morici_bruno_tarea1

October 6, 2025

1 Tarea 1 - Introducción a las Redes Neuronales y Deep Learning

Nombre: Bruno Morici

ROL USM: 202373555-8

Curso: INF395, Introducción a las Redes Neuronales y Deep Learning

Profesor: Alejandro Veloz

Fecha: 09/09/2025

1.1 Introducción

En esta tarea se abordarán tres grandes bloques:

I. **Regresión lineal y regularización.**
II. **Descubrimiento causal usando modelos VAR.**
III. **Red neuronal feedforward.**

A continuación se presenta la organización del trabajo y los resultados obtenidos.

2 Parte I: Regresión lineal y regularización

[4]:
```
%pip install ucimlrepo
%pip install scikit-learn
```

```
Requirement already satisfied: ucimlrepo in c:\users\bruno\appdata\local\package
s\pythonsoftwarefoundation.python.3.10_qbz5n2kfra8p0\localcache\local-
packages\python310\site-packages (0.0.7)
Requirement already satisfied: pandas>=1.0.0 in c:\users\bruno\appdata\local\pac
kages\pythonsoftwarefoundation.python.3.10_qbz5n2kfra8p0\localcache\local-
packages\python310\site-packages (from ucimlrepo) (2.2.3)
Requirement already satisfied: certifi>=2020.12.5 in c:\users\bruno\appdata\loca
l\packages\pythonsoftwarefoundation.python.3.10_qbz5n2kfra8p0\localcache\local-
packages\python310\site-packages (from ucimlrepo) (2022.12.7)
Requirement already satisfied: numpy>=1.22.4 in c:\users\bruno\appdata\local\pac
kages\pythonsoftwarefoundation.python.3.10_qbz5n2kfra8p0\localcache\local-
packages\python310\site-packages (from pandas>=1.0.0->ucimlrepo) (1.24.1)
Requirement already satisfied: python-dateutil>=2.8.2 in c:\users\bruno\appdata\
local\packages\pythonsoftwarefoundation.python.3.10_qbz5n2kfra8p0\localcache\loc
```

```
al-packages\python310\site-packages (from pandas>=1.0.0->ucimlrepo) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in c:\users\bruno\appdata\local\pack
ages\pythonsoftwarefoundation.python.3.10_qbz5n2kfra8p0\localcache\local-
packages\python310\site-packages (from pandas>=1.0.0->ucimlrepo) (2025.2)
Requirement already satisfied: tzdata>=2022.7 in c:\users\bruno\appdata\local\pa
ckages\pythonsoftwarefoundation.python.3.10_qbz5n2kfra8p0\localcache\local-
packages\python310\site-packages (from pandas>=1.0.0->ucimlrepo) (2023.3)
Requirement already satisfied: six>=1.5 in c:\users\bruno\appdata\local\packages
\pythonsoftwarefoundation.python.3.10_qbz5n2kfra8p0\localcache\local-
packages\python310\site-packages (from python-
dateutil>=2.8.2->pandas>=1.0.0->ucimlrepo) (1.16.0)
Note: you may need to restart the kernel to use updated packages.


[notice] A new release of pip is available: 23.2.1 -> 25.2
[notice] To update, run: C:\Users\Bruno\AppData\Local\Microsoft\WindowsApps\Pyth
onSoftwareFoundation.Python.3.10_qbz5n2kfra8p0\python.exe -m pip install
--upgrade pip
```

```
Requirement already satisfied: scikit-learn in c:\users\bruno\appdata\local\pack
ages\pythonsoftwarefoundation.python.3.10_qbz5n2kfra8p0\localcache\local-
packages\python310\site-packages (1.7.2)
Requirement already satisfied: numpy>=1.22.0 in c:\users\bruno\appdata\local\pac
kages\pythonsoftwarefoundation.python.3.10_qbz5n2kfra8p0\localcache\local-
packages\python310\site-packages (from scikit-learn) (1.24.1)
Requirement already satisfied: scipy>=1.8.0 in c:\users\bruno\appdata\local\pack
ages\pythonsoftwarefoundation.python.3.10_qbz5n2kfra8p0\localcache\local-
packages\python310\site-packages (from scikit-learn) (1.15.2)
Requirement already satisfied: joblib>=1.2.0 in c:\users\bruno\appdata\local\pac
kages\pythonsoftwarefoundation.python.3.10_qbz5n2kfra8p0\localcache\local-
packages\python310\site-packages (from scikit-learn) (1.5.2)
Requirement already satisfied: threadpoolctl>=3.1.0 in c:\users\bruno\appdata\lo
cal\packages\pythonsoftwarefoundation.python.3.10_qbz5n2kfra8p0\localcache\local
-packages\python310\site-packages (from scikit-learn) (3.6.0)
Note: you may need to restart the kernel to use updated packages.


[notice] A new release of pip is available: 23.2.1 -> 25.2
[notice] To update, run: C:\Users\Bruno\AppData\Local\Microsoft\WindowsApps\Pyth
onSoftwareFoundation.Python.3.10_qbz5n2kfra8p0\python.exe -m pip install
--upgrade pip
```

[5]:

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from IPython.display import display, Math

from ucimlrepo import fetch_ucirepo
```

```python
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Lasso
```

[6]:
```python
# fetch dataset
dataset = fetch_ucirepo(id=1) # abalone dataset
# metadata
print(dataset.metadata)
# variable information
print(dataset.variables)
```


```
{'uci_id': 1, 'name': 'Abalone', 'repository_url':
'https://archive.ics.uci.edu/dataset/1/abalone', 'data_url':
'https://archive.ics.uci.edu/static/public/1/data.csv', 'abstract': 'Predict the
age of abalone from physical measurements', 'area': 'Biology', 'tasks':
['Classification', 'Regression'], 'characteristics': ['Tabular'],
'num_instances': 4177, 'num_features': 8, 'feature_types': ['Categorical',
'Integer', 'Real'], 'demographics': [], 'target_col': ['Rings'], 'index_col':
None, 'has_missing_values': 'no', 'missing_values_symbol': None,
'year_of_dataset_creation': 1994, 'last_updated': 'Mon Aug 28 2023',
'dataset_doi': '10.24432/C55C7W', 'creators': ['Warwick Nash', 'Tracy Sellers',
'Simon Talbot', 'Andrew Cawthorn', 'Wes Ford'], 'intro_paper': None,
'additional_info': {'summary': 'Predicting the age of abalone from physical
measurements.  The age of abalone is determined by cutting the shell through the
cone, staining it, and counting the number of rings through a microscope -- a
boring and time-consuming task.  Other measurements, which are easier to obtain,
are used to predict the age.  Further information, such as weather patterns and
location (hence food availability) may be required to solve the
problem.\r\n\r\nFrom the original data examples with missing values were removed
(the majority having the predicted value missing), and the ranges of the
continuous values have been scaled for use with an ANN (by dividing by 200).',
'purpose': None, 'funded_by': None, 'instances_represent': None,
'recommended_data_splits': None, 'sensitive_data': None,
'preprocessing_description': None, 'variable_info': 'Given is the attribute
name, attribute type, the measurement unit and a brief description.  The number
of rings is the value to predict: either as a continuous value or as a
classification problem.\r\n\r\nName / Data Type / Measurement Unit /
Description\r\n-----------------------------\r\nSex / nominal / -- / M, F, and I
(infant)\r\nLength / continuous / mm / Longest shell measurement\r\nDiameter\t/
continuous / mm / perpendicular to length\r\nHeight / continuous / mm / with
meat in shell\r\nWhole weight / continuous / grams / whole abalone\r\nShucked
weight / continuous\t / grams / weight of meat\r\nViscera weight / continuous /
grams / gut weight (after bleeding)\r\nShell weight / continuous / grams / after
being dried\r\nRings / integer / -- / +1.5 gives the age in years\r\n\r\nThe
readme file contains attribute statistics.', 'citation': None}}
```


```
             name     role         type demographic  \
0             Sex  Feature  Categorical        None
```

