Tarea 3

Autor: Arturo Lazcano **Profesor:** Felipe Tobar
Auxiliares: Catherine Benavides, Camila Bergasa, Víctor Caro, Camilo Carvajal Reyes, Diego Cortez M., Stefano Schiappacasse

Parte 1: *Clustering* y reducción de dimensionalidad

(a) En consideración a la aplicación de *PCA*, se procede a normalizar los datos como es usual y así también mantener unos vectores y ejes de futuros gráficos en escala más pequeña. Sin embargo, es necesario recalcar que esta normalización no es necesaria ya que en el formato en que se importa la base de datos y la naturaleza de las imagenes en blanco y negro, estas vienen con un valor mínimo (0) y máximo (255) por píxel.
Para elescalamiento se usa la función de sklearn *MinMaxScaler* y no *StandardScaler* pues esta última afecta la distribución de los datos y, por lo tanto, su media.

Luego, se puede observar como se ven las 10 componentes principales entregadas por el algoritmo *PCA*. Estas están en formato vector, para luego ser redimensionadas a nu formato compatible con imágenes.

```
PC1: [-0.02036971 -0.02217274 -0.02447324 ... -0.01553858 -0.01410884
 -0.01221962]
PC2: [ 0.0306322   0.03223464  0.03243738 ... -0.03766823 -0.03777706
 -0.03748186]
PC3: [-0.05965864 -0.06424096 -0.06901722 ...  0.01059249  0.00740709
  0.00526045]
PC4: [0.04392366 0.04044749 0.03312362 ... 0.06958651 0.07094279 0.06971673]
PC5: [-0.05048452 -0.05455727 -0.05877552 ... -0.03296829 -0.03033899
 -0.02822036]
PC6: [0.0138909  0.02315071 0.03307649 ... 0.07481735 0.07462111 0.0691986 ]
PC7: [-0.039962   -0.04301862 -0.04184375 ...  0.01006037  0.00246417
 -0.00383216]
PC8: [ 0.0639709   0.07100482  0.07418846 ... -0.02089841 -0.02639396
 -0.02852102]
PC9: [-0.05861164 -0.06024825 -0.06019125 ...  0.07046052  0.07076744
  0.0661601 ]
PC10: [-0.00467666 -0.00383393 -0.00352302 ... -0.02102548 -0.0226218
 -0.02379451]

Media de las componentes: [0.40612392 0.42166588 0.44215922 ... 0.3891851  0.38463843 0.38076706]
```

Figura 1: Componentes principales en formato vector.

En la siguiente figura se muestra como estas componentes principales son representadas como una foto en blanco y negro de 28×28.

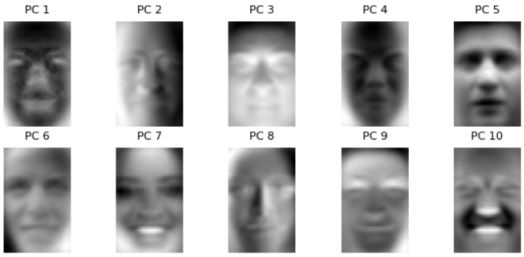

Figura 2: Componentes principales en formato imagen.

Figura 3: Media de componentes principales en formato imagen.

Se observa que cada componente principal se enfoca en ciertas partes de las caras o de la foto en general. Por ejemplo, se puede ver como la comp. principal 2 le da más importancia (color) al lado izquierdo de la foto mientras que la comp. principal 3 le da importancia todo menos la frente junto con el borde superior de la imagen (y la nariz ligeramente). Así, se puede ver como cada componente principal se enfoca en una parte distinta de la cara o foto.

Por otro lado, la media de los datos se ve en la última imagen, donde los píxeles más oscuros están relacionados a los ojos, boca, un poco la nariz y la parte externa de la foto que no pertenece a la cara.

(b) En la siguiente imagen se puede observar el resultado de calcular la raíz del error cuadrático medio (RMSE) sobre todo el conjunto de entrenamiento y el conjunto de test por separado. Esto se grafica con respecto a la cantidad de componentes principales que se usan para dicho cálculo.

