Tecnológico de Monterrey

Instituto Tecnológico y de Estudios Superiores de Monterrey

Momento de Retroalimentación: Módulo 2 Análisis y Reporte sobre el desempeño del modelo

TC3006C.101 Inteligencia artificial avanzada para la ciencia de datos I

Profesores:

Ivan Mauricio Amaya Contreras

Blanca Rosa Ruiz Hernandez

Antonio Carlos Bento

Frumencio Olivas Alvarez

Hugo Terashima Marín

Alumno:

Alberto H Orozco Ramos - A00831719

5 de Agosto de 2023.

Momento de Retroalimentación: Módulo 2 Análisis y Reporte sobre el desempeño del modelo. (Portafolio Análisis)

Instrucciones

Entregable: Análisis y Reporte sobre el desempeño del modelo.

1. Escoge una de las 2 implementaciones que tengas y genera un análisis sobre su desempeño en un set de datos. Este análisis lo deberás documentar en un reporte con indicadores claros y gráficas comparativas que respalden tu análisis.
2. El análisis debe de contener los siguientes elementos:
3. Separación y evaluación del modelo con un conjunto de prueba y un conjunto de validación (Train/Test/Validation).
4. Diagnóstico y explicación el grado de bias o sesgo: bajo medio alto
5. Diagnóstico y explicación el grado de varianza: bajo medio alto
6. Diagnóstico y explicación el nivel de ajuste del modelo: underfitt fitt overfitt
7. Basándote en lo encontrado en tu análisis utiliza técnicas de regularización o ajuste de parámetros para mejorar el desempeño de tu modelo y documenta en tu reporte cómo mejoró este.

Análisis y Reporte sobre el Desempeño del Modelo

A lo largo de este reporte estaré analizando el rendimiento de mi algoritmo de redes neuronales que desarrollé para la entrega Momento de Retroalimentación. Módulo 2. Uso de framework o biblioteca de aprendizaje máquina para la implementación de una solución, en la cual hizo uso de una base de datos de películas de Marvel, en resumidas cuentas, contiene información acerca de las recaudaciones, presupuesto, críticas, porcentaje de aceptación, entre otras características relacionadas que sirven para determinar cuál es la película más exitosa hecha por Marvel.

Este entregable será muy parecido al anterior, sin embargo se ahondará en los siguientes puntos:

- Separación y evaluación del modelo con un conjunto de prueba y un conjunto de validación (Train/Test/Validation).
- Diagnóstico y explicación el grado de bias o sesgo: bajo medio alto
- Diagnóstico y explicación el grado de varianza: bajo medio alto
- Diagnóstico y explicación el nivel de ajuste del modelo: underfitt fitt overfitt

Antes de empezar

Lo primero que debemos hacer es cargar los datos (listas) y librerías, esto incluye el framework con el que se trabajará, en este caso es SciKitLearn. Para ello, debemos:

Ejecutar esta celda solo si estas utilizando Google Colab. Se deberá subir el archivo CSV a nuestro Drive y montarlo dentro de Colab:

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.
mount("/content/drive", force_remount=True).
```

Importar bibliotecas

Para este entregable ahora podemos implementar más de las librerías que se encuentran relacionadas con implementación de modelos de Machine Learning, en este caso usaremos SKLearn para implementar redes neuronales:

In [ ]:
```
# Importamos tensorflow como herramienta de apoyo para la implementación de keras y
import tensorflow as tf
# Importamos Keras
from tensorflow import keras
# Importamos GridSearch para aplicarlo al refinamiento del modelo
from sklearn.model_selection import GridSearchCV
# Importamos la funcion train_test_split() para dividir los datos de entrada
from sklearn.model_selection import train_test_split
# Importamos la función de StandardScaler para estandarizar los datos de entrada
from sklearn.preprocessing import StandardScaler
# Importamos las funciones de mean_squared_error y mean_absolute_error para evaluar
from sklearn.metrics import mean_squared_error, mean_absolute_error
# Importamos KerasRegressor para realizar un GridSearch
from tensorflow.keras.wrappers.scikit_learn import KerasRegressor

# Importamos librerías adicionales para el tratamiento y proyección de los datos
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from prettytable import PrettyTable
```

Colocamos la ruta dónde se encuentra el archivo CSV y cargamos los datos:

In [ ]:
```
# Editar el path de ser necesario
data = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/Machine Learning/Marvel_
```

Extraemos los datos de interés y los dividimos por segmentos

Empezando por la separación del dataset, esto lo podemos ver reflejado desde la implementación del código. Este proceso se lleva a cabo con la implementación de las siguientes herramientas:

In [ ]:
```
# Extraemos las columnas con datos para generar una predicción:
# Lista con todos los valores de la variable independiente
X = data[['opening weekend ($m)']][:len(data)//2]]
# Lista con todos los valores de la variable dependiente
y = data[['second weekend ($m)']][:len(data)//2]]

# Dividimos los datos en sets de entrenamiento, validación y pruebas
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.2, random_sta
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, rand