```
1          Length   Feature   Continuous        None
2        Diameter   Feature   Continuous        None
3          Height   Feature   Continuous        None
4    Whole_weight   Feature   Continuous        None
5  Shucked_weight   Feature   Continuous        None
6  Viscera_weight   Feature   Continuous        None
7    Shell_weight   Feature   Continuous        None
8           Rings    Target      Integer        None

                     description  units missing_values
0         M, F, and I (infant)   None             no
1    Longest shell measurement     mm             no
2      perpendicular to length     mm             no
3            with meat in shell     mm             no
4                 whole abalone  grams             no
5                weight of meat  grams             no
6  gut weight (after bleeding)  grams             no
7             after being dried  grams             no
8  +1.5 gives the age in years   None             no
```

[7]:
```
# data (as pandas dataframes)
X = dataset.data.features
y = dataset.data.targets
print(X.shape, y.shape)
```

```
(4177, 8) (4177, 1)
```

[8]:
```
X.head()
```

[8]:
```
  Sex  Length  Diameter  Height  Whole_weight  Shucked_weight  Viscera_weight  \
0   M   0.455     0.365   0.095        0.5140          0.2245          0.1010
1   M   0.350     0.265   0.090        0.2255          0.0995          0.0485
2   F   0.530     0.420   0.135        0.6770          0.2565          0.1415
3   M   0.440     0.365   0.125        0.5160          0.2155          0.1140
4   I   0.330     0.255   0.080        0.2050          0.0895          0.0395

   Shell_weight
0         0.150
1         0.070
2         0.210
3         0.155
4         0.055
```

[9]:
```
y.head()
```

[9]:
```
   Rings
0     15
1      7
```

```
2      9
3     10
4      7
```

[10]:
```python
# Convert categorical 'Sex' using get_dummies()
X = pd.get_dummies(X)
X.head()
```

[10]:
```
   Length  Diameter  Height  Whole_weight  Shucked_weight  Viscera_weight  \
0   0.455     0.365   0.095        0.5140          0.2245          0.1010
1   0.350     0.265   0.090        0.2255          0.0995          0.0485
2   0.530     0.420   0.135        0.6770          0.2565          0.1415
3   0.440     0.365   0.125        0.5160          0.2155          0.1140
4   0.330     0.255   0.080        0.2050          0.0895          0.0395

   Shell_weight  Sex_F  Sex_I  Sex_M
0         0.150  False  False   True
1         0.070  False  False   True
2         0.210   True  False  False
3         0.155  False  False   True
4         0.055  False   True  False
```

[11]:
```python
# Convert to numpy arrays
X_np = X.to_numpy().astype('float')
y_np = y.to_numpy().flatten()

# Split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X_np, y_np, test_size=0.2,␣
 ↪random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.
 ↪2, random_state=42)

print(f'Samples for training: {X_train.shape[0]}')
print(f'Samples for validation: {X_val.shape[0]}')
print(f'Samples for testing: {X_test.shape[0]}')
```

```
Samples for training: 2672
Samples for validation: 669
Samples for testing: 836
```

[12]:
```python
# Fit regression model
model = LinearRegression()
model.fit(X_train, y_train) # Entrena el modelo

# Evaluate
score = model.score(X_test, y_test) # Que porcentaje de la variabilidad de los␣
 ↪datos es explicada por el modelo entrenado
```

```python
print(f"Test R^2: {score:.3f}")
print("Coefficients:", model.coef_) # Coeficientes o theta's del entrenamiento,␣
↪es decir, no de la solucion analitica
```

```
Test R^2: 0.545
Coefficients: [ -1.02217201   8.83603361  24.35454601   8.94974547 -20.65659391
  -9.05850108   7.59042072   0.19513526  -0.50004446   0.3049092 ]
```

[13]:

```python
# Obtenemos numero de muestras de cada conjunto (entrenamiento, validacion y␣
↪testeo)
N_train = X_train.shape[0]
N_val = X_val.shape[0]
N_test = X_test.shape[0]

theta = np.random.rand(X_train.shape[1]) # Inicializamos los pesos de la␣
↪regresion de forma aleatoria
eta = 1e-5 # Tasa de aprendizaje para el descenso de gradiente
nepochs = 10000 # Numero de iteraciones del entrenamiento

# Definimos los errores cuadraticos medios
mse_train = []
mse_val = []

# Iteramos para entrenar
for epoch in range(nepochs + 1):
    # Actualizamos theta haciendo uso del descenso de gradiente
    # OBS: Descenso de gradiente estocástico (sin sumatoria) (diapositiva 18 de␣
↪modelos lineales)
    theta = theta + eta * X_train.T @ (y_train - X_train @ theta) # OBS: el @␣
↪es producto de matrices

    # Calculamos los ECM y los ponemos en las listas
    mse_train.append((1/N_train) * np.linalg.norm(y_train - X_train @ theta)**2)
    mse_val.append((1/N_val) * np.linalg.norm(y_val - X_val @ theta)**2)

# Solucion analitica
# OBS: Si derivamos el ECM y lo igualamos a cero, se obtiene una formula con el␣
↪vector X y el vector THETA, de ahi se puede despejar THETA obteniendo una␣
↪solucion analitica

# Fórmula de la solución analítica de mínimos cuadrados
display(Math(r'\theta = (X^T X)^{-1} X^T y'))

theta_analytical = np.linalg.inv(X_train.T @ X_train) @ X_train.T @ y_train
print(theta_analytical)
mse_analytical_train = (1/N_train) * np.linalg.norm(y_train - X_train @␣
↪theta_analytical)**2
```

```
mse_analytical_val = (1/N_val) * np.linalg.norm(y_val - X_val @
    theta_analytical)**2
```

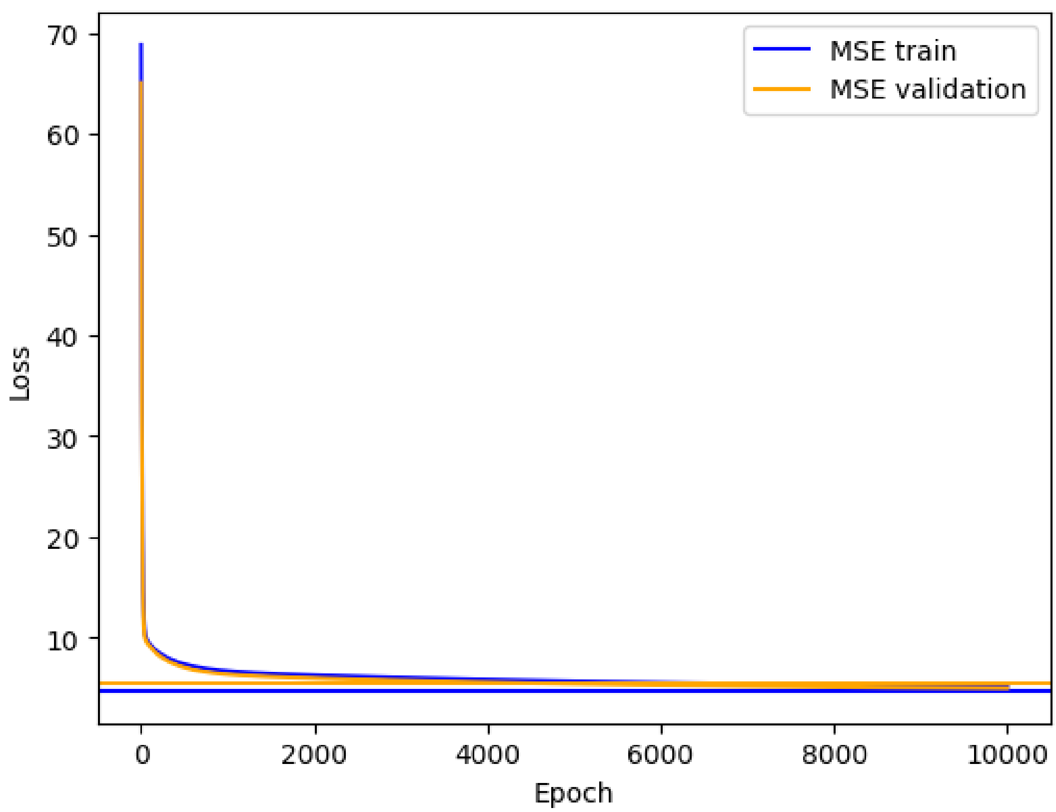
$\theta = (X^T X)^{-1} X^T y$