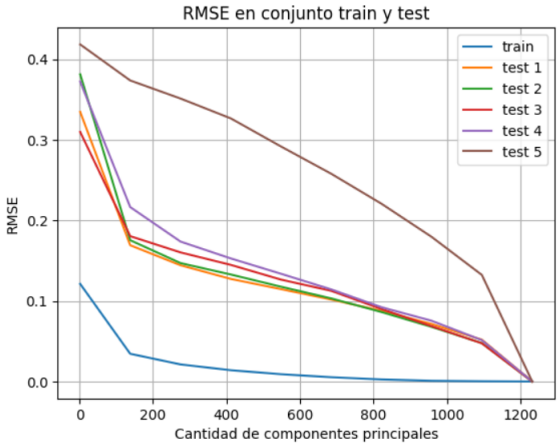

Figura 4: RMSE en conjunto de entrenamiento y prueba por separado.

Aquí se puede notar como el conjunto de entrenamiento tiende más rápidamente a 0 que las fotos de prueba. Esto tiene sentido pues el modelo entrena con ellas, mientras que las fotos de test son nuevas para el modelo, por lo cual se termina en una dependencia de la generalización del modelo en sí.
Por otro lado, se puede ver como, a pesar de la velocidad que toman, cada foto de prueba (y el conjunto train) termina en un error cuadrático medio 0 cuando se usan todas las componentes principales. Esto se debe a que se usan todas las características posibles (píxeles) para la reconstrucción mediante *PCA*.
Así, para cantidades bajas de componentes principales ($[0, 200]$), la reconstrucción presenta problemas y estos se ven reflejados en un RMSE de $[0.15, 0.45]$ en la nueva escala. Mientras que valores medianos ($[200, 800]$) presenta una caída más lenta del RMSE, estando en el intervalo $[0.1, 0.15]$.

En conclusión, *PCA* hace un buen trabajo si se piensa en reconstruir con una alta cantidad de componentes principales, ya que si no, este algoritmo obtiene resultados precarios como se pudo ver en la figura anterior.
Por último, en términos de tiempo de ejecución, es un algoritmo que resultó bastante eficiente en esta base de datos, demorando en el orden de segundos, aunque se lleguen a usar el máximo de componentes principales para la reconstrucción.

(c) En las siguientes figuras se muestra como el algoritmo *K-means* separa los datos en 20 clusters. Para ello se seleccionó solo la parte de la boca en cada imagen.

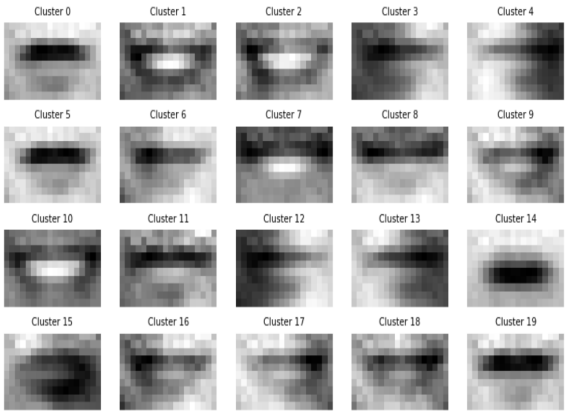

Figura 5: Clusters formados por K-means en píxeles de la boca.

A continuación se observa el mismo resultado anterior añadiendo las caras correspondientes a cada cluster entregado por *K-means*.

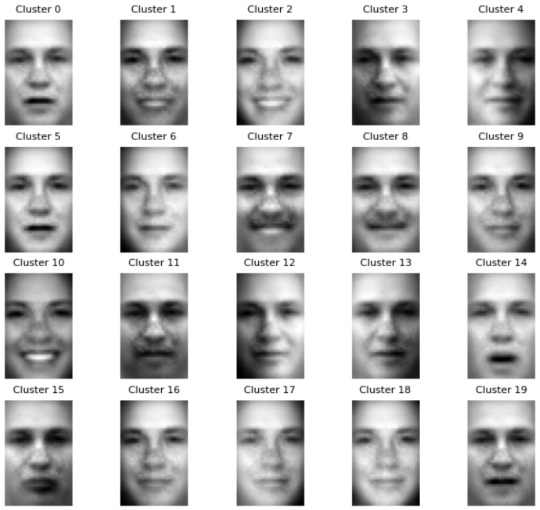

Figura 6: Clusters formados por K-means mostrando cara completa.

