



Práctica 4

Objetivo general: El objetivo de esta práctica es aplicar los conceptos fundamentales estudiados en el tercer bloque de la asignatura Fundamentos de Teoría de la Información, con particular énfasis en los modelos de compresión basados en tensores, redes neuronales y más concretamente en autoencoders. Para facilitar su comprensión, la práctica está dividida en dos partes, cada una centrada en un tema específico. Es importante comprender en profundidad cada apartado, tanto en la implementación del código como en los resultados obtenidos. Cada sección incluye preguntas para garantizar que los conceptos se han interiorizado correctamente. Los resultados deberán ser analizados y justificados con rigor técnico.

Práctica 4.1a: Matriciación y vectorización de tensores. Tensores de rango 1.

Objetivo: El propósito de esta práctica es desarrollar habilidades prácticas en la matriciación de tensores en los diferentes modos y en la generación de tensores de rango 1 a partir del producto exterior de vectores.

Material: Se proporciona el archivo **Práctica4.1a-alumnos.ipynb**, un notebook de Jupyter que contiene una estructura básica con fragmentos de código para guiarte a lo largo de los apartados que debes completar.

1. **Generación de un tensor de 3 dimensiones**
 - a. Genera un tensor de tamaño $I \times J \times K$, donde cada capa de tamaño $J \times K$ tenga un valor entero constante.
 - b. Imprime el tensor generado y verifica los valores de sus entradas.
2. **Matriciación del tensor**
 - a. Realiza la matriciación del tensor en sus tres modos.
 - b. Indica las dimensiones de las matrices resultantes en cada caso.
 - c. Representa los valores de las matrices usando gráficos (por ejemplo, con `imshow`) y verifica que los resultados sean coherentes con la teoría.
3. **Vectorización del tensor**
 - a. Vectoriza el tensor convirtiéndolo en un vector unidimensional.
 - b. Indica las dimensiones del vector resultante.
 - c. Representa los valores del vector y verifica que el resultado sea consistente con lo estudiado en las clases teóricas.
4. **Extensión a tensores de mayor dimensión**
 - a. Repite los pasos 1, 2 y 3 para un tensor de 4 dimensiones ($I \times J \times K \times L$).
 - b. Asegúrate de verificar y representar los resultados obtenidos en cada paso.
5. **Generación de tensores de rango 1**

- a. Genera un tensor de rango 1 y dimensión 3 ($I \times J \times K$) a partir del producto exterior de tres vectores de valores enteros.
- b. Verifica los valores de las entradas del tensor generado.
- c. Comprueba que los resultados sean consistentes con las propiedades teóricas de los tensores de rango 1.

Práctica 4.1b: Implementación del modelo PARAFAC con CVXPY

Objetivo: El propósito de esta parte de la práctica es implementar el modelo de PARAFAC para la descomposición de tensores visto en las clases de teoría de la asignatura. La idea principal es entender cómo se realiza la descomposición del tensor mediante la minimización de un problema de optimización por bloques.

Material: Se proporciona el notebook **Práctica4.1b-alumnos.ipynb**, el cual contiene funciones y fragmentos de código que servirán como guía para completar los distintos apartados.

1. Generación del tensor

- a. Genera un tensor de dimensiones específicas ($I \times J \times K$).
- b. Asegúrate de verificar que los valores del tensor sean consistentes con las características requeridas.

2. Matriciación del tensor

- a. Obtén las tres matricizaciones del tensor $X(1)$, $X(2)$, $X(3)$.
- b. Indica las dimensiones de las matrices obtenidas y verifica que los resultados coincidan con lo discutido en teoría.

3. Definición de variables a optimizar

- a. Define las variables **A**, **B** y **C** en CVXPY, correspondientes a las matrices factor en la descomposición PARAFAC.
- b. Estas matrices deben tener dimensiones $I \times F$, $J \times F$, y $K \times F$, respectivamente, donde F es el rango deseado.

4. Función objetivo

- a. Define las funciones objetivo que se minimizarán para cada paso iterativo, utilizando CVXPY.
- b. Asegúrate de que la función objetivo corresponda a la minimización de la norma de Frobenius al cuadrado entre el tensor matricizado y la aproximación generada por los productos Khatri-Rao.