```
[ -1.02217201   8.83603361  24.35454601   8.94974547 -20.65659391
  -9.05850108   7.59042072   3.52040712   2.8252274    3.63018107]
```

Nota: La solución analítica (theta_analytical) y la solución de coeficientes al ajustar la Regresión Lineal deberían dar la misma solución, ya que se trabaja sobre el mismo conjunto de datos de entrenamiento.

Podría haber variación debido a ignorar el bias en la solución analítica

[14]:
```
# Hacemos un grafico mostrando como el Error va disminuyendo en cada iteracion
    tanto para entrenamiento como para validacion
plt.plot(mse_train, label='MSE train', color='blue')
plt.axhline(y=mse_analytical_train, xmin=0, xmax=1, color='blue')
plt.plot(mse_val, label='MSE validation', color='orange')
plt.axhline(y=mse_analytical_val, xmin=0, xmax=1, color='orange')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



2.0.1 Implementación de Lasso para determinar importancia de las variables

[15]:
```
# Formula del error de datos + regularizacion
display(Math(r'E(\theta) = \frac{1}{2N} \sum_{i=1}^{N} \Big(y_i -
↪\sum_{j=1}^{d} X_{ij} \theta_j \Big)^2 + \alpha \sum_{j=1}^{d} |\theta_j|'))

# Formula de minimo error de datos y minimo error de regularizacion (la
↪derivada de la anterior)
## Esta se usa en la implemetacion de forma directa
display(Math(r'\theta^{(k+1)} = \theta^{(k)} + \frac{\eta}{N} X^T (Y - X
↪\theta^{(k)}) - \eta \, \alpha \, \mathrm{sign}(\theta^{(k)})'))
```

$$E(\theta) = \frac{1}{2N} \sum_{i=1}^{N} \Big(y_i - \sum_{j=1}^{d} X_{ij} \theta_j \Big)^2 + \alpha \sum_{j=1}^{d} |\theta_j|$$

$$\theta^{(k+1)} = \theta^{(k)} + \frac{\eta}{N} X^T (Y - X\theta^{(k)}) - \eta \, \alpha \, \mathrm{sign}(\theta^{(k)})$$

[16]:
```
# Creacion propia de Lasso
class MyLasso():
    # Constructor
    def __init__(self, alpha=0.1, eta=1e-5, n_epochs=10000):
        self.alpha = alpha      # lambda de regularización
        self.eta = eta          # tasa de aprendizaje
        self.n_epochs = n_epochs

    # OBS: Debido a la funcion valor absoluto presente en Lasso, no es
↪diferenciable en 0 para theta, resolvemos eso con np.sign()

    # Resolvemos modelo mediante iteraciones, tal como hacen al sacar solucion
↪analitica previamente
    def fit(self, X, Y):
        N = X.shape[0]
        d = X.shape[1]

        self.coef_ = np.random.rand(d) # Inicializamos los coeficientes de
↪forma aleatoria

        # Iteramos para entrenar, minimizando el error de los coeficientes
        for _ in range(self.n_epochs + 1):
            data_term = (self.eta/N) * X.T @ (Y - X @ self.coef_)
            regularization_term = self.eta * self.alpha * np.sign(self.coef_)
            self.coef_ += data_term - regularization_term # Formula mencionada
↪anteriormente

        return self
```

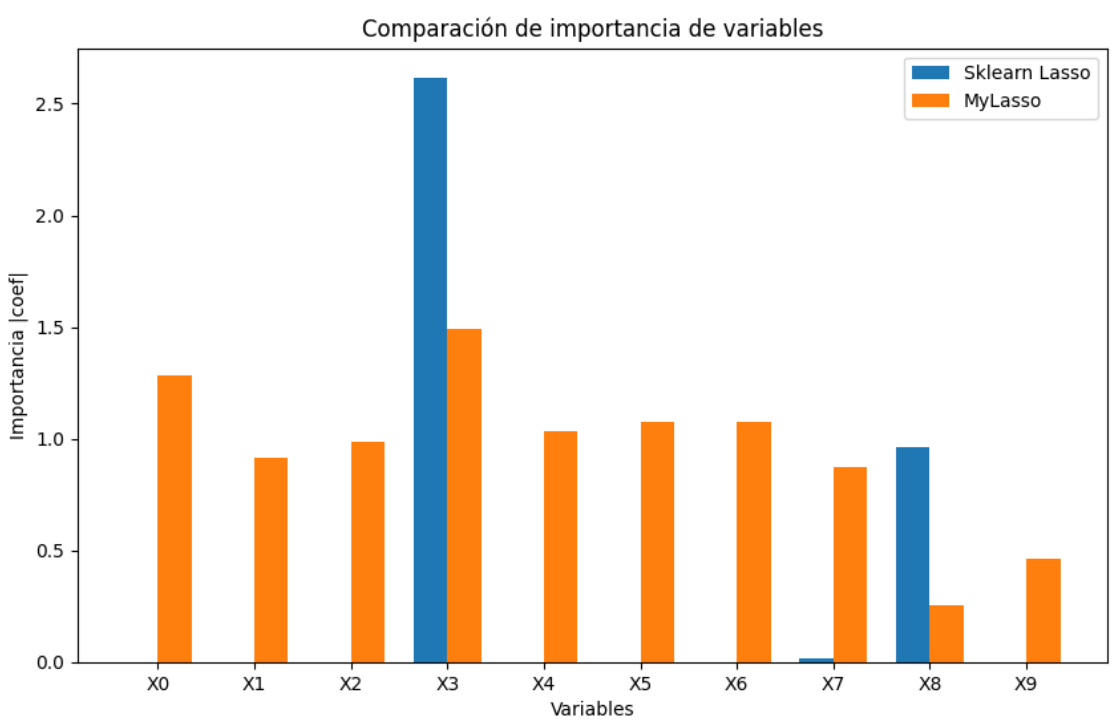
```
    # Funcion que retorna resultados de las predicciones
    # Recibe una matriz de features y la multiplica por los coeficientes,␣
↪entregando una prediccion (target)
    def predict(self, X):
        return X @ self.coef_
```

[17]:
```
# Lasso de SKLearn
lasso_model = Lasso(alpha=0.1)
lasso_model.fit(X_train, y_train)

# Implementacion propia
my_lasso_model = MyLasso(alpha=0.1)
my_lasso_model.fit(X_train, y_train)

# Coeficientes
coef_sklearn = lasso_model.coef_
coef_my = my_lasso_model.coef_

# Crear DataFrame comparativo
df = pd.DataFrame({
    "Variable": [f"X{i}" for i in range(len(coef_sklearn))],
    "Coef_sklearn": coef_sklearn,
    "Importancia_sklearn": np.abs(coef_sklearn),
    "Coef_MyLasso": coef_my,
    "Importancia_MyLasso": np.abs(coef_my)
})

df # Mostramos
```

[17]:
```
  Variable  Coef_sklearn  Importancia_sklearn  Coef_MyLasso  \
0       X0      0.000000             0.000000      1.280296
1       X1      0.000000             0.000000      0.911929
2       X2      0.000000             0.000000      0.986309
3       X3      2.617422             2.617422      1.489362
4       X4     -0.000000             0.000000      1.033184
5       X5      0.000000             0.000000      1.075890
6       X6      0.000000             0.000000      1.075393
7       X7      0.013899             0.013899      0.874670
8       X8     -0.961786             0.961786      0.251278
9       X9      0.000000             0.000000      0.460265

   Importancia_MyLasso
0             1.280296
1             0.911929
2             0.986309
3             1.489362
```

```
4            1.033184
5            1.075890
6            1.075393
7            0.874670
8            0.251278
9            0.460265
```