Así, en las figuras anteriores se observan 20 clusters, donde en esta sección se trabajará con el cluster 2. Esta elección es arbitraria, sin embargo, se elige pues es uno de los clusters que captura imágenes con una clara sonrisa donde se ven los dientes.

En el siguiente gráfico se puede ver la proyección de todos lo datos (usando *PCA*), donde se distingue solo el cluster elegido. Acá se puede notar que no parece un conjunto separable del resto de datos, sin embargo, esto puede deberse a que la proyección es en $\mathbb{R}^2$ pero la data original es de una dimensión mucho mayor, donde en este espacio podrían estar mayormente separados.

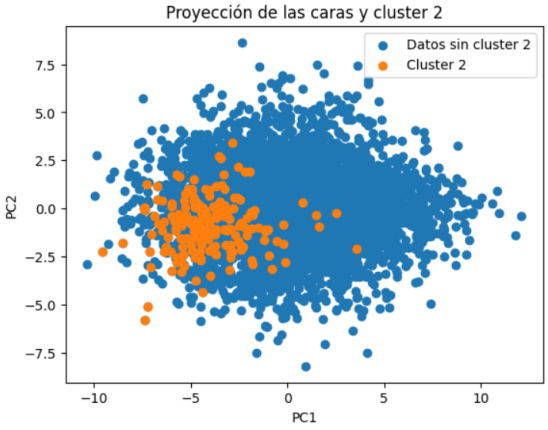

Figura 7: Cluster elegido en comparación a toda la data.

Luego, se procede a cambiar una de las componentes principales (en este caso la segunda) y reemplazarla por la media de los datos − la media del cluster 2.

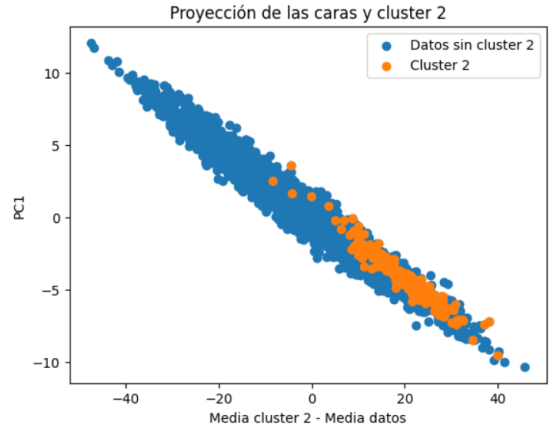

Figura 8: Proyección con una componente principal y resta de medias.

Notar que este gráfico depende ampliamente por el cluster elegido, ya que la proyección cambia radicalmente si elegimos otro cluster. Esto pues el eje x presenta la media del cluster.
Vemos que acá tampoco se logra una separabilidad clara, esto puede deberse al algoritmo de proyección que se está usando. Otros posibles algoritmos de reducción de dimensionalidad son Isomap, T-SNE, autoencoders, entre otros. Por lo cual probar alguna de estos otras o incluso kernel PCA puede ayudar a lograr una mayor separación en esta proyección.

(d) En esta sección y lo que queda de este trabajo ya se cambia la base de datos a trabajar por *fashion MNIST*.
En la siguiente figura se puede ver como se realiza clustering mediante *K-means* con 10 clusters. Este número se debe a que son 10 clases las que vienen en el dataset.
Luego de realizar clustering, se proyectan usando 2 componentes principales mediante *PCA*.

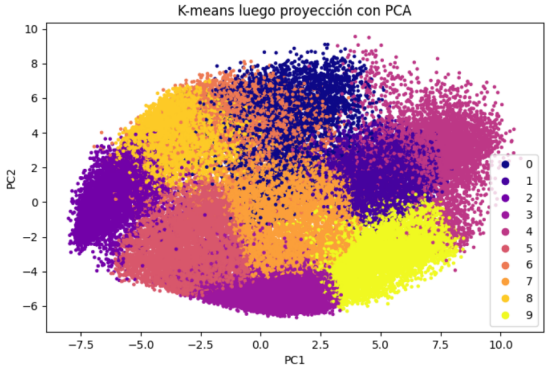

Figura 9: Clusters de K-means proyectados con PCA.

