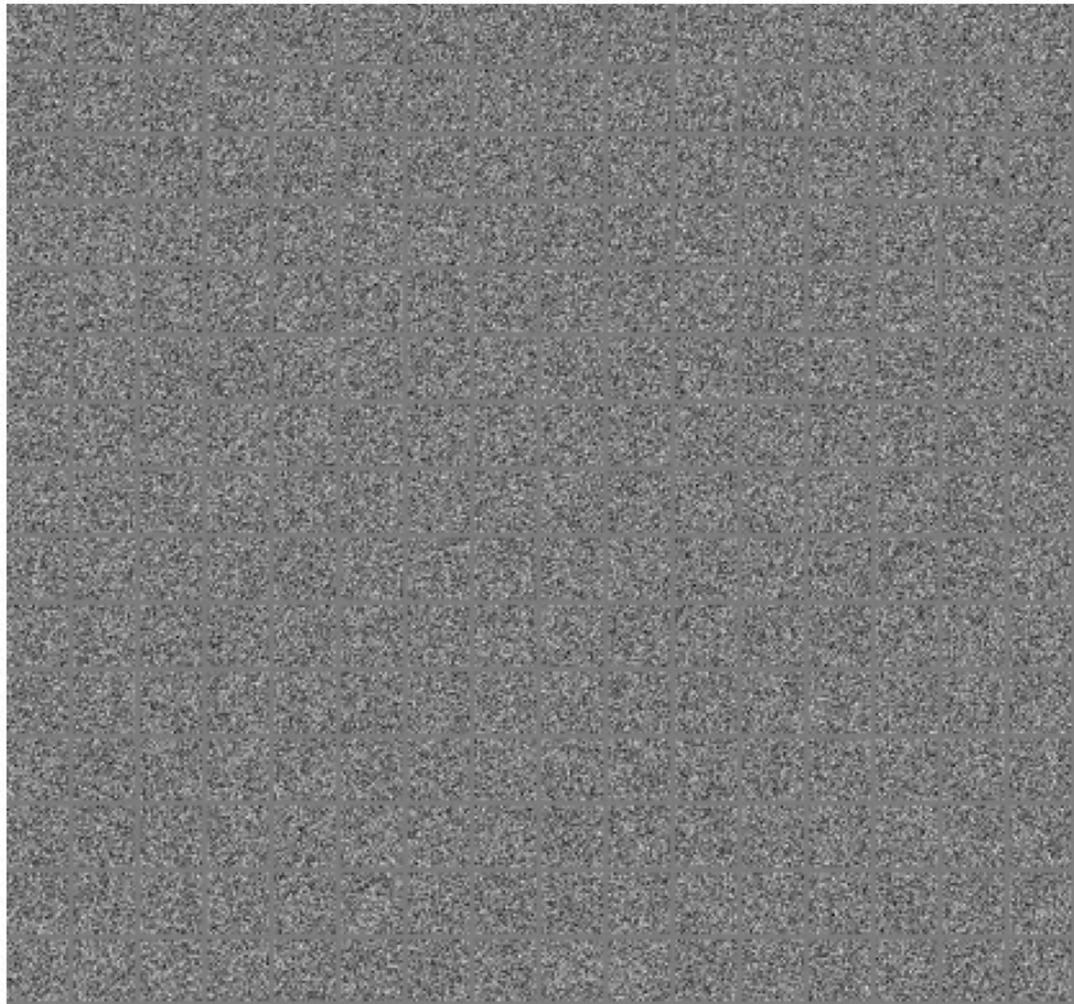


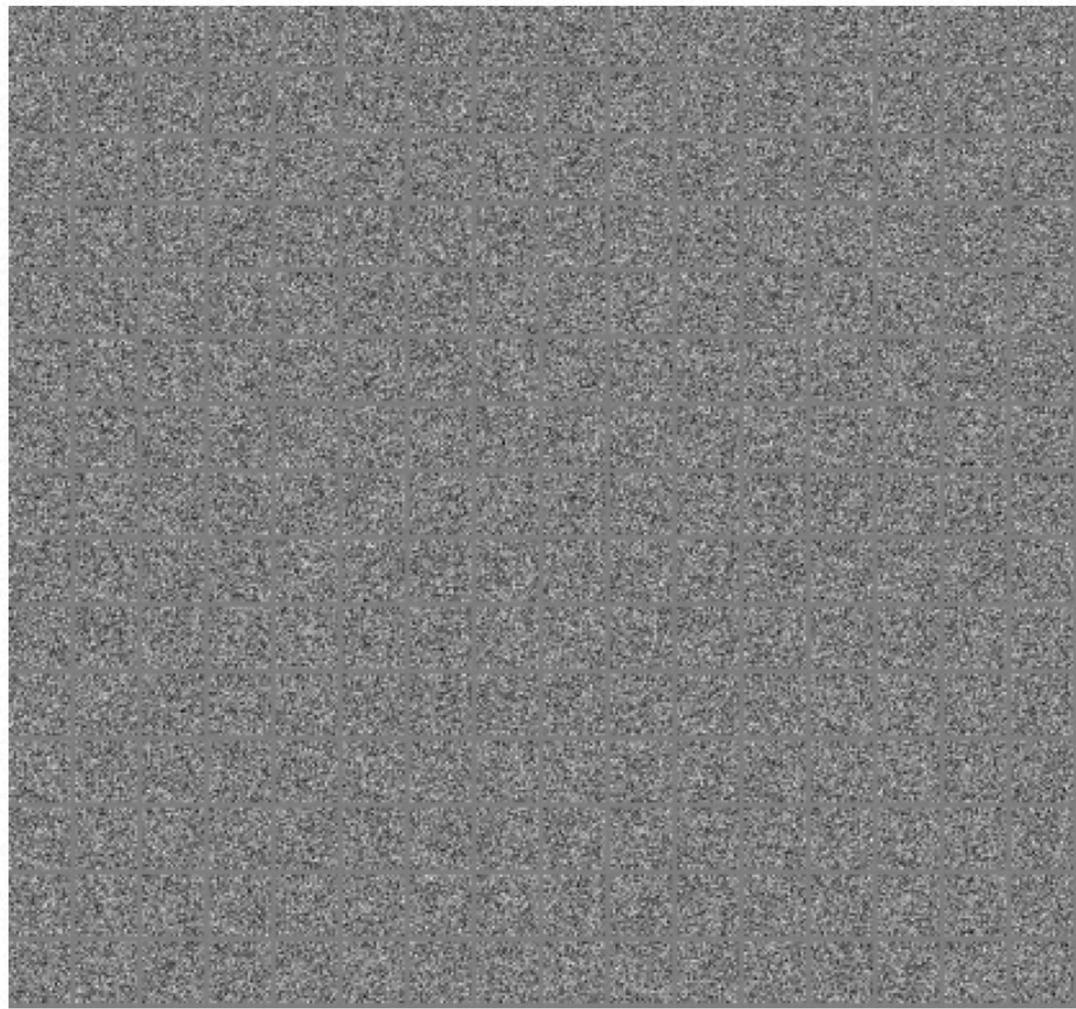
Graficas Loss



Primer iter, 18k iter, 40k iter

Atomos
Iteración 0



Atoms
Final

Tiempo de ejecución

Este entrenamiento se completó en 2m 38s. Es decir que consumió en promedio 4ms en procesar cada dato.

Cantidad de datos utilizados

En este entrenamiento se utilizaron 40.000 datos en batches de 200 elementos.