# Estandarizamos los datos de pruebas y validación
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)
X_test_scaled = scaler.transform(X_test)
```

Básicamente train_test_split es una función encargada de separa datos de forma aleatoria, con el fin de obtener 2 o más subsets de un mismo conjunto de datos. Después de extraer los datos del archivo “Marvel_Movies.csv”:

Training Set

El set de entrenamiento este compuesto de las variables X_train y y_train. Estas dos son utilizadas dentro de la sección de entrenamiento, que sirve para que el modelo puedo probar algunos valores iniciales, entienda un poco del contexto de la situación y en base a ello pueda empezar a generar algunas cuántas predicciones que, con cada iteración, reduzca el valor del error lo más posible y esto le permita posteriormente proporcionar resultados coherentes y predicciones certeras.

Test / Validation Set

El set de pruebas o validación este compuesto de las variables X_test y y_test. Estas dos son utilizadas dentro de la sección de predicciones, la cual consiste en que, en base al entrenamiento realizado previamente, el modelo pueda empezar a empezar a recibir valores de entrada (X_test) sin recibir los valores de salida (y_test), generando predicciones por si mismo, claro que se espera que teniendo un buen entrenamiento y un MSE pequeño, el rango de error de las pruebas sea de igual forma pequeño y muy preciso en comparación con los datos reales (y_test). Para la validación, no es totalmente necesario declarar variables

adicionales para comparar los resultados del modelo con los valores reales, sin embargo esto se puede hacer con los valores de X_val y y_val, son exactamente los mismos valores que X_test y y_test y no son modificados en ningún momento, solo serán utilizados como herramienta de comparación en contraste de los datos generados por las estimaciones del modelo.

Construimos la Red Neuronal

El modelo se conforma de 4 capas, la capa de entrada conformada de 25 neuronas, 50 para la segunda, 70 para la tercera y una última de la capa de salida:

In [ ]:
```
model = keras.Sequential([
    keras.layers.Dense(25, activation='tanh', input_shape=(1,)), # Capa de entrada
    keras.layers.Dense(50, activation='tanh'), # Segunda Capa con 50 neuronas y una
    keras.layers.Dense(70, activation='relu'), # Tercer capa con 70 neuronas y una
    keras.layers.Dense(1) # Capa de salida con una neurona
])

# Compilamos el modelo
model.compile(optimizer='adam', loss='mean_squared_error')
```

Entrenamos el modelo

Entrenamos la red neuronal con el set de entrenamiento de datos. Después de probar distintos tipos de variables, encontré que por lo menos el batch_size tiene que ser pequeño y 100 iteraciones son necesarias para obtener un buen resultado (evidentemente con la configuración previamente determinado):

In [ ]:
```
history = model.fit(X_train_scaled, y_train, epochs=100, batch_size=4, validation_d
```

```
Epoch 1/100
6/6 [==============================] - 1s 32ms/step - loss: 4709.5610 - val_loss: 99
7.9565
Epoch 2/100
6/6 [==============================] - 0s 7ms/step - loss: 4672.8589 - val_loss: 98
2.0233
Epoch 3/100
6/6 [==============================] - 0s 7ms/step - loss: 4627.9902 - val_loss: 96
6.7639
Epoch 4/100
6/6 [==============================] - 0s 7ms/step - loss: 4587.2339 - val_loss: 95
0.4958
Epoch 5/100
6/6 [==============================] - 0s 11ms/step - loss: 4544.9438 - val_loss: 93
0.5075
Epoch 6/100
6/6 [==============================] - 0s 7ms/step - loss: 4492.7070 - val_loss: 90
8.1805
Epoch 7/100
6/6 [==============================] - 0s 7ms/step - loss: 4429.8442 - val_loss: 88
0.4424
Epoch 8/100
6/6 [==============================] - 0s 7ms/step - loss: 4359.3511 - val_loss: 84
8.3078
Epoch 9/100
6/6 [==============================] - 0s 8ms/step - loss: 4274.8701 - val_loss: 81
2.4136
Epoch 10/100
6/6 [==============================] - 0s 8ms/step - loss: 4169.3687 - val_loss: 76
8.5904
Epoch 11/100
6/6 [==============================] - 0s 7ms/step - loss: 4054.1750 - val_loss: 72
1.7755
Epoch 12/100
6/6 [==============================] - 0s 7ms/step - loss: 3922.3127 - val_loss: 67
3.1280
Epoch 13/100
6/6 [==============================] - 0s 7ms/step - loss: 3765.8875 - val_loss: 62
0.4235
Epoch 14/100
6/6 [==============================] - 0s 7ms/step - loss: 3595.0579 - val_loss: 56
7.5769
Epoch 15/100
6/6 [==============================] - 0s 8ms/step - loss: 3413.3152 - val_loss: 51
6.9240
Epoch 16/100
6/6 [==============================] - 0s 7ms/step - loss: 3221.3538 - val_loss: 47
2.6277
Epoch 17/100
6/6 [==============================] - 0s 7ms/step - loss: 3012.9062 - val_loss: 43
5.4329
Epoch 18/100
6/6 [==============================] - 0s 7ms/step - loss: 2815.4353 - val_loss: 40
7.0928
Epoch 19/100
6/6 [==============================] - 0s 7ms/step - loss: 2604.7441 - val_loss: 39
```

