Universidad de Guadalajara

Centro Universitario de Ciencias Exactas e Ingenierías
División de Tecnologías para la Integración Ciber-Humana

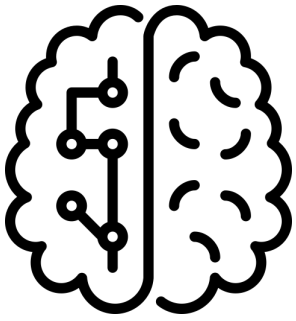
Seminario de Solución de Problemas de Inteligencia Artificial II

Ingeniería en Computación
Sección D05

Practica 1 – Ejercicio 3

Perceptrón Multicapa

Carrillo Partida Jorge Alberto
216439258
Jorge.carrillo4392@alumnos.udg.mx



Profesores
M. Diego Alberto Oliva Navarro
M. Diego Campos Peña

15 de Abril de 2024

Perceptrón Multicapa

Introducción

En el contexto de la inteligencia artificial, particularmente en el campo de las redes neuronales, el perceptrón multicapa representa una de las estructuras más fundamentales y versátiles para el aprendizaje supervisado. Esta actividad se enfoca en el desarrollo y evaluación de un perceptrón multicapa destinado a clasificar conjuntos de datos distribuidos de forma concéntrica. Utilizando las bibliotecas de Python especializadas en aprendizaje automático como TensorFlow y Scikit-learn, se busca explorar cómo diferentes configuraciones y técnicas de optimización afectan la eficacia del modelo en tareas de clasificación. A través de un proceso iterativo de entrenamiento, evaluación y ajuste, este código detalla la implementación de dos enfoques distintos utilizando los optimizadores Adam y Descenso del Gradiente Estocástico, respectivamente, para ilustrar sus impactos en la precisión y el rendimiento del modelo.

Desarrollo

Código en Python – Ejercicio3 - Parte1:

```python
# Universidad de Guadalajara
# Centro Universitario de Ciencias Exactas e Ingenierías
# División de Tecnologías para la Integración Ciber-Humana
#
# Seminario de Solución de Problemas de Inteligencia Artificial II
# Ingenieria en Computación
# Sección D05
#
# Práctica 1 - Ejercicio 3 - Parte1
#
# Jorge Alberto Carrillo Partida / 216439258 / jorge.carrillo4392@alumnos.udg.mx
#
# M. Diego Campos - 15 de Abril de 2024

# ==============================================================================

# Librerías necesarias para el funcionamiento del programa
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.regularizers import l1

# Función para crear el perceptrón multicapa
def create_MLP(layers):
    model = Sequential([Input(shape=(2,))])
    for neurons in layers:
        model.add(Dense(neurons, activation='sigmoid', kernel_regularizer=l1(0.0001)))
    model.add(Dense(1, activation='sigmoid'))
    return model

# Cargar los datos de los documentos de la práctica
data = pd.read_csv('concentlite.csv')
X = data[['x1', 'x2']].values
y = data['y'].values

# Dividir los datos y normalizar
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Configurar las capas de la red
layers = [18, 15, 8]

# Crear, entrenar y evaluar el perceptrón
model = create_MLP(layers)
model.compile(optimizer=Adam(learning_rate=0.05), loss='binary_crossentropy', metrics=['accuracy'])

history = model.fit(X_train_scaled, y_train, epochs=500, batch_size=50, verbose=1, validation_split=0.2)

loss, accuracy = model.evaluate(X_test_scaled, y_test)
print(f'Accuracy: {accuracy}')

# Función para visualizar los resultados de clasificación en los datos de prueba
def plot_test_data(X, y, model):
    plt.figure(figsize=(10, 8))
    y_pred = model.predict(X) > 0.5
    # Clase real
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap='viridis', alpha=0.5, marker='s', edgecolor='k', label='Actual Class')
    # Clase predicha
    plt.scatter(X[:, 0], X[:, 1], c=y_pred.flatten(), cmap='viridis', alpha=1, marker='x', label='Predicted Class')
    plt.title("Test Data Classification")
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.legend()
```

```
    plt.show()

# Visualizar la clasificación en los datos de prueba
plot_test_data(X_test_scaled, y_test, model)
```