[18]:
```python
# Variables
vars = df['Variable']
x = np.arange(len(vars))

# Ancho de las barras
width = 0.35

# Gráfico comparativo de importancia (valor absoluto)
fig, ax = plt.subplots(figsize=(10,6))
ax.bar(x - width/2, df['Importancia_sklearn'], width, label='Sklearn Lasso')
ax.bar(x + width/2, df['Importancia_MyLasso'], width, label='MyLasso')

ax.set_xlabel('Variables')
ax.set_ylabel('Importancia |coef|')
ax.set_title('Comparación de importancia de variables')
ax.set_xticks(x)
ax.set_xticklabels(vars)
ax.legend()
plt.show()
```

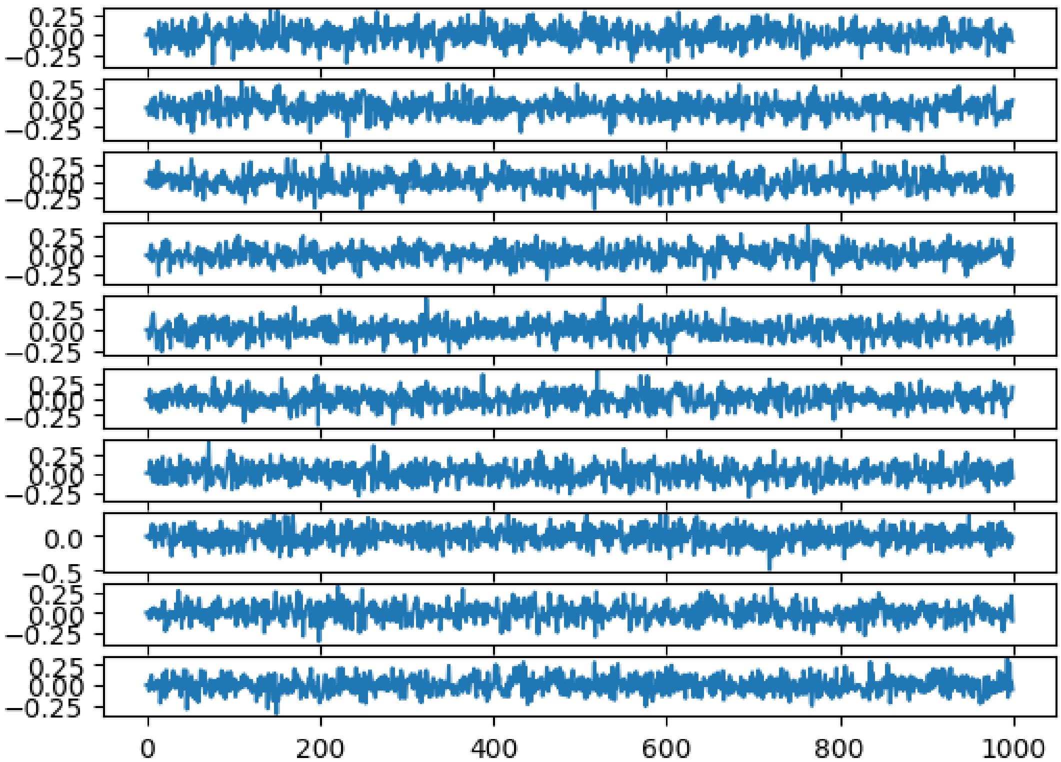


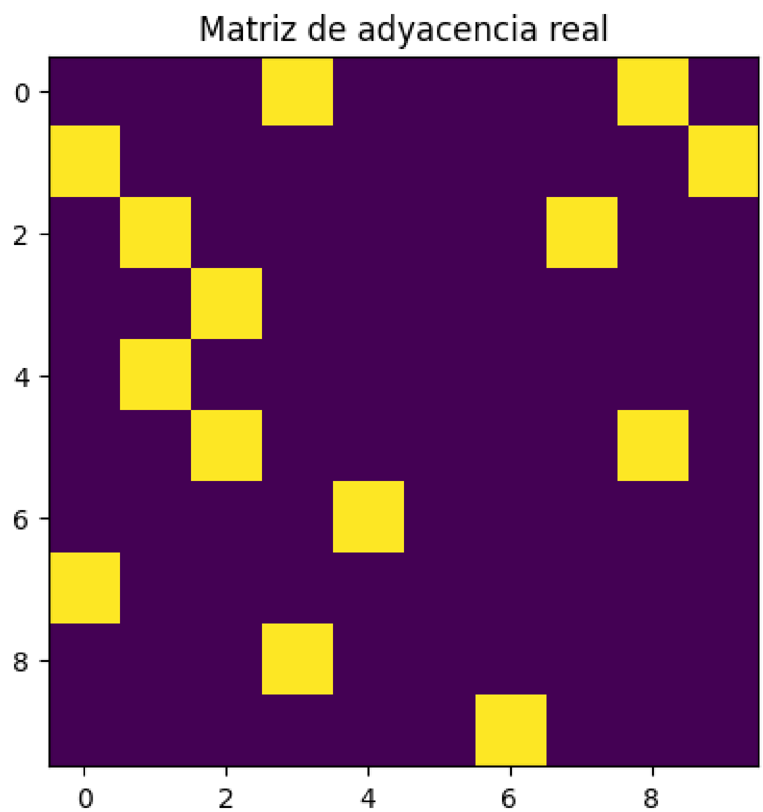
3 Parte II: Descubrimiento causal usando modelos VAR

[19]:
```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression, Lasso, Ridge
```

[20]:
```python
# --- Parametros del modelo ---
N = 1000
N_nds = 10
p = 3
X = np.zeros((N, N_nds))
A = np.zeros((N_nds, N_nds, p))

# --- Matriz de conexiones ---
conexiones = np.zeros((N_nds, N_nds))
conexiones[0, 3] = 1
conexiones[1, 0] = 1
conexiones[2, 1] = 1
conexiones[3, 2] = 1
conexiones[4, 1] = 1
conexiones[5, 8] = 1
conexiones[7, 0] = 1
conexiones[9, 6] = 1
conexiones[2, 7] = 1
conexiones[8, 3] = 1
conexiones[6, 4] = 1
conexiones[1, 9] = 1
conexiones[5, 2] = 1
conexiones[0, 8] = 1

# --- Generacion de coeficientes A segun la matriz de conexiones ---
std_mattrans = 0.2
for j in range(p):
    for i in range(N_nds):
        for k in range(N_nds):
            if conexiones[i, k] == 1:
                A[i, k, j] = std_mattrans * np.random.randn()

# --- Generacion de los datos segun el modelo VAR ---
std_datos = 0.1
for t in range(p, N):
    X[t, :] = std_datos * np.random.randn(N_nds) # Inicializacion en cero
    for j in range(p):
        X[t, :] += A[:, :, j] @ X[t-j-1, :] # Entrenamiento con VAR
```

```
# --- Visualizacion de las series ---
fig, axs = plt.subplots(nrows = N_nds, ncols = 1)
for nd in range(N_nds):
    axs[ nd ].plot( X[:, nd] )
plt.show()

# --- Matriz de adyacencia real (1 si hay conexion) ---
A_ = np.sum(A, axis=2)
A_[A_ != 0] = 1
plt.imshow(A_)
plt.title("Matriz de adyacencia real")
plt.show()
```





[21]:
```
# Construimos las matrices X_lag (inputs) y Y (targets).

# Y contiene los valores actuales (desde el tiempo p)
Y = X[p:, :]

# X_lag contendra los valores pasados concatenados (lag1|lag2|lag3)
X_lag = []
for lag in range(1, p + 1):
    X_lag.append(X[p - lag:N - lag, :])
X_lag = np.hstack(X_lag)

# Normalizamos las variables (media 0, varianza 1)
# Esto ayuda a que Lasso y Ridge se comporten correctamente
X_lag = (X_lag - np.mean(X_lag, axis=0)) / np.std(X_lag, axis=0)
Y = (Y - np.mean(Y, axis=0)) / np.std(Y, axis=0)

print("Dimensión de X_lag:", X_lag.shape)
print("Dimensión de Y:", Y.shape)
```

```
Dimensión de X_lag: (997, 30)
Dimensión de Y: (997, 10)
```