```
0.5373
Epoch 20/100
6/6 [==============================] - 0s 7ms/step - loss: 2407.6875 - val_loss: 38
2.4320
Epoch 21/100
6/6 [==============================] - 0s 7ms/step - loss: 2214.5076 - val_loss: 38
8.2175
Epoch 22/100
6/6 [==============================] - 0s 7ms/step - loss: 2024.6733 - val_loss: 39
8.7867
Epoch 23/100
6/6 [==============================] - 0s 7ms/step - loss: 1841.8584 - val_loss: 41
1.0446
Epoch 24/100
6/6 [==============================] - 0s 7ms/step - loss: 1684.5704 - val_loss: 42
5.6158
Epoch 25/100
6/6 [==============================] - 0s 7ms/step - loss: 1523.6168 - val_loss: 44
3.8269
Epoch 26/100
6/6 [==============================] - 0s 7ms/step - loss: 1395.8358 - val_loss: 44
9.5019
Epoch 27/100
6/6 [==============================] - 0s 7ms/step - loss: 1265.1337 - val_loss: 45
3.6588
Epoch 28/100
6/6 [==============================] - 0s 8ms/step - loss: 1138.6207 - val_loss: 44
8.4029
Epoch 29/100
6/6 [==============================] - 0s 8ms/step - loss: 1048.2828 - val_loss: 44
0.2676
Epoch 30/100
6/6 [==============================] - 0s 7ms/step - loss: 944.1674 - val_loss: 430.
6722
Epoch 31/100
6/6 [==============================] - 0s 7ms/step - loss: 851.7125 - val_loss: 420.
3643
Epoch 32/100
6/6 [==============================] - 0s 8ms/step - loss: 789.4615 - val_loss: 409.
7734
Epoch 33/100
6/6 [==============================] - 0s 7ms/step - loss: 715.7034 - val_loss: 395.
6070
Epoch 34/100
6/6 [==============================] - 0s 7ms/step - loss: 664.2027 - val_loss: 381.
8739
Epoch 35/100
6/6 [==============================] - 0s 7ms/step - loss: 613.0632 - val_loss: 375.
5060
Epoch 36/100
6/6 [==============================] - 0s 8ms/step - loss: 579.1811 - val_loss: 363.
9500
Epoch 37/100
6/6 [==============================] - 0s 10ms/step - loss: 544.3904 - val_loss: 35
3.7731
Epoch 38/100
```

```
6/6 [==============================] - 0s 7ms/step - loss: 513.3525 - val_loss: 342.
7154
Epoch 39/100
6/6 [==============================] - 0s 7ms/step - loss: 493.3540 - val_loss: 338.
1211
Epoch 40/100
6/6 [==============================] - 0s 7ms/step - loss: 478.7944 - val_loss: 327.
0826
Epoch 41/100
6/6 [==============================] - 0s 9ms/step - loss: 465.0573 - val_loss: 321.
0597
Epoch 42/100
6/6 [==============================] - 0s 7ms/step - loss: 448.3573 - val_loss: 310.
3001
Epoch 43/100
6/6 [==============================] - 0s 7ms/step - loss: 441.2306 - val_loss: 301.
8479
Epoch 44/100
6/6 [==============================] - 0s 8ms/step - loss: 434.2829 - val_loss: 297.
1542
Epoch 45/100
6/6 [==============================] - 0s 7ms/step - loss: 428.2415 - val_loss: 297.
3064
Epoch 46/100
6/6 [==============================] - 0s 7ms/step - loss: 424.1071 - val_loss: 288.
1541
Epoch 47/100
6/6 [==============================] - 0s 8ms/step - loss: 419.1018 - val_loss: 277.
9388
Epoch 48/100
6/6 [==============================] - 0s 7ms/step - loss: 415.0001 - val_loss: 273.
5025
Epoch 49/100
6/6 [==============================] - 0s 9ms/step - loss: 413.2065 - val_loss: 274.
1231
Epoch 50/100
6/6 [==============================] - 0s 7ms/step - loss: 410.2151 - val_loss: 268.
6681
Epoch 51/100
6/6 [==============================] - 0s 7ms/step - loss: 406.9783 - val_loss: 261.
2529
Epoch 52/100
6/6 [==============================] - 0s 7ms/step - loss: 405.3244 - val_loss: 255.
2744
Epoch 53/100
6/6 [==============================] - 0s 8ms/step - loss: 401.3757 - val_loss: 249.
1263
Epoch 54/100
6/6 [==============================] - 0s 7ms/step - loss: 401.5996 - val_loss: 240.
5543
Epoch 55/100
6/6 [==============================] - 0s 7ms/step - loss: 398.7896 - val_loss: 238.
8915
Epoch 56/100
6/6 [==============================] - 0s 8ms/step - loss: 396.8849 - val_loss: 237.
3648
```