Código en Python – Ejercicio3 – Parte2:

```
# Universidad de Guadalajara
# Centro Universitario de Ciencias Exactas e Ingenierías
# División de Tecnologías para la Integración Ciber-Humana
#
# Seminario de Solución de Problemas de Inteligencia Artificial II
# Ingenieria en Computación
# Sección D05
#
# Práctica 1 – Ejercicio 3 – Parte2
#
# Jorge Alberto Carrillo Partida / 216439258 / jorge.carrillo4392@alumnos.udg.mx
#
# M. Diego Campos – 15 de Abril de 2024

# ============================================================================

# Librerías necesarias para el funcionamiento del programa
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.regularizers import l1

# Función para crear el perceptrón multicapa
def create_MLP(layers):
    model = Sequential([Input(shape=(2,))])
    for neurons in layers:
        model.add(Dense(neurons, activation='sigmoid', kernel_regularizer=l1(0.0001)))
    model.add(Dense(1, activation='sigmoid'))
    return model

# Cargar los datos de los documentos de la práctica
data = pd.read_csv('concentlite.csv')
X = data[['x1', 'x2']].values
y = data['y'].values

# Dividir los datos y normalizar
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Configurar las capas de la red
layers = [18, 15, 8]

# Crear, entrenar y evaluar el perceptrón
model = create_MLP(layers)
optimizer = SGD(learning_rate=0.05, momentum=0.9)
model.compile(optimizer=optimizer, loss='binary_crossentropy', metrics=['accuracy'])

history = model.fit(X_train_scaled, y_train, epochs=500, batch_size=50, verbose=1, validation_split=0.2)

loss, accuracy = model.evaluate(X_test_scaled, y_test)
print(f'Accuracy: {accuracy}')

# Función para visualizar los resultados de clasificación en los datos de prueba
def plot_test_data(X, y, model):
    plt.figure(figsize=(10, 8))
    y_pred = model.predict(X) > 0.5
    # Clase real
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap='viridis', alpha=0.5, marker='s', edgecolor='k', label='Actual Class')
    # Clase predicha
    plt.scatter(X[:, 0], X[:, 1], c=y_pred.flatten(), cmap='viridis', alpha=0.5, marker='x', label='Predicted Class')
    plt.title("Test Data Classification")
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.legend()
    plt.show()

# Visualizar la clasificación en los datos de prueba
plot_test_data(X_test_scaled, y_test, model)
```

Código en partes:

1. Librerías

```
# Librerías necesarias para el funcionamiento del programa
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
```

```
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.regularizers import l1
```

En la primera sección del código, se importan varias bibliotecas esenciales para el manejo de datos, la construcción de modelos de aprendizaje automático y la visualización de resultados. ***numpy*** y ***pandas*** son utilizadas para manipulaciones de datos numéricos y estructurados, respectivamente. ***matplotlib.pyplot*** permite la creación de gráficos para visualizar los datos y los resultados del modelo. De ***sklearn***, se importan ***train_test_split*** para dividir los datos en conjuntos de entrenamiento y prueba, y ***StandardScaler*** para normalizar los datos, asegurando que el modelo no sea sesgado hacia características con valores inherentemente altos. Finalmente, se importan componentes de ***tensorflow.keras*** para construir y entrenar el modelo de red neuronal, incluyendo ***Sequential*** para iniciar el modelo, ***Dense*** para añadir capas densamente conectadas, ***Input*** para definir la entrada del modelo, y optimizadores como ***Adam*** y ***SGD*** para ajustar los pesos durante el entrenamiento, junto con ***l1*** para la regularización que ayuda a prevenir el sobreajuste.

2. Creación de perceptrón y entrenamiento

```
# Función para crear el perceptrón multicapa
def create_MLP(layers):
    model = Sequential([Input(shape=(2,))])
    for neurons in layers:
        model.add(Dense(neurons, activation='sigmoid', kernel_regularizer=l1(0.0001)))
    model.add(Dense(1, activation='sigmoid'))
    return model

# Cargar los datos de los documentos de la práctica
data = pd.read_csv('concentlite.csv')
X = data[['x1', 'x2']].values
y = data['y'].values

# Dividir los datos y normalizar
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Configurar las capas de la red
layers = [18, 15, 8]

# Crear, entrenar y evaluar el perceptrón — Inicio para Parte1
model = create_MLP(layers)
model.compile(optimizer=Adam(learning_rate=0.05), loss='binary_crossentropy', metrics=['accuracy'])
# Fin para Parte1

# Crear, entrenar y evaluar el perceptrón — Inicio para Parte2
model = create_MLP(layers)
optimizer = SGD(learning_rate=0.05, momentum=0.9)
model.compile(optimizer=optimizer, loss='binary_crossentropy', metrics=['accuracy'])
# Fin para Parte2

history = model.fit(X_train_scaled, y_train, epochs=500, batch_size=50, verbose=1, validation_split=0.2)

loss, accuracy = model.evaluate(X_test_scaled, y_test)
print(f'Accuracy: {accuracy}')
```