5. Descomposición del tensor

- a. Resuelve el problema de optimización de manera iterativa, alternando entre la actualización de las variables **A**, **B** y **C**.
- b. Para cada iteración, evalúa los resultados y el error de reconstrucción.
- c. Al finalizar, comprueba que el tensor reconstruido a partir de las matrices **A**, **B** y **C** se asemeja al tensor original.

6. Descomposición del tensor mediante Alternating Least Squares (ALS)

- a. Realiza los pasos 1-5 resolviendo cada uno de los subproblemas mediante Least Squares.

7. Descomposición del tensor mediante tensorly

- a. Realiza los pasos 1-5 resolviendo el problema de optimización mediante la librería de Python tensorly.

Práctica 4.2a: Compresión y reducción de dimensionalidad usando redes neuronales artificiales: aplicaciones de autoencoders en compresión y visualización

Objetivo: Entender y aplicar los conceptos clave relacionados con compresión y reducción de dimensionalidad usando diferentes tipos de autoencoders, y su uso en compresión y visualización

Material: Se proporciona el notebook de Jupyter **Práctica4.2-alumnos.ipynb**, que contiene ejemplos de código que deben ser estudiados y ejecutados. Adicionalmente, algunos apartados de este notebook será necesario completar los ejercicios propuestos, implementando el código faltante y respondiendo a las preguntas incluidas.

1. Importar el *dataset* MNIST usando `tf.keras.datasets.mnist.load_data()`, y visualizar el tipo de datos que se usarán para el entrenamiento del autoencoder. Razonar la importancia de los siguientes parámetros en los resultados de la reducción de dimensionalidad: normalización de los datos, el número de capas y neuronas, la función de activación, el optimizador y la función de coste
2. Visualizar la arquitectura (número de neuronas y capas) del encoder, decoder y autoencoder usando la función `summary()`. ¿Son razonables las arquitecturas en relación a la simetría del autoencoder?
3. Usar la clase `TestEncoder` como callback para visualizar las representaciones latentes durante el entrenamiento del autoencoder. Probar como función de activación las siguientes funciones: RELU y sigmoide. ¿Qué diferencias significativas se observan durante el entrenamiento? ¿Y las representaciones latentes?
4. Analice la curva de aprendizaje usando el método la función `history()`, el cual permite visualizar los errores (en relación a la función de coste) durante el entrenamiento, y evaluando el impacto del número de épocas en el aprendizaje del autoencoder. Entrenar 2 modelos con los siguientes números de épocas: (1) 10; (2) 30.
5. Observar la dimensionalidad de los datos de entrada, y representaciones latentes. ¿Existe reducción de dimensionalidad?

Práctica 4.2b: Denoising Autoencoder Convolucional: regularización del entrenamiento y aplicaciones de autoencoders para clasificación

En el campo del aprendizaje profundo, los autoencoders han demostrado ser herramientas útiles para el aprendizaje no supervisado. Un autoencoder básico consiste en una red neuronal que aprende a codificar datos de entrada en una representación *comprimida* y luego reconstruirlos, minimizando la pérdida entre la entrada original y la salida reconstruida. La arquitectura del autoencoder, como hemos visto en clase, sigue una arquitectura simétrica entre encoder y decoder.

Entre las variantes de autoencoders, el Denoising Autoencoder (DAE) se destaca por su capacidad para aprender representaciones robustas al ruido, añadiendo un nivel de regularización que mejora la generalización del modelo. Este ruido puede ser generado de diferente naturaleza tal como Gaussiano, sal y pimienta, etc.

Cuando se combina esta arquitectura con redes convolucionales, conocidas por su eficacia en la extracción de características espaciales en datos como imágenes, surgen los **Denoising Autoencoders Convolucionales (CAE-DAE)**. Estos modelos no solo son útiles para la reconstrucción de datos, sino que también tienen aplicaciones relevantes en tareas supervisadas como la clasificación, donde las representaciones aprendidas pueden ser reutilizadas como características discriminativas.