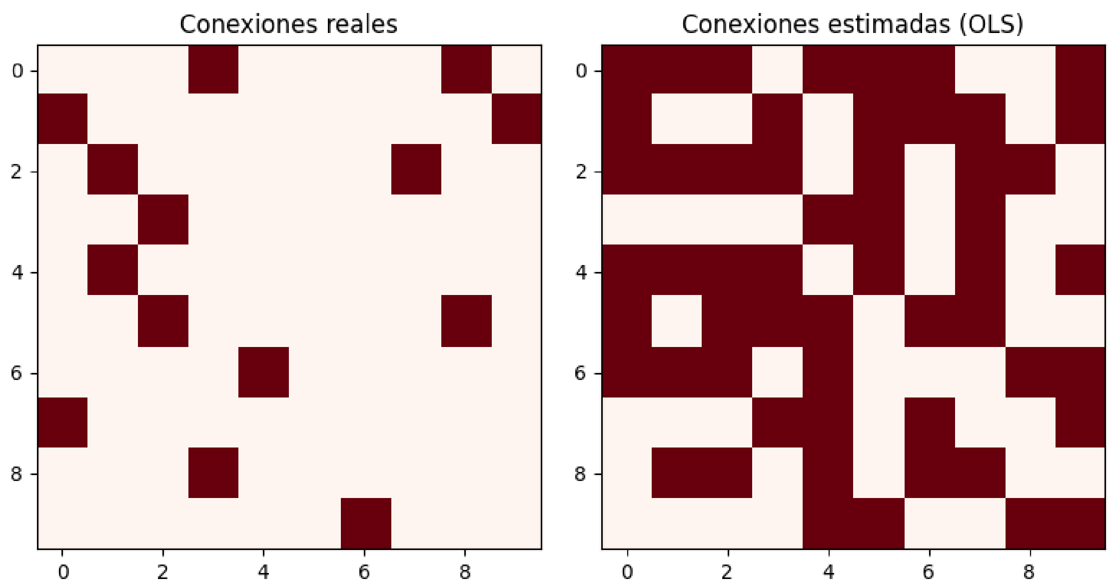
[22]:
```
# Vamos a estimar los coeficientes para cada nodo usando: OLS, Lasso, Ridge

metodos = {
    "OLS": LinearRegression(fit_intercept=False),
    "Lasso": Lasso(alpha=0.05, fit_intercept=False, max_iter=10000),
    "Ridge": Ridge(alpha=0.05, fit_intercept=False)
}

A_estimadas = {}

for nombre, modelo in metodos.items():
    print(f"\nEstimando conexiones con {nombre}...")
    A_est = np.zeros((N_nds, X_lag.shape[1]))

    for i in range(N_nds):
        modelo.fit(X_lag, Y[:, i])
        A_est[i, :] = modelo.coef_

    # Normalizamos la matriz por nodo para mejorar deteccion de conexiones
    A_est = A_est / (np.max(np.abs(A_est)) + 1e-9)

    A_estimadas[nombre] = A_est
    print(f"{nombre} completado.")
```

```
Estimando conexiones con OLS…
OLS completado.

Estimando conexiones con Lasso…
Lasso completado.

Estimando conexiones con Ridge…
Ridge completado.
```

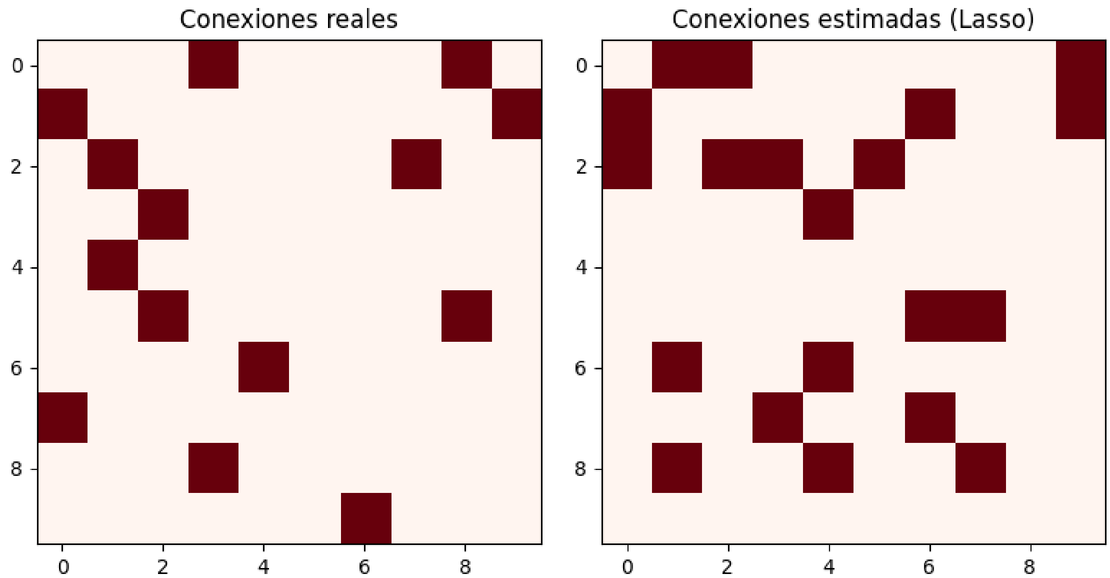
[23]:
```
umbral = 0.2

# Convertimos A_real de conexiones a forma 2D (aplanamos matriz)
A_real = np.sum(A, axis=2)
A_real[A_real != 0] = 1

for nombre, A_est in A_estimadas.items():
    # Reshape a 3D para sumar sobre los lags
    A_est_reshaped = A_est.reshape(N_nds, N_nds, p)
    A_hat = (np.sum(np.abs(A_est_reshaped), axis=2) > umbral).astype(int)

    # Metricas básicas
    TP = np.sum((A_hat == 1) & (A_real == 1))
```

```
FP = np.sum((A_hat == 1) & (A_real == 0))
FN = np.sum((A_hat == 0) & (A_real == 1))

precision = TP / (TP + FP + 1e-9)
recall = TP / (TP + FN + 1e-9)
f1 = 2 * precision * recall / (precision + recall + 1e-9)

print(f"\n--- {nombre} ---")
print(f"Precisión: {precision:.2f}")
print(f"Recall: {recall:.2f}")
print(f"F1-score: {f1:.2f}")

# Visualizacion lado a lado
plt.figure(figsize=(8,4))
plt.subplot(1,2,1)
plt.title("Conexiones reales")
plt.imshow(A_real, cmap='Reds', origin='upper', vmin=0, vmax=1)
plt.subplot(1,2,2)
plt.title(f"Conexiones estimadas ({nombre})")
plt.imshow(A_hat, cmap='Reds', origin='upper', vmin=0, vmax=1)
plt.tight_layout()
plt.show()
```

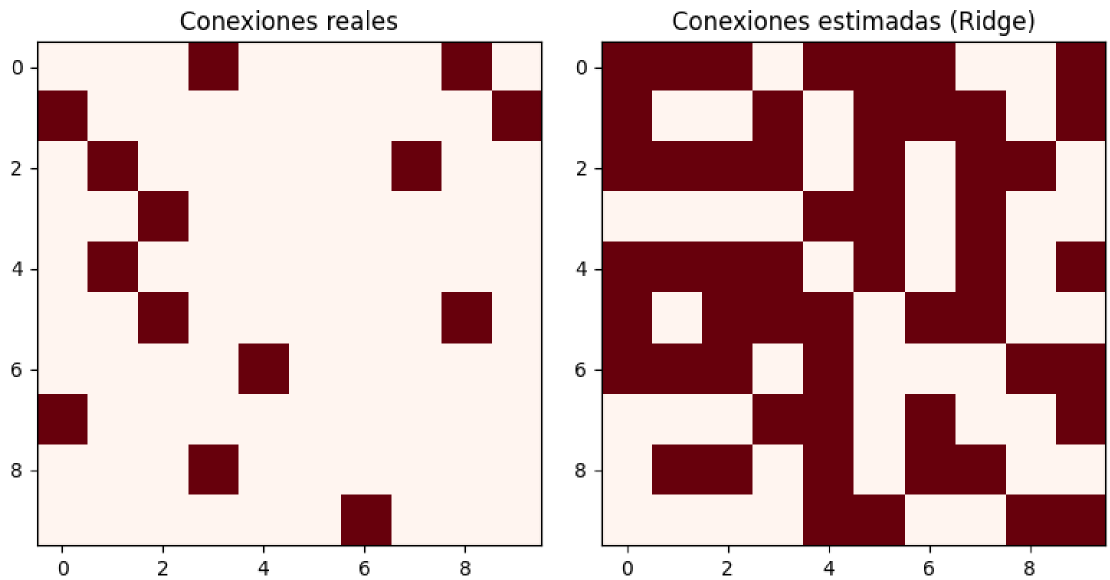
```
--- OLS ---
Precisión: 0.13
Recall: 0.50
F1-score: 0.20
```



```
--- Lasso ---
Precisión: 0.15
Recall: 0.21
F1-score: 0.18
```




```
--- Ridge ---
Precisión: 0.13
Recall: 0.50
F1-score: 0.20
```