Por otro lado, se prueba el método anterior de forma inversa, es decir, se realiza *PCA* para proyectar los datos y luego realizar clustering.
Se puede ver que esto carece más de sentido pues se pierde mucha información al proyectar en $\mathbb{R}^2$ y donde *K-means* separa los clusters en 2 dimensiones, formando un clásico diagrama de Voronoi.

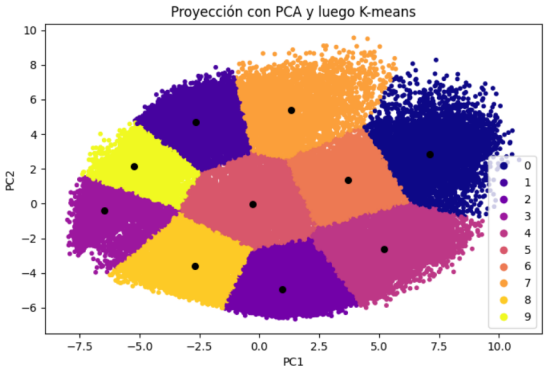

Figura 10: PCA para proyectar y luego K-means.

Por último, para poder ver que algoritmo pudiese tener mejores resultados en la separación de los datos, se grafican los datos reales usando *PCA*.

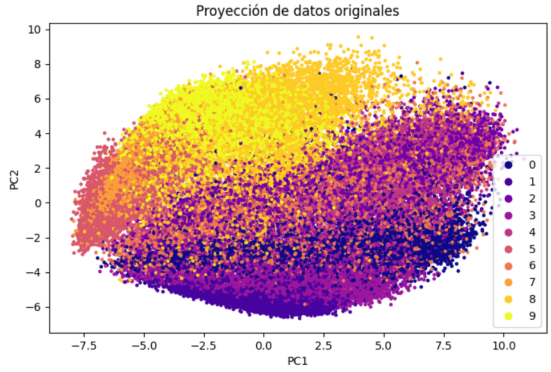

Figura 11: Clases reales de los datos.

En conclusión, comparando las primeras dos figuras con las clases reales de los datos, parece ser que realizar clustering y luego proyectar es la opción que mejor se acerca a la realidad del dataset. Sin embargo, la proyección en 2D no parece muy clara. Es por esto que se propone realizar la proyección en $\mathbb{R}^3$ o, al igual que antes, proponer algoritmos distintos de reducción de dimensionalidad como puede ser kernel PCA, t-SNE, Isomap, entre otros. Incluso se puede probar lo realizado en la parte (c), donde se reemplaza una componente principal por a media de un cluster − la media de los datos. Esto podría capturar una separación de un cluster con respecto al resto.

Parte 2: Redes neuronales

(a) Sea $\hat{p}_i^{(j)}$ la probabilidad estimada de la muestra i-ésima para la clase j, donde $\hat{p}_i^{(j)} \in [0,1]$. Por otro lado, sea $p_i^{(j)}$ la probabilidad real (empírica) de la i-ésima muestra para la clase j, donde $p_i^{(j)} \in \{0,1\}$ (podemos pensar los labels en formato one hot encoder).
Así, $y_i^{(j)} = p_i^{(j)} = \mathbb{1}_{\{y_i^{(j)}=1\}}$ son los labels "duros", es decir, 1 si la muestra i corresponde a la clase k y 0 si no.
Recordemos que el estimador de máxima verosimilitud se define como

$$\hat{\theta} = \underset{\theta}{\text{argmax}}\ P(\theta|X) \overset{\text{Bayes}}{=} \underset{\theta}{\text{argmax}} \frac{P(X|\theta)P(\theta)}{P(X)} = \underset{\theta}{\text{argmax}}\ P(X|\theta)P(\theta) \tag{1}$$