En esta parte de la práctica exploraremos el impacto de la regularización durante el entrenamiento de autoencoders convolucionales y analizaremos cómo sus representaciones pueden ser aprovechadas para mejorar el rendimiento en tareas de clasificación.

Objetivo: Aplicar ruido estocástico como estrategia de regularización para el entrenamiento de autoencoders y generar representaciones latentes más robustas. Además, combinar el denoising autoencoder con capas convolucionales para crear un Denoising Autoencoder Convolucional que nos permita trabajar de manera más eficiente con imágenes.

Material: Se proporciona el notebook de Jupyter **Práctica4.2-alumnos.ipynb**, que contiene ejemplos de código que deben ser estudiados y ejecutados. Adicionalmente, algunos apartados de este notebook será necesario completar los ejercicios propuestos, implementando el código faltante y respondiendo a las preguntas incluidas.

1. Importar el dataset MNIST y visualizar el tipo de datos que se usarán para el entrenamiento del autoencoder. Comprobar la dimensionalidad de los datos de entrada. Dividir en conjuntos de train y test para el escenario de clasificación multiclas.
2. Visualizar la arquitectura (número de neuronas y capas) del encoder, decoder y autoencoder usando la función *summary()*. ¿Son razonables las arquitecturas en relación a la simetría del encoder y decoder?
3. Añadir ruido estocástico usando la función *add_noise()*. Realizar pruebas con diferentes valores de noise-ration, entre 0.2 y 0.8. Visualizar el impacto de añadir

ruido con diferentes noise-ratio en las imágenes. ¿Cuál es la importancia de controlar la potencia de ruido?

4. Generar datos ruidosos usando las imágenes MNIST, con una potencia de ruido de 0.4. Entrenar el autoencoder usando como entrada x-train-noisy y como salida los datos de train sin ruido.
5. Aplicar autoencoder.predict() y observar el resultado de las imágenes resultantes del autoencoder. ¿Qué aplicación tiene el denoising autoencoder convolutional?
6. Visualizar las curvas de aprendizaje del Multilayer Perceptron.

Práctica 4.2c: Autoencoder: aplicaciones para tareas predictivas

Objetivo: Aplicar una arquitectura de autoencoder simple para reducción de dimensionalidad y usar las representaciones latentes como vectores para un problema de clasificación. Comparar el rendimiento del autoencoder con el método de transformación linea Principal Component Analysis (PCA).

1. Descargar el dataset “Predict Students' Dropout and Academic Success” del repositorio UCI Irvine. [Link](#).
2. Revisar e indicar la dimensionalidad del dataset: número de muestras, tipo de variables (categóricas, numéricas) y desbalanceo de clases.
3. Realizar la etapa de análisis exploratorio de datos: visualización de distribución de datos, análisis de NaN values y outliers.
4. Normalizar los datos usando una estrategia que permite tener rangos dinámicos similares para las variables continuas y categóricas. Tenga en cuenta que las variables categóricas no son ordinales. Se puede usar técnicas de codificación de variables categóricas tanto supervisadas como no supervisadas.
5. Aplicar estrategia de under-sampling para balancear el dataset.
6. Entrenar diferentes modelos de autoencoder con tamaño de dimensión latente = 3, y variando el número de capas ocultas.
7. Entrenar un modelo PCA, visualizar la varianza acumulada y quedarse con las tres primeras componentes principales.
8. Entrenar un modelo predictivo Random Forest de la librería scikit-learn usando tanto la salida del autoencoder y PCA, y comparar los resultados predictivos. ¿Qué beneficios tiene el autoencoder vs PCA?
9. Entrenar un modelo predictivo Random Forest, usando el dataset inicial, previo a la reducción de dimensionalidad. Comparar resultados predictivos con el apartado anterior.
10. Visualizar en 3D las representaciones latentes del autoencoder y las componentes principales de PCA.

Nota: En todas las partes de la práctica, será fundamental implementar correctamente el código, comprender los conceptos subyacentes y analizar los resultados obtenidos.