Algoritmo para entrenar configuración 5

```
In [ ]: conf = {  
    "a": 1,  
    "b": 0,  
    "d": 1,  
    "opt": "lars",  
    "lamd": 0.01,  
    "train_b_s": 200,  
    "iteraciones": 200,  
    "log": 10,  
    "dict_size": 250,  
}
```

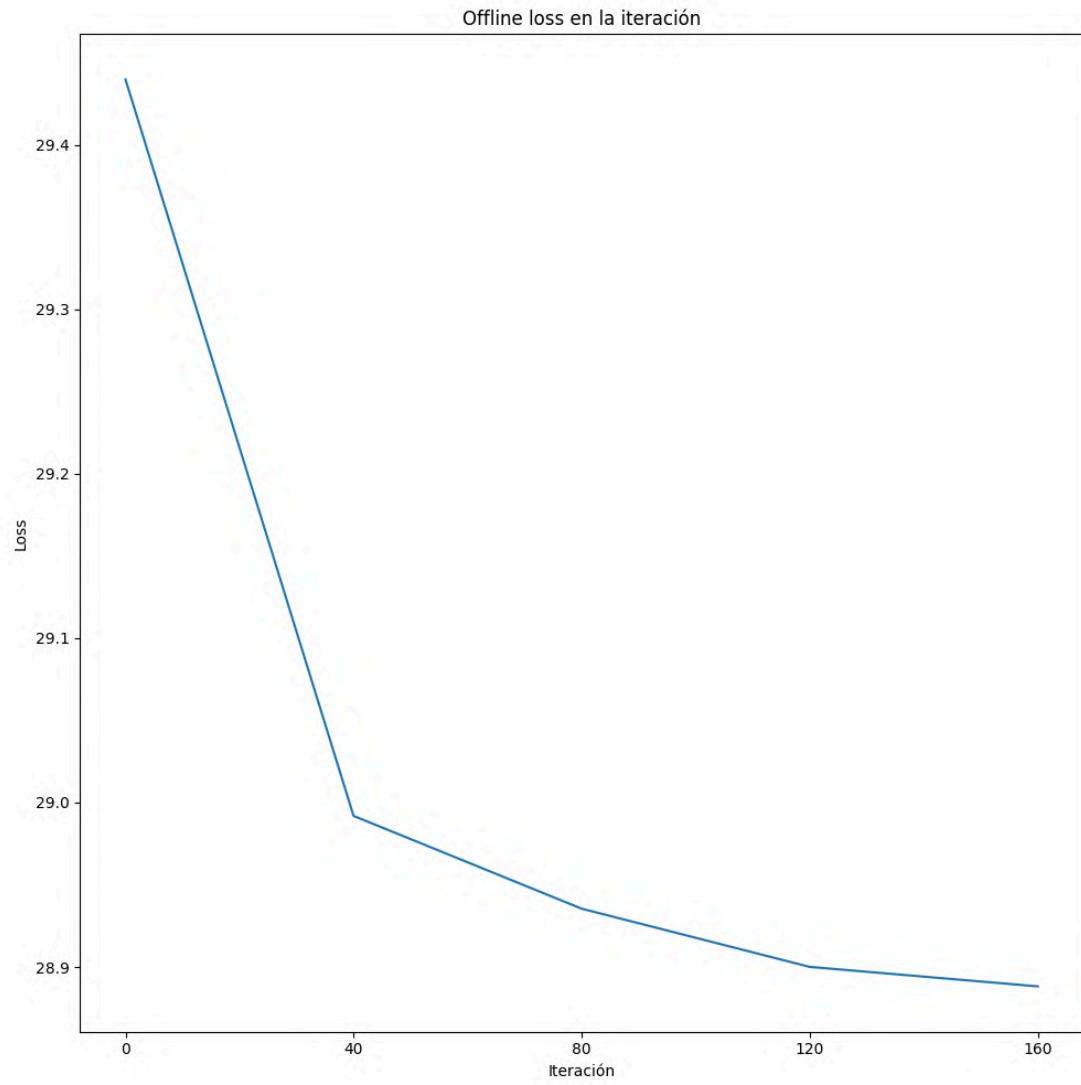
```
base_dir = f"dict-{conf['dict_size']}_{its}-{conf['iteraciones']}_{a}-{conf['ODL']}_luisa"
ODL = OnlineDictionaryLearning(
    test_batch_size=1000,
    base_dir=base_dir,
    log_step=conf["log"],
)
D = ODL.learn(
    it=conf["iteraciones"],
    lam=conf["lamd"],
    k=conf["dict_size"],
    optimizer=conf["opt"],
    init_A_mod=conf["a"],
    init_B_mod=conf["b"],
    init_D_mod=conf["d"],
    train_batch_size=conf["train_b_s"],
)
gif_name = f"proceso_diccionario_{its}-{conf['iteraciones']}_{lam}-{conf['lam']}_{path}"
paths.append(f"{base_dir}/{gif_name}")
```