Acá se usó que $P(X)$ no afecta al problema de optimización. Así, como es usual en redes neuronales, se asume $P(\theta)$ como distribución uniforme. Con esto, el estimador de máxima verosimilitud queda

$$\hat{\theta} \overset{\text{iid}}{=} P(X|\theta) = \underset{\theta}{\text{argmax}} \prod_{i=1}^{N} P((x_i, y_i)|\theta) = \underset{\theta}{\text{argmax}} \prod_{i=1}^{N}\prod_{j=1}^{K} \left(\hat{p}_i^{(j)}\right)^{\mathbb{1}_{\{y_i^{(j)}=1\}}} = \underset{\theta}{\text{argmax}} \prod_{i=1}^{N}\prod_{j=1}^{K} \left(\hat{p}_i^{(j)}\right)^{p_i^{(j)}} \tag{2}$$

Esto pues, la pitatoria que va hasta K es la versión multiclase de lo que ya hemos visto en el curso en caso binario.
Luego, por ser $-\log(\cdot)$ una función monótona decreciente, podemos aplicarla a la ecuación anterior cambiando el problema maximizar a minimizar la expresión. Entonces,

$$\hat{\theta} = \underset{\theta}{\text{argmin}} - \log P(X,\theta) = \underset{\theta}{\text{argmin}} - \sum_{i=1}^{N}\sum_{j=1}^{K} p_i^{(j)} \log\left(\hat{p}_i^{(j)}\right) = \underset{\theta}{\text{argmin}} \sum_{i=1}^{N}\sum_{j=1}^{K} p_i^{(j)} \cdot \left(-\log\left(\hat{p}_i^{(j)}\right)\right) \tag{3}$$

$$= \underset{\theta}{\text{argmin}} \sum_{i=1}^{N}\sum_{j=1}^{K} p_i^{(j)} \log\left(\frac{1}{\hat{p}_i^{(j)}}\right) \tag{4}$$

Luego, se puede incorporar el factor $1/N$ pues este no afecta al problema de optimización, resultando:

$$\hat{\theta} = \underset{\theta}{\text{argmin}} \left[\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{K} p_i^{(j)} \log\left(\frac{1}{\hat{p}_i^{(j)}}\right)\right] = \underset{\theta}{\text{argmin}} \left[\frac{1}{N}\sum_{i=1}^{N} H(p_i, \hat{p}_i)\right] = \underset{\theta}{\text{argmin}}\ L(p_\theta, p) \tag{5}$$

Donde p no es más que la distribución de probabilidad empírica y p_θ las estimaciones. Notar que en toda este argumento, se ha omitido que las estimaciones dependen de los puntos (x_i) y los parámetros (θ), por lo cual tiene sentido el problema de optimización en los parámetros.

(b) Considerando un problema de clasificación binaria con una función de activación tipo sigmoide

(σ). Se puede definir la probabilidad de pertenecer a la clase 1 como

$$p_\theta(y=1|x) = \sigma\left(\sum_{i=1}^{N} x_i w_j + b\right) \tag{6}$$

Donde $\theta = (w_1, ..., w_N, b)$ son los parámetros representando los pesos y el sesgo del modelo.

Así, para poder entrenar este modelo, se debe definir una función de pérdida adecuada. En este caso, se puede hacer uso de la función *logg loss*.

$$L_{\log}(y,p) = -[(y\log(p) + (1-y)\log(1-p))] \tag{7}$$

Donde $y \in \{0,1\}$ es el label real y $p = P(y=1)$ es la estimación. Luego, para usar descenso de gradiente estocástico (SGD), se necesita calcular las derivadas parciales de esta función de pérdida con respecto al parámetro θ, es decir, w y b.
Usando la notación $\theta x^i := b + w_1 x_1 + ... + w_N x_N$. Notemos que,

$$\log\sigma(\theta x^i) = -\log(1+e^{-\theta x^i}) \tag{8}$$

$$\log(1-\sigma(\theta x^i)) = \log\left(\frac{1+e^{-\theta x^i}}{1+e^{-\theta x^i}} - \frac{1}{1+e^{-\theta x^i}}\right) = \log(e^{-\theta x^i}) - \log(1+e^{-\theta x^i}) = -\theta x^i - \log(1+e^{-\theta x^i}) \tag{9}$$