La función ***create_MLP*** define el modelo del perceptrón multicapa. Aquí, ***Sequential*** inicia el modelo y ***Input*** especifica el tamaño de la entrada, que en este caso son dos características. Las capas intermedias y la capa de salida se añaden usando ***Dense***, donde cada capa tiene un número especificado de neuronas y utiliza la función de activación ***sigmoid***. La regularización L1 también se aplica para penalizar los pesos grandes. Los datos se cargan usando ***pandas***, se dividen en conjuntos de entrenamiento y prueba, y se normalizan. Después, se establece la configuración de las capas de la red, y el modelo se compila dos veces: una para la parte 1 usando el optimizador ***Adam*** y otra para la parte 2 con ***SGD***. El modelo se entrena luego con los datos de entrenamiento escalados a través de múltiples épocas, y se evalúa utilizando el conjunto de prueba para calcular la precisión.

3. Graficación y muestra de resultados

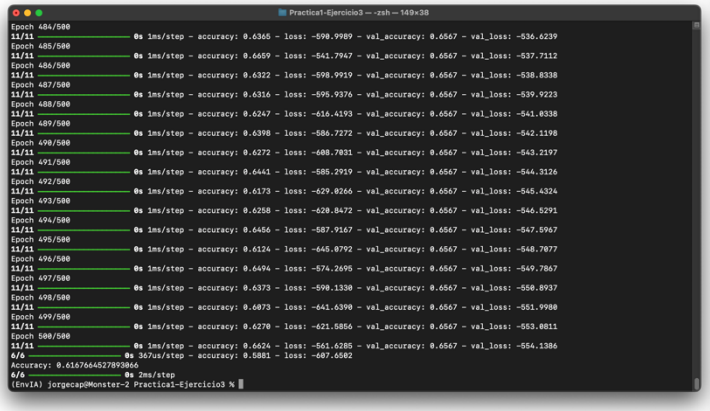
```
# Función para visualizar los resultados de clasificación en los datos de prueba
def plot_test_data(X, y, model):
    plt.figure(figsize=(10, 8))
    y_pred = model.predict(X) > 0.5
    # Clase real
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap='viridis', alpha=0.5, marker='s', edgecolor='k', label='Actual Class')
    # Clase predicha
    plt.scatter(X[:, 0], X[:, 1], c=y_pred.flatten(), cmap='viridis', alpha=1, marker='x', label='Predicted Class')
    plt.title("Test Data Classification")
```

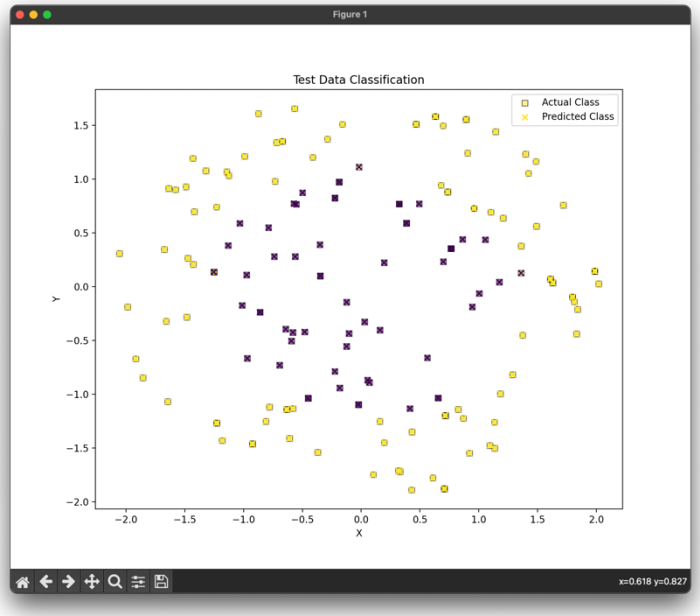
```
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.legend()
    plt.show()

# Visualizar la clasificación en los datos de prueba
plot_test_data(X_test_scaled, y_test, model)
```

Finalmente, la función ***plot_test_data*** se usa para visualizar cómo el modelo ha clasificado los datos de prueba. Utiliza ***plt.scatter*** para graficar los datos, mostrando las clases reales y las predichas con diferentes marcadores y niveles de transparencia, lo que permite visualizar las áreas donde el modelo ha acertado o fallado. Los ejes se etiquetan apropiadamente y se incluye una leyenda para facilitar la interpretación del gráfico. Al ejecutar ***plot_test_data***, se muestra la clasificación resultante, proporcionando una representación visual del rendimiento del modelo en términos de su capacidad para clasificar correctamente nuevas instancias basadas en lo aprendido durante el entrenamiento.