Configuración 6

La diferencia con el caso base está en el parámetro de regularización y la inicialización del diccionario. Se utiliza $\lambda = 0.01$ e inicialización "0" del diccionario, es decir utilizando elementos del dataset como átomos.

Gif del entrenamiento

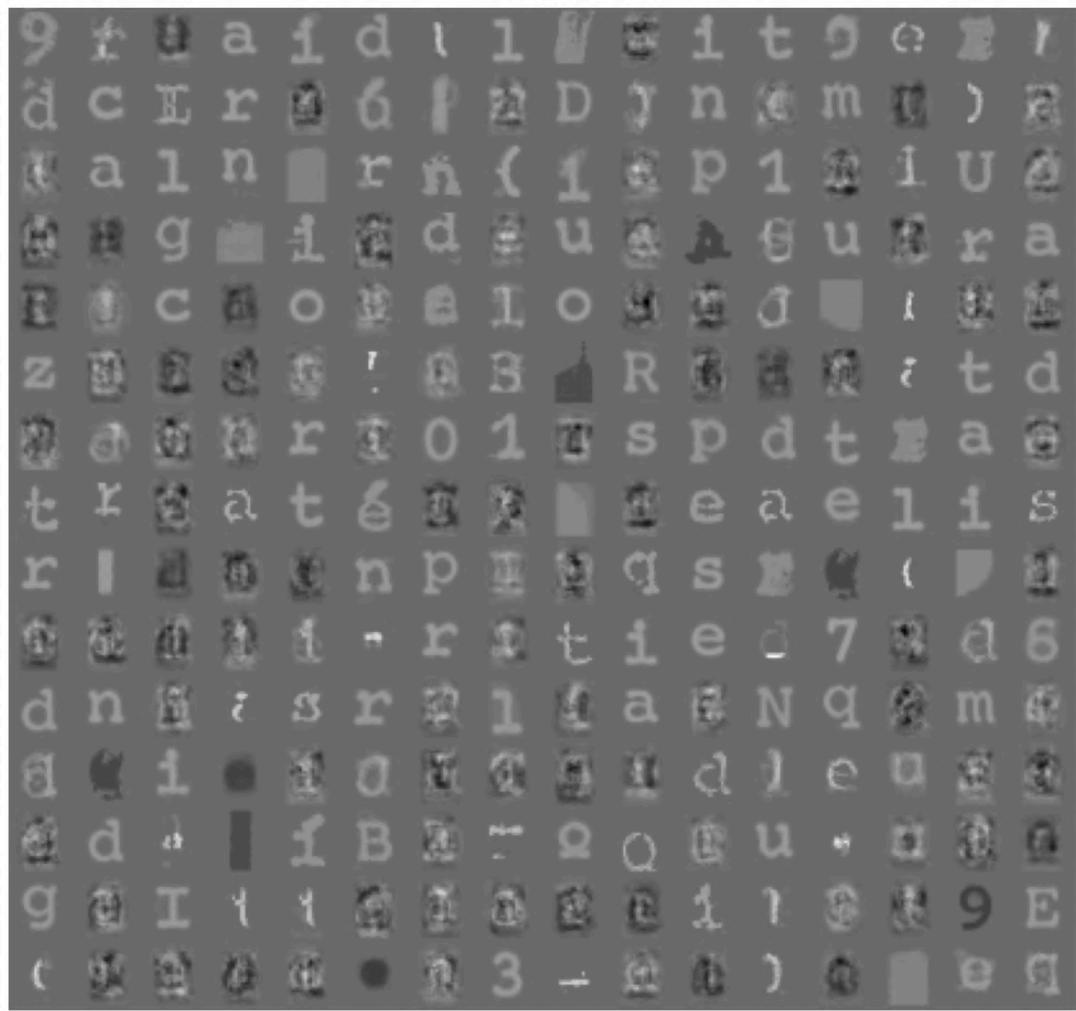
Graficas Loss



Primer iter, 18k iter, 40k iter

Atomas
Iteración 0

B	t	a	i	d	t	l	1	9	5	t	g	e	8	1
d	c	E	r	o	e	k	z	7	u	m)	8	0	0
u	a	l	n	u	r	u	(1	p	1	4	1	0	0
s	h	g	e	i	u	u)	u	u	u	9	u	1	0
o	o	c	o	o	u	a	l	0	o	o	0	o	0	0
s	o	o	o	o	o	o	o	0	o	o	o	o	o	0
s	o	o	o	o	o	o	o	0	o	o	o	o	o	0
t	o	o	o	o	o	o	o	0	o	o	o	o	o	0
r	:	o	o	o	o	o	o	0	o	o	o	o	o	0
@	o	o	o	o	o	o	o	0	o	o	o	o	o	0
d	n	o	o	o	o	o	o	0	o	o	o	o	o	0
a	o	o	o	o	o	o	o	0	o	o	o	o	o	0
o	d	o	o	o	o	o	o	0	o	o	o	o	o	0
g	o	o	o	o	o	o	o	0	o	o	o	o	o	0
c	o	o	o	o	o	o	o	0	o	o	o	o	o	0

Atomas
Final

Tiempo de ejecución

Este entrenamiento se completó en 3m 03s. Es decir que consumió en promedio 3.5ms en procesar cada dato.

Cantidad de datos utilizados

En este entrenamiento se utilizaron 40.000 datos en batches de 200 elementos.