Así, si consideramos la función de costo anterior por iteración realizada, podemos definir la función objetivo como

$$J(\theta) = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(\sigma(\theta x^i)) + (1-y)\log(1-\sigma(\theta x^i)) \tag{10}$$

Por lo cual, usando las simplificaciones anteriores, $J(\theta)$ se reduce a

$$J(\theta) = -\frac{1}{N}\sum_{i=1}^{N}[y_i\theta x^i - \theta x^i - \log(1+e^{-\theta x^i})] = -\frac{1}{N}\sum_{i=1}^{N}[y_i\theta x^i - \log(1+e^{\theta x^i})] \tag{11}$$

Por último, se calculan las derivadas parciales como:

$$\frac{\partial}{\partial\theta_j} y_i\theta x^i = y_i x_j^i, \quad \frac{\partial}{\partial\theta_j}\log(1+e^{\theta x^i}) = \frac{x_j^i e^{\theta x^i}}{1+e^{\theta x^i}} = x_j^i\sigma(\theta x^i) \tag{12}$$

$$\implies \frac{\partial}{\partial\theta_j} J(\theta) = -\frac{1}{N}\sum_{i=1}^{N}(\sigma(\theta x^i) - y^i)x_j^i \tag{13}$$

Así, el algoritmo SGD calcula estas derivadas para actualizar sus parámetros w y b de la forma $\theta = \theta - \eta\nabla L_{\log}(\theta)$. En caso de agregar una capa, esto pasa a ser una red neuronal de 2 capas, por lo que la salida de la capa oculta viene dada por la función se activación σ, se comparan los valores del output (función de pérdida) con los reales y se ajustan los pesos y sesgo mediante SGD en cada capa.
El algoritmo *backpropagation* es crucial en estos escenarios pues permite calcular de forma eficiente los gradientes necesarios. El algoritmo funciona en dos pasos principales: la propagación forward y la propagación backward. Para la etapa forward, los datos de entrada se alimentan a través de la red neuronal, capa por capa, hasta llegar a la capa de salida. Cada neurona realiza una suma ponderada de las entradas y aplica una función de activación (sigmoide en este caso), produciendo una salida que se convierte en la entrada para la siguiente capa. Este proceso se repite hasta llegar a la capa de salida, donde se genera la salida final de la red. Luego, se calcula la función de pérdida utilizando la salida predicha por la red y los valores reales.
En cambio, en la etapa backward, el objetivo es calcular las derivadas parciales de la función de pérdida con respecto a los pesos de la red, comenzando desde la capa de salida y retrocediendo hacia la capa de entrada. Esto se logra debido a la regla de la cadena. Estas derivadas parciales se utilizan para ajustar los pesos mediante SGD, que actualiza los pesos en dirección opuesta al gradiente de la función de pérdida. Estos pasos se repiten iterativamente y se ogra un entrenamiento óptimo de la red neuronal.

(c) En el archivo adjunto .ipynb se logra implementar exitosamente la función descenso de gradiente mini-batch. Luego, con el fin de probar esta función se hace uso de la base de datos *fashion MNIST*.
En la siguiente imagen se puede ver la evolución de las pérdidas que obtiene el modelo (regresión logística) en cada iteración que va realizando.

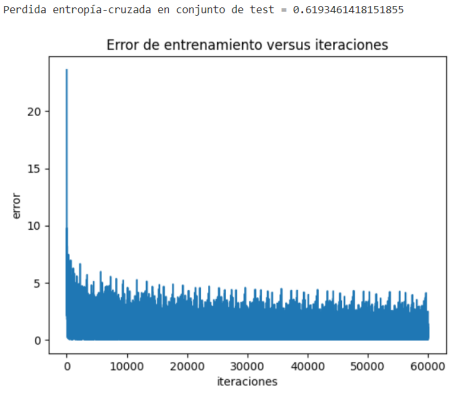

Figura 12: Pérdida regresión logística con descenso gradiente mini-batch.