```
Epoch 57/100
6/6 [==============================] - 0s 7ms/step - loss: 396.2077 - val_loss: 237.
9890
Epoch 58/100
6/6 [==============================] - 0s 8ms/step - loss: 393.3666 - val_loss: 227.
8648
Epoch 59/100
6/6 [==============================] - 0s 7ms/step - loss: 391.4272 - val_loss: 222.
7317
Epoch 60/100
6/6 [==============================] - 0s 7ms/step - loss: 391.0409 - val_loss: 221.
2667
Epoch 61/100
6/6 [==============================] - 0s 8ms/step - loss: 387.9845 - val_loss: 215.
8253
Epoch 62/100
6/6 [==============================] - 0s 8ms/step - loss: 386.7026 - val_loss: 211.
6145
Epoch 63/100
6/6 [==============================] - 0s 7ms/step - loss: 384.7795 - val_loss: 209.
5624
Epoch 64/100
6/6 [==============================] - 0s 7ms/step - loss: 385.1240 - val_loss: 197.
6944
Epoch 65/100
6/6 [==============================] - 0s 7ms/step - loss: 384.4830 - val_loss: 198.
9500
Epoch 66/100
6/6 [==============================] - 0s 7ms/step - loss: 378.5881 - val_loss: 197.
8549
Epoch 67/100
6/6 [==============================] - 0s 9ms/step - loss: 378.0811 - val_loss: 196.
3516
Epoch 68/100
6/6 [==============================] - 0s 7ms/step - loss: 375.1862 - val_loss: 189.
2943
Epoch 69/100
6/6 [==============================] - 0s 7ms/step - loss: 372.8882 - val_loss: 182.
5033
Epoch 70/100
6/6 [==============================] - 0s 8ms/step - loss: 374.2914 - val_loss: 176.
3186
Epoch 71/100
6/6 [==============================] - 0s 9ms/step - loss: 371.0659 - val_loss: 173.
5145
Epoch 72/100
6/6 [==============================] - 0s 7ms/step - loss: 370.0969 - val_loss: 170.
4386
Epoch 73/100
6/6 [==============================] - 0s 7ms/step - loss: 365.1167 - val_loss: 167.
5234
Epoch 74/100
6/6 [==============================] - 0s 8ms/step - loss: 366.0713 - val_loss: 155.
0600
Epoch 75/100
6/6 [==============================] - 0s 7ms/step - loss: 363.5367 - val_loss: 156.
```

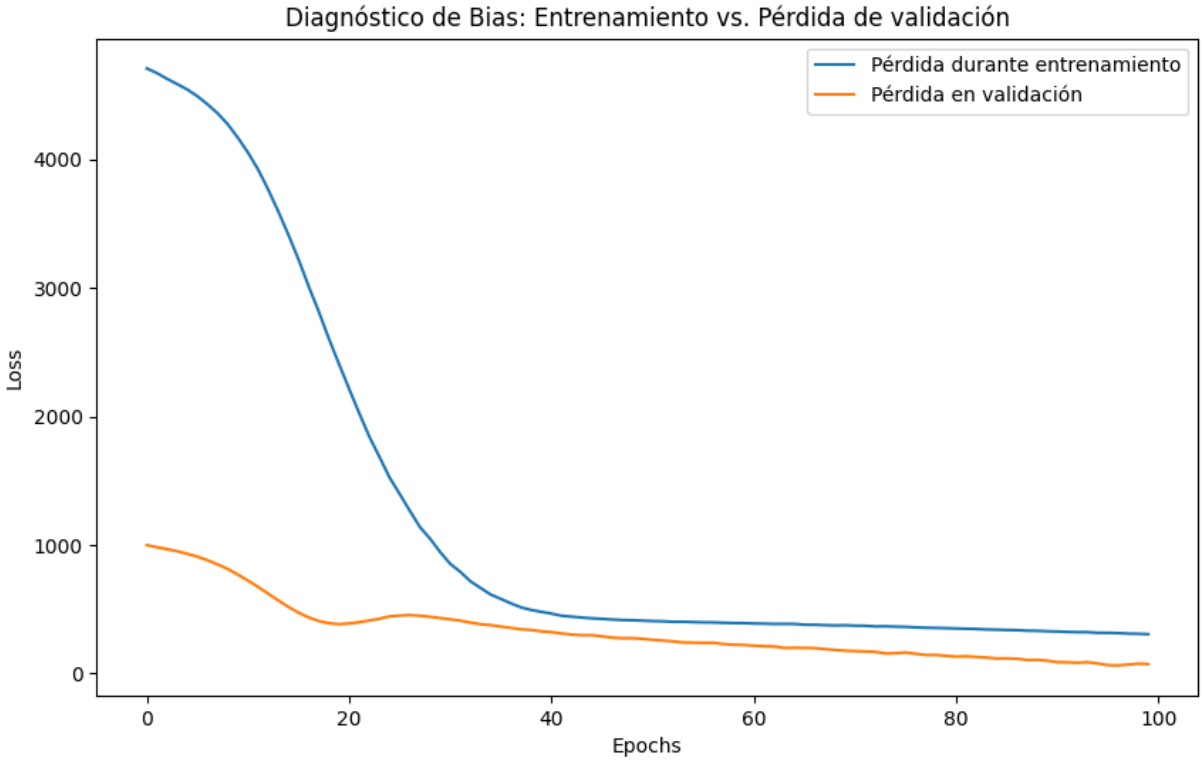
```
4772
Epoch 76/100
6/6 [==============================] - 0s 8ms/step - loss: 361.7549 - val_loss: 162.
2110
Epoch 77/100
6/6 [==============================] - 0s 8ms/step - loss: 357.8917 - val_loss: 152.
5766
Epoch 78/100
6/6 [==============================] - 0s 7ms/step - loss: 354.9098 - val_loss: 142.
9638
Epoch 79/100
6/6 [==============================] - 0s 7ms/step - loss: 353.3069 - val_loss: 143.
1161
Epoch 80/100
6/6 [==============================] - 0s 8ms/step - loss: 351.6259 - val_loss: 136.
5910
Epoch 81/100
6/6 [==============================] - 0s 8ms/step - loss: 349.1891 - val_loss: 130.
1989
Epoch 82/100
6/6 [==============================] - 0s 7ms/step - loss: 347.3023 - val_loss: 133.
1166
Epoch 83/100
6/6 [==============================] - 0s 7ms/step - loss: 345.1865 - val_loss: 127.
6737
Epoch 84/100
6/6 [==============================] - 0s 9ms/step - loss: 341.5679 - val_loss: 122.
8360
Epoch 85/100
6/6 [==============================] - 0s 12ms/step - loss: 339.6742 - val_loss: 11
5.8777
Epoch 86/100
6/6 [==============================] - 0s 10ms/step - loss: 337.2421 - val_loss: 11
6.6149
Epoch 87/100
6/6 [==============================] - 0s 10ms/step - loss: 336.0146 - val_loss: 11
3.5773
Epoch 88/100
6/6 [==============================] - 0s 13ms/step - loss: 332.1215 - val_loss: 10
4.0628
Epoch 89/100
6/6 [==============================] - 0s 15ms/step - loss: 330.9419 - val_loss: 10
4.7791
Epoch 90/100
6/6 [==============================] - 0s 12ms/step - loss: 327.7764 - val_loss: 98.
8867
Epoch 91/100
6/6 [==============================] - 0s 14ms/step - loss: 325.7419 - val_loss: 87.
5102
Epoch 92/100
6/6 [==============================] - 0s 11ms/step - loss: 322.8111 - val_loss: 86.
1919
Epoch 93/100
6/6 [==============================] - 0s 10ms/step - loss: 320.5023 - val_loss: 82.
5093
Epoch 94/100
```