Algoritmo para entrenar configuración 6

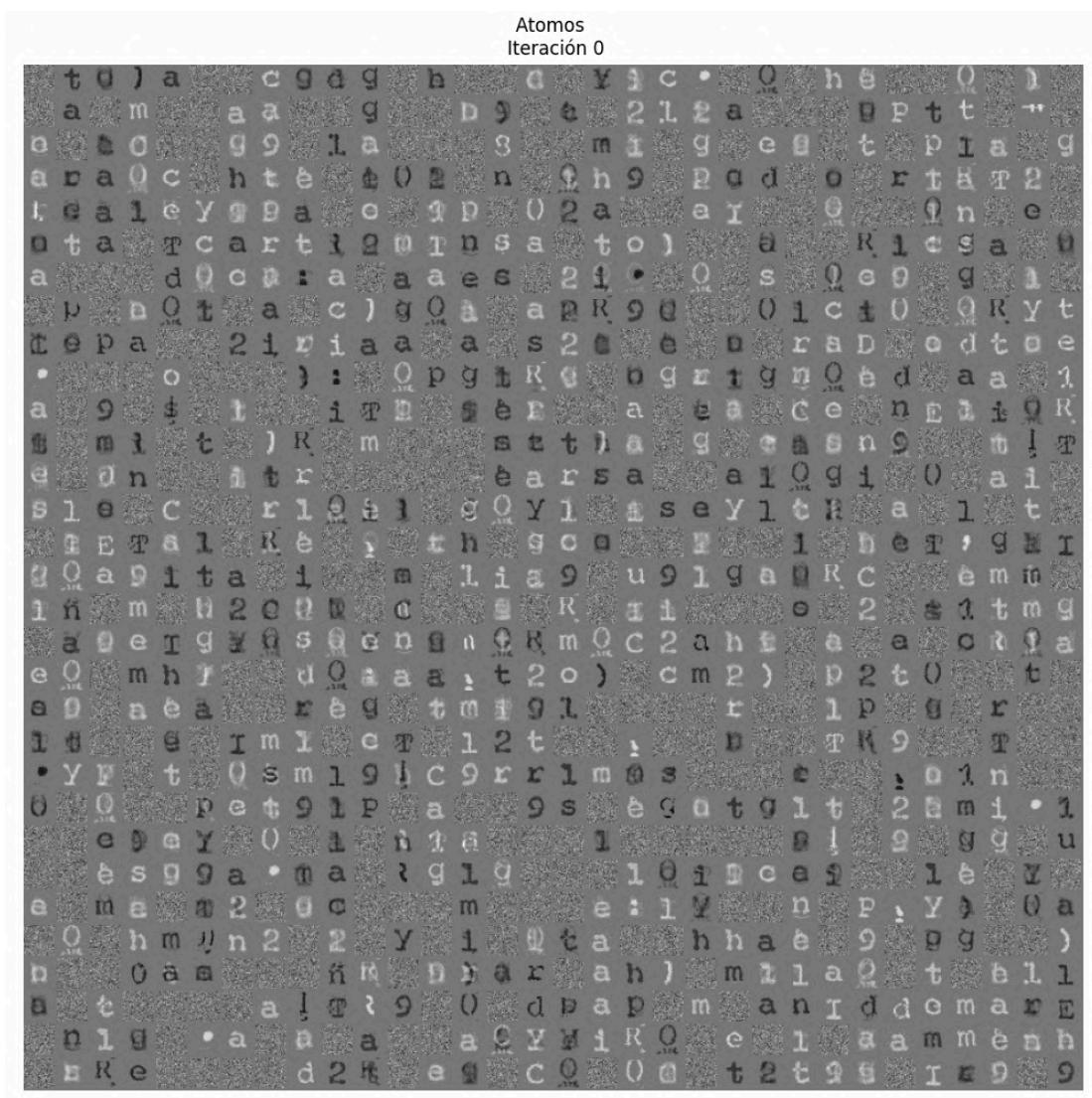
```
In [ ]: conf = (
    {
        "a": 1,
        "b": 0,
        "d": 0,
        "opt": "lars",
        "lamd": 0.01,
        "train_b_s": 200,
        "iteraciones": 200,
        "log": 10,
        "dict_size": 250,
    },
```

```
)  
  
base_dir = f"dict-{conf['dict_size']}_{its}-{conf['iteraciones']}_{a}-{conf['  
ODL = OnlineDictionaryLearning(  
    luisa,  
    test_batch_size=1000,  
    base_dir=base_dir,  
    log_step=conf["log"],  
)  
D = ODL.learn(  
    it=conf["iteraciones"],  
    lam=conf["lamd"],  
    k=conf["dict_size"],  
    optimizer=conf["opt"],  
    init_A_mod=conf["a"],  
    init_B_mod=conf["b"],  
    init_D_mod=conf["d"],  
    train_batch_size=conf["train_b_s"],  
)  
gif_name = f"proceso_diccionario_its-{conf['iteraciones']}_{lam}-{conf['lam']}  
paths.append(f"{base_dir}/{gif_name}")
```

Configuración 7

En esta configuración se agranda el diccionario a 1000 átomos. La diferencia con la configuración de base es que se emplea la inicialización "0" del diccionario.