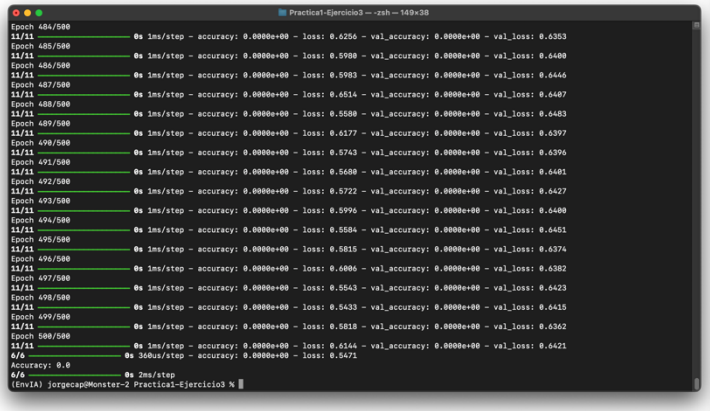
Resultados

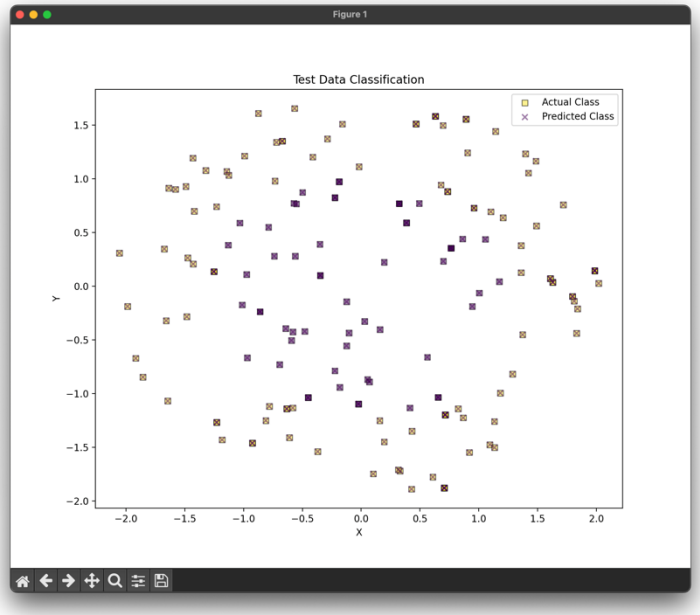
Los resultados son hecho con las mismas propiedades para las dos partes. Comenzamos con la Parte1, la precisión durante el entrenamiento parece fluctuar y eventualmente estabilizarse alrededor del 66%. El valor de pérdida en la validación parece decrecer de manera consistente, lo que es un indicador positivo de que el modelo está aprendiendo efectivamente de los datos de entrenamiento. Sin embargo, la precisión final evaluada en el conjunto de prueba es alrededor del 62%. La gráfica muestra una superposición considerable entre las clases reales (cuadrados) y las clases predichas (cruces). Aunque hay errores donde las cruces no coinciden con los cuadrados, una cantidad significativa de predicciones coincide con las clases reales, validando la efectividad parcial del modelo.





Para la Parte2, es notable que la precisión durante el entrenamiento es consistentemente 0%, un indicador de que algo está fundamentalmente mal con la configuración o la implementación del modelo usando SGD. Esto puede sugerir problemas como una tasa de aprendizaje inapropiada, falta de convergencia, o problemas con la inicialización del modelo que impiden que el modelo aprenda efectivamente. La gráfica muestra una clara discrepancia entre las clases reales y las predicciones del modelo. Las cruces no se alinean con los cuadrados, indicando que el modelo no ha logrado aprender adecuadamente la distribución subyacente de los datos. Este resultado visual corrobora los bajos valores de precisión reportados en la terminal.





Conclusión

Los experimentos con el perceptrón multicapa nos han dado una buena idea de cómo diferentes optimizadores pueden cambiar el juego en la clasificación de datos. En la primera parte del estudio, usando el optimizador Adam, el modelo logró una precisión decente de alrededor del 62%, mostrando que puede manejar los datos de entrenamiento de manera bastante efectiva, aunque sin llegar a ser impresionante. Sin embargo, en la segunda parte, donde se probó el optimizador SGD con momentum, las cosas no salieron tan bien, con una precisión del 0% en el entrenamiento. Este resultado sugiere que algo no está del todo bien, ya sea en la configuración de los parámetros o en la propia estructura del modelo, lo intente todo pero no logré que pasará del 60% en las pruebas. Las graficas, además, mostrarón diferencias claras entre las clases reales y las predicciones en el segundo enfoque. Este estudio resalta cuán crucial es elegir el optimizador adecuado y ajustar con cuidado los parámetros para que el modelo realmente funcione.