```
6/6 [==============================] - 0s 17ms/step - loss: 320.8429 - val_loss: 87.
3757
Epoch 95/100
6/6 [==============================] - 0s 11ms/step - loss: 315.3235 - val_loss: 76.
4105
Epoch 96/100
6/6 [==============================] - 0s 16ms/step - loss: 315.1294 - val_loss: 63.
9175
Epoch 97/100
6/6 [==============================] - 0s 12ms/step - loss: 313.3167 - val_loss: 61.
7105
Epoch 98/100
6/6 [==============================] - 0s 9ms/step - loss: 309.3064 - val_loss: 67.5
593
Epoch 99/100
6/6 [==============================] - 0s 9ms/step - loss: 307.7658 - val_loss: 75.0
709
Epoch 100/100
6/6 [==============================] - 0s 9ms/step - loss: 304.8067 - val_loss: 71.8
623
```

Análisis del Grado del Bias o Sesgo del entrenamiento

Para poder graficar el grado de sesgo del modelo durante el proceso de entrenamiento, tuvimos que crear una variable "history" para registrar dicho entrenamiento para posteriormente graficar esta variabilidad:

In [ ]:
```
plt.figure(figsize=(10, 6))
# Obtenemos la pérdida durante el entrenamiento
plt.plot(history.history['loss'], label='Pérdida durante entrenamiento')
# Obtenemos la pérdida de validación durante el entrenamiento
plt.plot(history.history['val_loss'], label='Pérdida en validación')
# Iteraciones en el eje x
plt.xlabel('Epochs')
# Pérdida en el eje y
plt.ylabel('Loss')
# Título de la gráfica
plt.title('Diagnóstico de Bias: Entrenamiento vs. Pérdida de validación')
plt.legend()
# Desplegamos la gráfica
plt.show()
```



Como podemos observar, inicialmente la pérdida durante entrenamiento incia con un valor muy alto, en este caso con una pérdida de más de 4000, que con el pasar de las iteraciones baja drásticamente entre la primer iteración hasta la repetición 40 o 50, que es dónde se empieza a estabilizar el modelo y deja de presentar pérdidas significativas, podríamos decir que se decreciendo pero muy ligeramente desde el valor 500 hasta el 400 aproximadamente hasta acercarse a la línea que representa la pérdida de validación.

En cuánto a este otro factor, la pérdida de validación empieza con un valor alto de 1000, pero a diferencia de la pérdida de entrenamiento, inmediatamente empieza a decrecer hasta la iteración 18, luego tiene un pequeño crecimiento entre la repetción 19 y 21, para luego seguir decreciendo ligeramente hasta acercarse bastante con los datos de pérdida de entrenamiento, claro que la pérdida de validación siempre se encuentra debajo de la péridda de entrenamiento.

Algunas de las conclusiones que podemos destacar son que tenemos un claro ejemplo de convergencia de pérdida de entrenamiento, esto quiere decir que como la périda de entrenamiento se encuentra disminuyendo tan drásticamente con cada iteración que pasa hasta que logra estabilizarse en el valor 500 aproximadamente, sugiere que el modelo si está aprendiendo efecivamente en base a los datos proporcionados de entrenamiento (X_train, y_test). Es así que, esta reducción indica que el modelo está ajustando los pesos de cada neurona y sus respectivos sesgos con el fin de minimizar el error de entrenamiento.

También podemos determinar un patrín de pérdida de validación, el cual se puede interpretar dese la disminución inicial con respecto a la pérdida de validación (alrededor de las iteraciones 18 o 19), esto indica que el modelo está mejorando inicialmente su capacidad

para generalizar a datos invisibles. Aún así, el ligero aumento que se encuentra en la iteración 20 o 21 y el posterior decremento en el resto de las iteraciones, sugiere que el rendimiento de generalizaciób del modelo ha alcanzado una meseta.

Por ende, tenemos por seguro que:

1. Es probable que el modelo haya aprendido a ajustarse bien a los datos de entrenamiento, como lo demuestra la reducción significativa de la pérdida de entrenamiento.
2. La mejora inicial en la pérdida de validación sugiere que el modelo es capaz de generalizarse, pero alcanza un nivel relativamente temprano de rendimiento.
3. El desempeño del modelo es razonable, aún así creo firmemente que este se puede mejorar auque puede resultar desafiante ya que ambas pérdidas se llegan a estabilizar bastante bien. Se podría considerar experimentar con diferentes metodologías o arquitecturas para lograr una mejora en el modelo.

Análisis del grado de varianza

Inicialmente, el modelo generaliza razonablemente bien, como se indica en la disminución de la pérdida de validación. Sin embargo, el rendimiento de generalización del modelo eventualmente se estabiliza, lo que sugiere que puede haber alcanzado su límite en la reducción de la varianza. La brecha entre la pérdida de entrenamiento y la pérdida de validación indica que aún hay margen para mejorar al reducir la diferencia entre ambas pérdidas. Para abordar esta varianza moderada, puedes considerar las siguientes acciones:

Experimentar con técnicas de regularización como dropout o regularización L2 para reducir el sobreajuste y acercar aún más las curvas de pérdida de entrenamiento y validación. Recopilar datos más diversos y representativos para posiblemente mejorar la capacidad de generalización del modelo. Probar diferentes arquitecturas de modelos o hiperparámetros para encontrar un mejor equilibrio entre el sesgo y la varianza.

Evaluamos el rendimiento del algoritmo durante el entrenamiento

El siguiente código se encarga de desplegar la métrica de Mean Squared Error (MSE) con respecto al rendimiento del modelo utilizando el set de datos de entrenamiento. En cuanto a los resultados, para considerar que el modelo es óptimo y ofrece buenas estimaciones, el valor de MSE debe ser lo más pequeño posible, de ser así, esto quiere decir que las predicciones se encuentran muy cerca de los valores reales:

In [ ]:
```
# Evaluamos el modelo con los datos de entrenamiento
loss = model.evaluate(X_test_scaled, y_test)
```