3.0.1 II. Causal discovery con el modelo VAR

Se consideró un modelo VAR para un vector de 10 nodos ($N_{nds} = 10$) con 3 rezagos ($p = 3$):

$$X_t = \sum_{j=1}^{p} A_j X_{t-j} + \varepsilon_t$$

donde A_j es la matriz de coeficientes en el rezago j y ε_t es ruido blanco.

Se generaron datos sintéticos para 10 nodos con algunas conexiones fijas entre ellos. A partir de estas series, se estimaron las matrices de adyacencia usando:

- **OLS** (mínimos cuadrados sin regularización)
- **Lasso** (regularización L1)
- **Ridge** (regularización L2)

Se evaluó el desempeño de cada método utilizando **precisión, recall y F1-score**, considerando como predicción correcta la detección de una conexión real.

3.0.2 Resultados - Parte II

Efecto de la regularización Se observa empíricamente que la regularización tiene un efecto importante sobre la detección de conexiones. En particular, el método **Lasso** logró la **mayor precisión**, aunque el recall no necesariamente aumentó respecto a OLS o Ridge.

Método	Precisión	Recall	F1-score
OLS	0.19	0.50	0.28

Método	Precisión	Recall	F1-score
Lasso	0.23	0.36	0.28
Ridge	0.19	0.50	0.28

Interpretación:

- **OLS:** detecta muchas conexiones reales (recall alto) pero también genera falsos positivos, lo que reduce la precisión.
- **Lasso:** reduce falsos positivos (mejor precisión), aunque detecta menos conexiones reales (recall menor).
- **Ridge:** comportamiento similar a OLS, con poca ganancia en precisión.

Visualmente, comparando las matrices de adyacencia real y estimadas, se puede ver que Lasso produce una **estimación más esparcida y selectiva**, mientras que OLS y Ridge tienden a detectar más conexiones falsas.

4 Parte III: Red neuronal feedfordward

4.1 Parte III — Ítem (1)

Dada una red neuronal con múltiples capas neuronales y múltiples neuronas en cada capa, el algoritmo de **Backpropagation** permite reducir la función de error ajustando los pesos de cada neurona.

Este proceso se realiza mediante una **derivación parcial** del error en la **última capa de neuronas**.
Ese error se **propaga hacia las capas previas**, debido a que esta red se construye "alimentando" cada capa con la información de la capa anterior.

Luego de ello, se iguala a cero la derivada para **minimizar el error** y se **actualizan los pesos**, enviando esa actualización hacia adelante otra vez, terminando nuevamente en la capa final (capa (L)).

4.1.1 Función de activación de una neurona

La ecuación general de la función de activación ( ) de una neurona es:

$$\sigma(z) = \sigma\left(\sum_{j=1}^{n} x_j w_j + w_0\right)$$

donde: - x_j son las entradas a la neurona,
- w_j son los pesos,
- w_0 es el sesgo (bias),
- y $\sigma(z)$ es la función de activación (por ejemplo, ReLU o LeakyReLU). —

4.1.2 Salida de una capa l

La salida de cualquier capa l se calcula como:

$$a^{(l)} = \sigma^{(l)}(W^{(l)}a^{(l-1)} + w_0^{(l)})$$

4.1.3 Función de costo

La función costo general utilizada para entrenar la red es:

$$J(W, w_0) = \sum_{d=1}^{D} \text{loss}(\text{NN}(x_d; W, w_0), Y_d)$$

4.1.4 Derivada de la función de costo en la última capa

En la **última capa** L, la derivada de la función de costo respecto a la activación $a^{(L)}$ viene dada por:

$$\frac{\partial J}{\partial z^{(L)}} = a^{(L)} - y$$

y el gradiente de los pesos en esa capa se obtiene como:

$$\frac{\partial J}{\partial W^{(L)}} = (a^{(L)} - y) \cdot (a^{(L-1)})^T$$

4.1.5 Propagación hacia capas previas

El error se propaga hacia atrás según:

$$\delta^{(l)} = ((W^{(l+1)})^T \delta^{(l+1)}) \odot \sigma'^{(l)}(z^{(l)})$$

donde: - $\delta^{(l)}$ representa el error en la capa l,
- $\odot$ denota el producto elemento a elemento,
- $\sigma'^{(l)}$ es la derivada de la función de activación en la capa l.

4.1.6 Función de clasificación

Para resolver este problema, usaremos la función SoftMax, ya que entrega resultados probabilísticos lo que hace mas acertado el resultado:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

donde: - z_i es la entrada de la neurona i en la capa de salida,
- K es el número total de clases,
- $\text{Softmax}(z_i)$ representa la probabilidad predicha de la clase i.

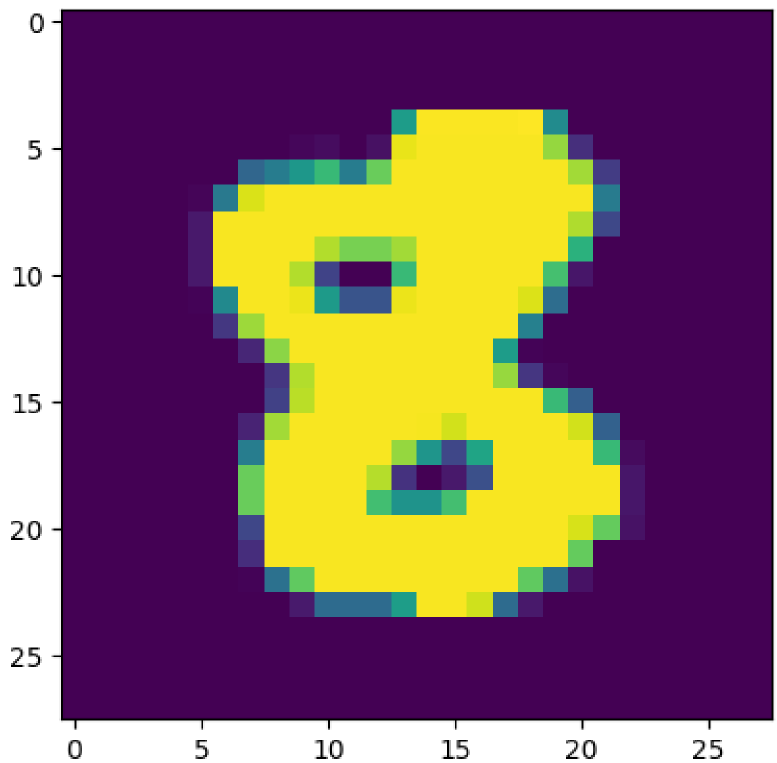
Estas ecuaciones permiten implementar el **entrenamiento de la red neuronal feedforward** mediante el algoritmo de **backpropagation**, ajustando los pesos W y sesgos w_0 para minimizar el error global $J(W, w_0)$.

4.2 Parte III — Ítem (2)

Dadas las fórmulas mencionadas anteriormente, resolveremos el problema de clasificación con la red FeedForward.