Gif del entrenamiento



Graficas Loss



Primer iter, 18k iter, 40k iter

Atomos
Iteración 0

t d) a c g a g b d a y i c o h e o d	p t t " g
a m a a g g b g a 2 1 2 a p t p r a g	o p t t " g
e b o 9 9 l a s m i g e s t p r a g	t p r a g
a b a c h t e 0 2 n h 9 2 a d o r t a t 2	r t a t 2
t e a l e y 9 a c 1 p 0 2 a a r a Q Q n e	e
o t a T c a r t i 2 0 t n s s a t o) a s R i c s a u	R i c s a u
a d 0 c a r a a a e s 2 1 o s Q e g	Q e g
p b O t a c J g o a a 2 R 9 Q 0 1 c t 0 Q R Y t e	Q R Y t e
c e p a 2 i v i a a a s 2 0 e o r a D o d t o	o r a D o d t o
o o) : Q p g h R g o g r i g n Q e d a a 1	Q e d a a 1
a 9 \$ t i T E g e E a a e s C e n E l i Q R	C e n E l i Q R
m i t) R m s e t h a g e a s n g 9 l i T	g e a s n g 9 l i T
e o n h t r e a r s a a l o g i 0 a i t	a l o g i 0 a i t
s l e C r l Q a l g Q Y 1 a s e Y l t R a 1	t R a 1
E T a l R e i t h g c o T 1 b e t , g H I	T 1 b e t , g H I
Q a 9 i t a i m L i a 9 u 9 1 g a R C e m m	R C e m m
l n m 0 2 0 2 R c e R a i e 2 a 1 t m g	e 2 a 1 t m g
a 0 e T g W s e n g a Q R m Q C 2 a h p 2 t 0	p 2 t 0
e O m h f d Q a a a t 2 0) c m 2)	a a o a f a
s 0 a e a r e g t m 1 9 1	t l p g r
l 0 S e r m I c T 1 2 t	t D T K 9 T
Y F t Q s m 1 9 1 C 9 r r l m s e , o 1 n	, o 1 n
0 Q p e t 9 1 P a 9 s e g o t g l t 2 8 m i * 1	* 1
e 0 e Y 0 a h 1 0 1	g g
es 9 9 a m a g 1 9 1 0 i c e 1 l e z	z
a m a m 2 0 c m e : 1 V n P , Y 0 a	P , Y 0 a
Q h m n 2 2 Y 1 Q t a h h a e 9 g g))
o o a m n R D y a r a h J m l a Q t b l l	t b l l
a t a l 0 9 0 d p a p m a n I d d e m a r E	d d e m a r E
n l g a a a a a Q Y M i R Q e 1 a a m m e n b	a a m m e n b
n R e d 2 t a g C Q 0 0 t 2 t 9 9 I z 9 9	I z 9 9

Atoms
Final

Tiempo de ejecución

Este entrenamiento se completó en 17m 46s. Es decir que en promedio se consumió 27ms en procesar cada dato.

Cantidad de datos utilizados

En este entrenamiento se utilizaron 40.000 datos en batches de 200 elementos.

Algoritmo para entrenar configuración 7

```
In [ ]: conf = (
    {
        "a": 1,
        "b": 0,
        "d": 1,
        "opt": "lars",
        "lamd": 0.01,
        "train_b_s": 200,
        "iteraciones": 200,
        "log": 10,
        "dict_size": 1000,
    },
```

```
)  
  
base_dir = f"dict-{conf['dict_size']}_{its}-{conf['iteraciones']}_{a}-{conf['  
ODL = OnlineDictionaryLearning(  
    luisa,  
    test_batch_size=1000,  
    base_dir=base_dir,  
    log_step=conf["log"],  
)  
D = ODL.learn(  
    it=conf["iteraciones"],  
    lam=conf["lamd"],  
    k=conf["dict_size"],  
    optimizer=conf["opt"],  
    init_A_mod=conf["a"],  
    init_B_mod=conf["b"],  
    init_D_mod=conf["d"],  
    train_batch_size=conf["train_b_s"],  
)  
gif_name = f"proceso_diccionario_its-{conf['iteraciones']}_{lam}-{conf['lam']}  
paths.append(f"{base_dir}/{gif_name}")
```