```
# Desplegamos el Mean Squared Error (MSE)
print("Mean Squared Error:", loss)
```

```
1/1 [==============================] - 0s 22ms/step - loss: 159.4192
Mean Squared Error: 159.41923522949222
```

Realizamos predicciones

Utilizamos el modelo entrenado para realizar las predicciones con respecto a los datos de prueba (X_test_scaled):

In [ ]:
```
second_wknd_pred = model.predict(X_test_scaled)
print("Predictions:", second_wknd_pred)
```

```
1/1 [==============================] - 0s 100ms/step
Predictions: [[34.584602]
 [40.747124]
 [90.82245 ]]
```

Evaluamos el modelo

Comprobamos el rendimiento de la red neuronal con el set de entrenamiento, en este caso utilizamos el MSE y MAE para medir el nivel de error que obtuvo el modelo de los valores estimados con respecto a los valores reales:

In [ ]:
```
# Calculamos las métricas de regresión
mse = mean_squared_error(y_test, second_wknd_pred)
rmse = np.sqrt(mse)
mae = mean_absolute_error(y_test, second_wknd_pred)

print("Mean Squared Error:", mse)
print("Root Mean Squared Error:", rmse)
print("Mean Absolute Error:", mae)
```

```
Mean Squared Error: 159.41923083757524
Root Mean Squared Error: 12.626132853632392
Mean Absolute Error: 10.530240885416669
```

Gráfica de Comparación

Graficamos los datos de "Opening Weekend (MUSD)" y "Second Weekend (MUSD)" originales que son parte del conjunto de pruebas (que es utilizado para generar las predicciones del modelo) con respecto a los datos originales de "Opening Weekend (MUSD)" con las predicciones de "Second Weekend (MUSD)":

Observaciones sobre la varianza

Por lo visto en la gráfica y dentro de la tabla comparativa siguiente, podemos observar que al menos en el caso de este set de datos que otorgó la función train_test_split, estamos ante

un posible caso de overfitting. Esto lo podemos asumir basándonos en el comportamiento que tienen los valores extimados con respecto a los valores reales, ya que solamente uno de estos realmente se encuentra cerca de un valor real, los demás se encuentran bastante alejados de los mismos, lo que vuelve al modelo un poco dudoso en cuanto a su precisión y su moderado nivel de varianza. Claro que sabemos que como los datos que otorga para cada set, la función train_test_split realiza un procedimiento aleatorio para ello y los resultados pueden cambiar, pero esto realmente no produce un cambio significativo en los resultados del modelo en sí mismo, siguen presentando un poco de error, aunque en realidad no es demasiado como lo podría ser con un modelo con hiperparámetros malos o poco eficientes para su caso y contexto.

In [ ]:
```
# Graficamos los datos originales
plt.scatter(X_test, y_test, label='Actual Data')