Se puede observar que, luego de unas pocas iteraciones, el error oscila en el rango $[4, 5]$. Después de esto, se procede a realizar la tarea de clasificación en un conjunto test obteniendo los resultados siguientes:

	precision	recall	f1-score	support
0	0.95	0.45	0.61	1000
1	0.97	0.96	0.96	1000
2	0.74	0.60	0.66	1000
3	0.49	0.97	0.65	1000
4	0.76	0.62	0.68	1000
5	0.79	0.52	0.63	1000
6	0.44	0.66	0.53	1000
7	0.42	1.00	0.59	1000
8	0.99	0.35	0.52	1000
9	0.99	0.08	0.15	1000
accuracy			0.62	10000
macro avg	0.75	0.62	0.60	10000
weighted avg	0.75	0.62	0.60	10000

Figura 13: *Classification report* del primer modelo.

Acá se puede ver que las clases mejores predichas fueron la $1, 8$ y 9 mientras que las peores fueron las $3, 6$ y 7 hablando de *precision*. Alcanzando un total de *Accuracy* de 0.62.

(d) En esta parte se modifica el código de la parte anterior, añadiendo un modelo no entrenado, un optimizador y un número de épocas para que el algoritmo sea más flexible.

En la siguiente figura se grafican las pérdidas obtenidas por este modelo, las cuales son menores que el anterior si se presta atención al eje y, oscilando en el intervalo $[2, 4]$ después de unas pocas iteraciones.

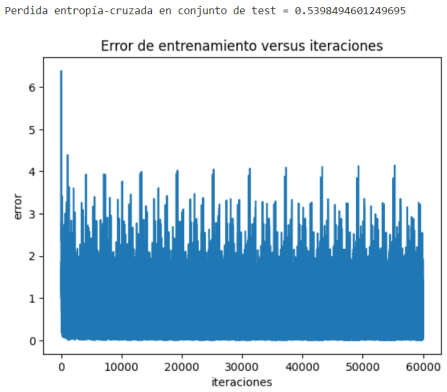

Figura 14: Pérdida regresión logística con optimizador SGD y descenso gradiente mini-batch.

Al igual que antes, se realiza la tarea de clasificar el mismo conjunto test. En este caso, se usa el mismo número de épocas para evaluar la eficiencia de un optimizador (pues también se usa una regresión logística en esta parte).

	precision	recall	f1-score	support
0	0.99	0.41	0.58	1000
1	1.00	0.87	0.93	1000
2	0.81	0.60	0.69	1000
3	0.57	0.96	0.71	1000
4	0.78	0.69	0.74	1000
5	0.96	0.39	0.56	1000
6	0.49	0.75	0.59	1000
7	0.44	1.00	0.61	1000
8	0.97	0.84	0.90	1000
9	0.99	0.31	0.47	1000
accuracy			0.68	10000
macro avg	0.80	0.68	0.68	10000
weighted avg	0.80	0.68	0.68	10000

Figura 15: *Classification report* del segundo modelo.

En esta figura se puede ver como los resultados son mejores que en la parte anterior, lo cual indica que añadir un optimizador al algoritmo es una buena idea. Sin embargo, sigue teniendo problemas con algunas clases particulares.

A modo de comparación, se ejecuta nuevamente este algoritmo cambiando el optimizador elegido de la librería *torch*. En este caso, se opta por el optimizador *Adagrad*, el cual se diferencia de *SGD* en que este último no toma en consideración los gradientes calculados de las iteraciones pasadas. Por otro lado, la tasa de aprendizaje en *Adagrad* es calculada en cada iteración (usando la tasa entregada como hiperparámetro).
Con esto, en la siguiente imagen se pueden ver las pérdidas que se obtuvieron usando este opti-

mizador junto con el rendimiento que obtuvo en la tarea de clasificación.