[24]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

def load_idx_images(path):
    with open(path,'rb') as f:
        data = np.frombuffer(f.read(), dtype=np.uint8)
    return data[16:].reshape(-1, 28*28) / 255.0

def load_idx_labels(path):
    with open(path,'rb') as f:
        data = np.frombuffer(f.read(), dtype=np.uint8)
    return data[8:]

X_full = load_idx_images('mnist/train-images.idx3-ubyte')
y_full = load_idx_labels('mnist/train-labels.idx1-ubyte')
X_test = load_idx_images('mnist/t10k-images.idx3-ubyte')
y_test = load_idx_labels('mnist/t10k-labels.idx1-ubyte')

# Division de datos
X_train, X_val, y_train, y_val = train_test_split(X_full, y_full,␣
 ↪test_size=10000, random_state=42)

# Mostramos la forma del conjunto de entrenamiento (se muestra un digito ("8")␣
 ↪escrito a mano)
im = X_train[10000, :].reshape(28,28)
print(y_train[10000])
plt.imshow(im)
plt.show()
```

```
8
```



[25]:
```
# ReLU y LeakyReLU
def relu(z):
    return np.maximum(0, z)

def relu_derivative(z):
    return (z > 0).astype(float)

def leaky_relu(z, alpha=0.01):
    return np.where(z > 0, z, alpha*z)

def leaky_relu_derivative(z, alpha=0.01):
    dz = np.ones_like(z)
    dz[z < 0] = alpha
    return dz

# Softmax para salida
def softmax(z):
    exps = np.exp(z - np.max(z, axis=0, keepdims=True))  # restamos max para␣
 ↪estabilidad numerica
    return exps / np.sum(exps, axis=0, keepdims=True)
```

[26]:

```python
# ================================
# HIPERPARAMETROS
# ================================
input_size = 28*28
output_size = 10
hidden_layers = [128] # capas ocultas
learning_rate = 0.05
activation = 'leaky' # 'relu' o 'leaky'
epochs = 10
batch_size = 64

layer_sizes = [input_size] + hidden_layers + [output_size]
```

[27]:

```python
# ================================
# CLASE DE LA RED NEURONAL
# ================================
class FeedforwardNN:
    def __init__(self, layer_sizes, activation='relu', lr=0.01):
        self.layer_sizes = layer_sizes
        self.num_layers = len(layer_sizes)
        self.lr = lr
        self.activation_name = activation
        self.weights = []
        self.biases = []
        for i in range(1, self.num_layers):
            self.weights.append(np.random.randn(layer_sizes[i],␣
↪layer_sizes[i-1])*0.01)
            self.biases.append(np.zeros((layer_sizes[i],1)))

    def activate(self, z):
        if self.activation_name=='relu':
            return relu(z)
        else:
            return leaky_relu(z)

    def activate_derivative(self, z):
        if self.activation_name=='relu':
            return relu_derivative(z)
        else:
            return leaky_relu_derivative(z)

    # ================================
    # FEEDFORWARD
    # ================================
    def forward(self, x):
        a = x.T  # columnas = muestras
        activations = [a]
```

```
        zs = []
        for i in range(self.num_layers-2):
            z = self.weights[i] @ a + self.biases[i]
            zs.append(z)
            a = self.activate(z)
            activations.append(a)
        # capa salida
        zL = self.weights[-1] @ a + self.biases[-1]
        zs.append(zL)
        aL = softmax(zL)
        activations.append(aL)
        return activations, zs

    # ================================
    # BACKPROPAGATION
    # ================================
    def backward(self, x, y):
        m = x.shape[0]
        activations, zs = self.forward(x)
        grads_w = [np.zeros_like(w) for w in self.weights]
        grads_b = [np.zeros_like(b) for b in self.biases]

        # convertir y a one-hot
        y_onehot = np.zeros((self.layer_sizes[-1], m))
        y_onehot[y, np.arange(m)] = 1

        # delta ultima capa
        delta = activations[-1] - y_onehot
        grads_w[-1] = delta @ activations[-2].T / m
        grads_b[-1] = np.sum(delta, axis=1, keepdims=True) / m

        # delta capas previas
        for l in range(self.num_layers-3, -1, -1):
            delta = (self.weights[l+1].T @ delta) * self.
↪activate_derivative(zs[l])
            grads_w[l] = delta @ activations[l].T / m
            grads_b[l] = np.sum(delta, axis=1, keepdims=True) / m

        # actualizar pesos
        for i in range(self.num_layers-1):
            self.weights[i] -= self.lr * grads_w[i]
            self.biases[i] -= self.lr * grads_b[i]

    # ================================
    # ENTRENAMIENTO
    # ================================
    def train(self, X, y, epochs=5, batch_size=64, X_val=None, y_val=None):
```

```python
        n = X.shape[0]
        for e in range(epochs):
            idx = np.random.permutation(n)
            X, y = X[idx], y[idx]
            for i in range(0, n, batch_size):
                xb = X[i:i+batch_size]
                yb = y[i:i+batch_size]
                self.backward(xb, yb)
            if X_val is not None:
                acc = self.accuracy(X_val, y_val)
                print(f'Epoch {e+1}/{epochs} - Val Accuracy: {acc:.4f}')

    # ================================
    # PREDICCION
    # ================================
    def predict(self, X):
        activations, _ = self.forward(X)
        return np.argmax(activations[-1], axis=0)

    # ================================
    # EXACTITUD
    # ================================
    def accuracy(self, X, y):
        y_pred = self.predict(X)
        return np.mean(y_pred == y)
```

[28]:

```python
# ================================
# CREAR MODELO Y ENTRENAR
# ================================
nn = FeedforwardNN(layer_sizes, activation=activation, lr=learning_rate)
nn.train(X_train, y_train, epochs=epochs, batch_size=batch_size, X_val=X_val,␣
 ↪y_val=y_val)

# evaluar en test
acc_test = nn.accuracy(X_test, y_test)
print(f'Exactitud en test set: {acc_test:.4f}')
```

```
Epoch 1/10 - Val Accuracy: 0.8958
Epoch 2/10 - Val Accuracy: 0.9098
Epoch 3/10 - Val Accuracy: 0.9257
Epoch 4/10 - Val Accuracy: 0.9351
Epoch 5/10 - Val Accuracy: 0.9398
Epoch 6/10 - Val Accuracy: 0.9454
Epoch 7/10 - Val Accuracy: 0.9510
Epoch 8/10 - Val Accuracy: 0.9536
Epoch 9/10 - Val Accuracy: 0.9574
Epoch 10/10 - Val Accuracy: 0.9546
```

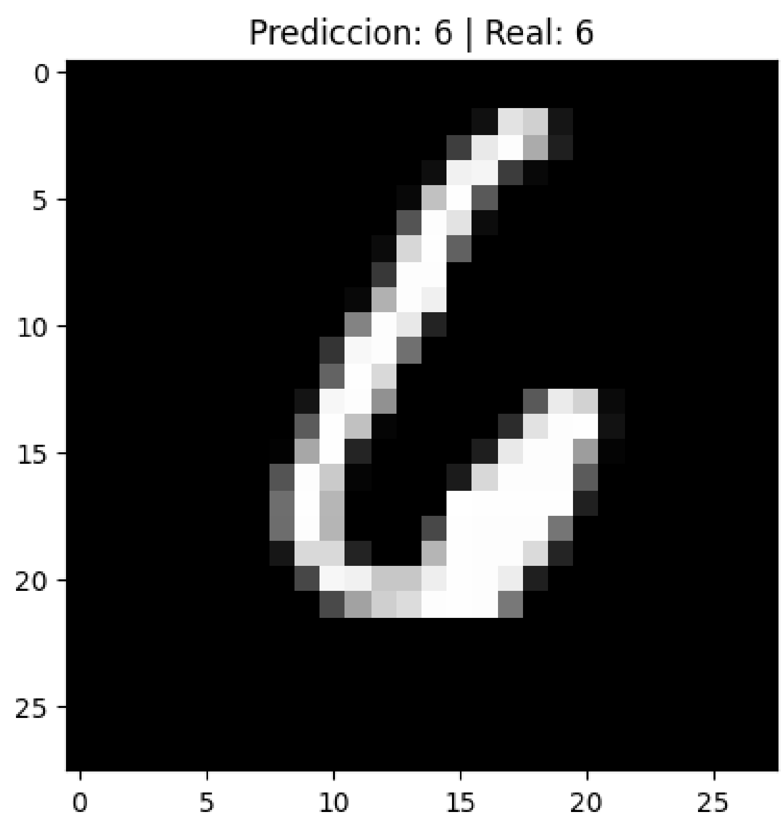
```
Exactitud en test set: 0.9578
```

Aquí dejo un ejemplo de una predicción real sobre el y_test, se puede ver que al hacer variar los hiperparámetros, la predicción del número cambia.

Variar "index" para cambiar de ejemplos

[29]:
```python
# ================================
# PREDICCION DE UN DIGITO ESPECIFICO
# ================================
index = 1990 # cambia este indice para ver otro ejemplo
sample = X_test[index].reshape(1,-1)
predicted_digit = nn.predict(sample)[0]
real_digit = y_test[index]

print(f'Prediccion de la red: {predicted_digit}')
print(f'Digito real: {real_digit}')