# Graficamos los datos estimados por la red neuronal
plt.scatter(X_test, second_wknd_pred, color='red', label='Predicted Data')

plt.xlabel('Opening Weekend ($MUSD)')
plt.ylabel('Second Weekend ($MUSD)')
plt.title('Actual vs. Predicciones de Second Weekend Revenue')
plt.legend()
plt.show()
```

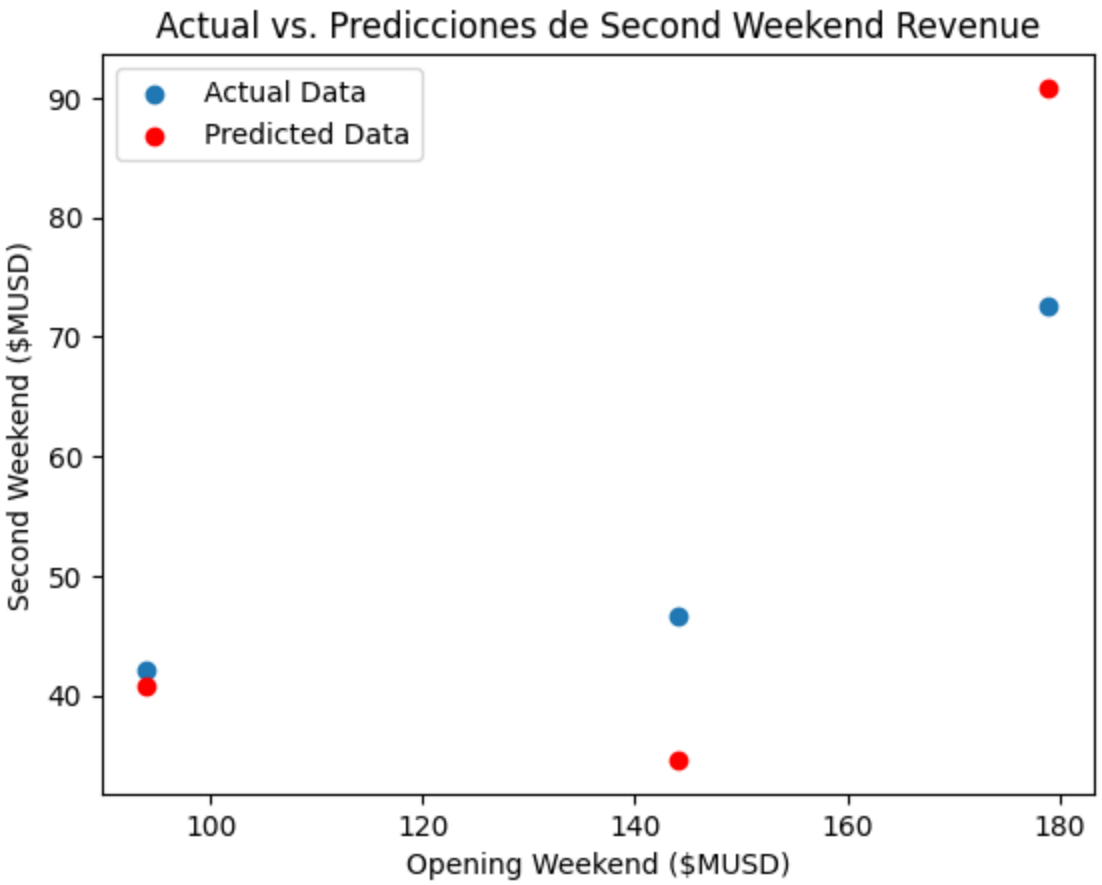


Tabla Comparativa de Visualización de Datos

Dentro de esta tabla se comparan los datos que se utilizaron para entrenamiento, en específico podemos encontrar los valores de la recaudación de apertura contra los valores de recaudación de la segunda semana junto a sus respectivas predicciones elaboradas por el algoritmo de redes neuronales:

In [ ]:
```
# Convertimos second_wknd_pred ndarray a una serie de pandas
second_wknd_pred_series = pd.Series(second_wknd_pred.flatten())

# Creamos la tabla comparativa
table = PrettyTable()
table.field_names = ["opening weekend ($m)", "second weekend ($m)", "Predicted 2nd

# Iteramos por cada una de las filas y añadimos los datos a la tabla
for x, y_true, y_pred in zip(X_test.values, y_test.values, second_wknd_pred_series)
    table.add_row([x[0], y_true[0], y_pred])

# Mostramos la tabla
print(table)
```

```
+----------------------+--------------------+-------------------------------+
| opening weekend ($m) | second weekend ($m) | Predicted 2nd Weekend ($MUSD) |
+----------------------+--------------------+-------------------------------+
|        144.0         |         46.6       |       34.58460235595703       |
|         94.0         |         42.1       |       40.74712371826172       |
|        179.0         |         72.6       |       90.82244873046875       |
+----------------------+--------------------+-------------------------------+
```

Definimos una función para el refinamiento del modelo

In [ ]:
```
def create_model(neurons_layer1, neurons_layer2, neurons_layer3):
    model = keras.Sequential([
        keras.layers.Dense(neurons_layer1, activation='tanh', input_shape=(1,)),
        keras.layers.Dense(neurons_layer2, activation='tanh'),
        keras.layers.Dense(neurons_layer3, activation='relu'),
        keras.layers.Dense(1)
    ])
    model.compile(optimizer='adam', loss='mean_squared_error')
    return model
```

Creamos un regresor de keras en conjunto con SciKitLearn

In [ ]:
```
model = KerasRegressor(build_fn=create_model, epochs=100, batch_size=4, verbose=0)
```