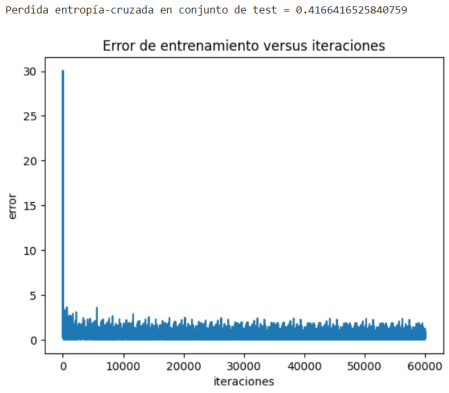

Figura 16: Pérdida regresión logística con optimizador Adagrad descenso gradiente mini-batch.

	precision	recall	f1-score	support
0	0.84	0.77	0.80	1000
1	0.99	0.94	0.97	1000
2	0.81	0.71	0.76	1000
3	0.74	0.94	0.83	1000
4	0.82	0.69	0.75	1000
5	0.81	0.69	0.74	1000
6	0.62	0.61	0.62	1000
7	0.50	1.00	0.66	1000
8	0.82	0.98	0.89	1000
9	0.99	0.13	0.23	1000
accuracy			0.74	10000
macro avg	0.80	0.74	0.72	10000
weighted avg	0.80	0.74	0.72	10000

Figura 17: *Classification report* del tercer modelo.

Así, se puede concluir que, en términos de pérdidas, parece similar a usar *SGD*, sin embargo, el *Accuracy* incrementa en un factor de ~ 0.6.
Se debe aclarar que ambos optimizadores están sujetos a la elección de los hiperparámetros correspondientes. Por otro lado, dependiendo de la tarea, el tiempo de ejecución es también una variable a considerar por lo que esta conclusión no es una regla práctica a seguir en cualquier dataset.

(e) Por último, en esta sección se hará uso de redes neuronales convolucionales (CNN). Estas son un tipo de redes neuronales muy usadas tanto en el área de NLP y procesamiento de imágenes. Esto pues, en su arquitectura, se realizan operaciones de convolución 1D o 2D para el caso de trabajar en texto o imágenes respectivamente.

Algunas ventajas de usar cnn es que poseen invariancia espacial, es decir, son capaces de aprender características que son invariantes a rotaciones, traslaciones, entre otros. Por otro lado, son capaces de aprender sin requerir un preprocesamiento de píxeles (en caso de las imágenes) y su estructura jerárquica, donde las primeras capas aprender patrones simples mientras que las ultimas algunos más complejos, ayuda a capturar los elementos importantes del input.

En la siguiente imagen se puede ver el rendimiento de una cnn implementada en el archivo ipynb. Esta posee 10 épocas y 3 capas de convolución 2D. Por otro lado, usa la función de activación *ReLu* junto con el optimizador *Adam*.

	precision	recall	f1-score	support
0	0.80	0.88	0.83	1000
1	0.99	0.98	0.99	1000
2	0.82	0.92	0.87	1000
3	0.93	0.91	0.92	1000
4	0.89	0.81	0.85	1000
5	0.99	0.97	0.98	1000
6	0.82	0.73	0.77	1000
7	0.96	0.96	0.96	1000
8	0.98	0.99	0.99	1000
9	0.96	0.97	0.97	1000
accuracy			0.91	10000
macro avg	0.91	0.91	0.91	10000
weighted avg	0.91	0.91	0.91	10000

Figura 18: *Classification report* de la CNN.

Acá se observa que el rendimiento en términos de *Accuracy* es notoriamente mayor a los algoritmos implementados en las partes anteriores. Sin embargo, este toma un tiempo de ~ 26 minutos, lo cual difiere en gran medida con los modelos anteriores que ejecutaban en el orden de segundos.

En la siguiente tabla se puede la pérdida de entropía cruzada que obtuvo cada algoritmo.
En conclusión, se puede ver que el uso de una cnn en procesamiento de imágenes es claramente superior a los algoritmos clásicos de regresión lineal usando descenso de gradiente mini-batch con un optimizador. Sin embargo, el tiempo que se tenga dispuesto a entrenar esta red puede ser crucial a la hora de elegir alguno de estos algoritmos. Por último, mencionar que para las redes neuronales, es posible usar el poder computacional que posee una GPU, paralelizando los cálculos y haciendo que su entrenamiento sea notoriamente más rápido.

	Pérdida en conjunto test
Regresión logística sin optimizador	0.619
Regresión logística con optimizador SGD	0.54
Regresión logística con optimizador Adagrad	0.417
CNN	0.32