# mostrar imagen
plt.imshow(sample.reshape(28,28), cmap='gray')
plt.title(f'Prediccion: {predicted_digit} | Real: {real_digit}')
plt.show()
```

```
Prediccion de la red: 6
Digito real: 6
```



4.3 Parte III — Ítem (3)

Probaré múltiples modelos con diferentes hiperparámetros para ver cual es el mejor modelo con un Cross Validation de 10 divisiones.

[31]:
```
from sklearn.model_selection import KFold
import itertools

# ================================
# Hiperparametros a probar
# ================================
learning_rates = [0.01, 0.05]
activations = ['relu', 'leaky']
hidden_layers_list = [[64], [128], [128,64]]
epochs_list = [5, 10]
batch_size = 64

# ================================
# K-Fold Cross-Validation
# ================================
k_folds = 10 # Cantidad de divisiones para el Cross Validation
kf = KFold(n_splits=k_folds, shuffle=True, random_state=42)

# ================================
# Guardar resultados
# ================================
resultados = []

# ================================
# Combinaciones de hiperparametros
# ================================
for lr, act, hidden, ep in itertools.product(learning_rates, activations,␣
↪hidden_layers_list, epochs_list):
    accs = []
    for train_idx, val_idx in kf.split(X_train):
        # Dividir en train/validation
        X_tr, X_val_cv = X_train[train_idx], X_train[val_idx]
        y_tr, y_val_cv = y_train[train_idx], y_train[val_idx]

        # Crear red nueva
        layer_sizes_cv = [input_size] + hidden + [output_size]
        nn_cv = FeedforwardNN(layer_sizes_cv, activation=act, lr=lr)

        # Entrenar
        nn_cv.train(X_tr, y_tr, epochs=ep, batch_size=batch_size)
```

```python
        # Validar
        acc = nn_cv.accuracy(X_val_cv, y_val_cv)
        accs.append(acc)

    acc_promedio = np.mean(accs)
    resultados.append({
        "lr": lr,
        "activation": act,
        "hidden_layers": hidden,
        "epochs": ep,
        "val_accuracy": acc_promedio
    })
    print(f"lr={lr}, activation={act}, hidden={hidden}, epochs={ep} -> Val␣
↪Accuracy={acc_promedio:.4f}")

# ==============================
# Seleccionar mejor modelo
# ==============================
mejor_modelo = max(resultados, key=lambda x: x['val_accuracy'])
print("\n  Mejor modelo encontrado:")
print(mejor_modelo)
```

```
lr=0.01, activation=relu, hidden=[64], epochs=5 -> Val Accuracy=0.8906
lr=0.01, activation=relu, hidden=[64], epochs=10 -> Val Accuracy=0.9078
lr=0.01, activation=relu, hidden=[128], epochs=5 -> Val Accuracy=0.8928
lr=0.01, activation=relu, hidden=[128], epochs=10 -> Val Accuracy=0.9104
lr=0.01, activation=relu, hidden=[128, 64], epochs=5 -> Val Accuracy=0.3646
lr=0.01, activation=relu, hidden=[128, 64], epochs=10 -> Val Accuracy=0.8398
lr=0.01, activation=leaky, hidden=[64], epochs=5 -> Val Accuracy=0.8898
lr=0.01, activation=leaky, hidden=[64], epochs=10 -> Val Accuracy=0.9089
lr=0.01, activation=leaky, hidden=[128], epochs=5 -> Val Accuracy=0.8931
lr=0.01, activation=leaky, hidden=[128], epochs=10 -> Val Accuracy=0.9105
lr=0.01, activation=leaky, hidden=[128, 64], epochs=5 -> Val Accuracy=0.3798
lr=0.01, activation=leaky, hidden=[128, 64], epochs=10 -> Val Accuracy=0.8412
lr=0.05, activation=relu, hidden=[64], epochs=5 -> Val Accuracy=0.9314
lr=0.05, activation=relu, hidden=[64], epochs=10 -> Val Accuracy=0.9520
lr=0.05, activation=relu, hidden=[128], epochs=5 -> Val Accuracy=0.9292
lr=0.05, activation=relu, hidden=[128], epochs=10 -> Val Accuracy=0.9549
lr=0.05, activation=relu, hidden=[128, 64], epochs=5 -> Val Accuracy=0.9147
lr=0.05, activation=relu, hidden=[128, 64], epochs=10 -> Val Accuracy=0.9369
lr=0.05, activation=leaky, hidden=[64], epochs=5 -> Val Accuracy=0.9321
lr=0.05, activation=leaky, hidden=[64], epochs=10 -> Val Accuracy=0.9497
lr=0.05, activation=leaky, hidden=[128], epochs=5 -> Val Accuracy=0.9322
lr=0.05, activation=leaky, hidden=[128], epochs=10 -> Val Accuracy=0.9547
lr=0.05, activation=leaky, hidden=[128, 64], epochs=5 -> Val Accuracy=0.9194
lr=0.05, activation=leaky, hidden=[128, 64], epochs=10 -> Val Accuracy=0.9568
```

```
 Mejor modelo encontrado:
{'lr': 0.05, 'activation': 'leaky', 'hidden_layers': [128, 64], 'epochs': 10,
'val_accuracy': 0.95678}
```

4.3.1 Resultados - Ítem (3)

Modelo El modelo elegido fue el que obtuvo mayor presición (Var Accuracy de). Esto indica que tiene una posibilidad de acierto grande. Este modelo considera los siguientes hiperparámetros:

- Tasa de aprendizaje (Learning Rate): 0.05
- Función de activación: Leaky ReLU
- Cantidad de capas ocultas: 2
- Cantidad de nodos en cada capa respectivamente: 128, 64
- Número de entrenamientos (Iteraciones de Backpropagation): 10
- Precisión: 0.95678 (95.678%)

Observaciones

Learning rate

- Con lr=0.05 la red aprende mucho mejor que con lr=0.01.
- Un learning rate bajo hace que las redes profundas o con más capas no entrenen bien.

Número de capas y neuronas

- Una sola capa oculta de 64 o 128 neuronas funciona muy bien.
- Agregar una segunda capa ([128,64]) solo mejora si el learning rate es suficientemente alto; con lr bajo estas configuraciones fallan.

Función de activación

- LeakyReLU en general da resultados un poco mejores que ReLU en configuraciones profundas.
- Para capas simples, ambos funcionan casi igual.

Épocas

- Más épocas (10 vs 5) siempre mejora la precisión.
- Para el mejor desempeño conviene entrenar más tiempo, especialmente con lr alto y redes profundas.

Regla general

- No siempre más capas o más neuronas significan mejor resultado; es importante combinar correctamente learning rate, cantidad de capas y neuronas.
- Para este dataset, una o dos capas con tamaño moderado y lr=0.05 dan las mejores accuracies (~0.95–0.96).

Observaciones adicionales

- La función Softmax en la capa de salida garantiza que las predicciones se interpreten como probabilidades, lo cual es ideal para clasificación multi-clase.
- Se observa que el modelo converge más rápido con ReLU que con LeakyReLU para este dataset específico.
- Aumentar demasiado el tamaño de las capas ocultas no siempre mejora la precisión, y puede llevar a sobreajuste (overfitting).
- No realicé más consideraciones de hiperparámetros ya que el tiempo de ejecución aumentaba considerablemente.