```
<ipython-input-31-cc1296524701>:1: DeprecationWarning: KerasRegressor is deprecated,
use Sci-Keras (https://github.com/adriangb/scikeras) instead. See https://www.adrian
gb.com/scikeras/stable/migration.html for help migrating.
  model = KerasRegressor(build_fn=create_model, epochs=100, batch_size=4, verbose=0)
```

Definimos el grid que contendrá los distinos valores a probar

En mi caso decidí probar variar la cantidad de neuronas por capa del modelo de redes neuronales:

In [ ]:
```
param_grid = {
    'neurons_layer1': [20, 50, 100],
    'neurons_layer2': [30, 60, 120],
    'neurons_layer3': [40, 80, 160]
}
```

Aplicamos GridSearch

In [ ]:
```
grid = GridSearchCV(estimator=model, param_grid=param_grid, scoring='neg_mean_squar
grid_result = grid.fit(X_train_scaled, y_train)
```

```
WARNING:tensorflow:5 out of the last 8 calls to <function Model.make_predict_functio
n.<locals>.predict_function at 0x78c40a62b490> triggered tf.function retracing. Trac
ing is expensive and the excessive number of tracings could be due to (1) creating @
tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3) pas
sing Python objects instead of tensors. For (1), please define your @tf.function out
side of the loop. For (2), @tf.function has reduce_retracing=True option that can av
oid unnecessary retracing. For (3), please refer to https://www.tensorflow.org/guid
e/function#controlling_retracing and https://www.tensorflow.org/api_docs/python/tf/f
unction for  more details.
WARNING:tensorflow:6 out of the last 10 calls to <function Model.make_predict_functi
on.<locals>.predict_function at 0x78c40a6304c0> triggered tf.function retracing. Tra
cing is expensive and the excessive number of tracings could be due to (1) creating
@tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3) pa
ssing Python objects instead of tensors. For (1), please define your @tf.function ou
tside of the loop. For (2), @tf.function has reduce_retracing=True option that can a
void unnecessary retracing. For (3), please refer to https://www.tensorflow.org/guid
e/function#controlling_retracing and https://www.tensorflow.org/api_docs/python/tf/f
unction for  more details.
```

Desplegamos los mejores hiper-parámetros arrojados por el modelo

In [ ]:
```
print("Mejores Hiperparámetros: ", grid_result.best_params_)
```

```
Mejores Hiperparámetros:  {'neurons_layer1': 100, 'neurons_layer2': 120, 'neurons_la
yer3': 80}
```

Evaluamos el modelo

In [ ]:
```
# Calculamos los errores correspondientes al modelo refinado
best_model = grid_result.best_estimator_
test_loss = best_model.score(X_test_scaled, y_test)
refined_predictions = best_model.predict(X_test_scaled)
refined_mae = mean_absolute_error(y_test, refined_predictions)



print("Prueba de pérdida con los mejores hiperparámetros: ", -test_loss)
```

```
print("Mean Squared Error (MSE):", mse)  # Print MSE
print("Root Mean Squared Error (RMSE):", np.sqrt(mse))  # Print RMSE
print("Mean Absolute Error (MAE):", refined_mae)  # Print MAE
```

```
Prueba de pérdida con los mejores hiperparámetros:  56.65126037597656
Mean Squared Error (MSE): 159.41923083757524
Root Mean Squared Error (RMSE): 12.626132853632392
Mean Absolute Error (MAE): 6.371757253011069
```

Conclusiones

Evaluación de Rendimiento del Modelo:

Los mejores hiperparámetros determinados por GridSearchCV son:

- Número de neuronas en la Capa 1: 100
- Número de neuronas en la Capa 2: 120
- Número de neuronas en la Capa 3: 80

La pérdida en la prueba con los mejores hiperparámetros es aproximadamente 56.65. Esta cifra refleja el nivel de precisión del modelo en datos no vistos.

1. **Comparación del MSE:** El modelo refinado tiene un MSE más bajo (56.651) en comparación con el modelo original (159.419), lo que indica que las predicciones del modelo refinado están más cerca de los valores reales.

2. **Comparación del RMSE:** El modelo refinado tiene un RMSE más bajo (aproximadamente 7.528) en comparación con el modelo original (12.626), lo que indica que las predicciones del modelo refinado tienen una dispersión menor.

3. **Comparación del MAE:** Compare el MAE del modelo refinado con el MAE del modelo original. Si el MAE del modelo refinado es más bajo, sugiere que las predicciones del modelo refinado están, en promedio, más cerca de los valores reales.

Interpretación:

- La elección de hiperparámetros indica una arquitectura de red neuronal relativamente compleja con un número considerable de neuronas en cada capa.

- La pérdida en la prueba, aunque no es excepcionalmente baja, es razonable y sugiere que el modelo realiza predicciones con una precisión moderada.

Evaluación General:

- El modelo ofrece un rendimiento razonable con los hiperparámetros seleccionados y ha sido refinado hasta proporcionar un rendimiento predictivo adecuado para la tarea en cuestión.

Recomendaciones:

Para mejorar aún más el rendimiento del modelo, se puede continuar experimentando con hiperparámetros, incluyendo diferentes arquitecturas de redes neuronales y técnicas de regularización.

También se puede explorar la posibilidad de recopilar más datos, ya que un conjunto de datos más grande a menudo ayuda a mejorar el rendimiento del modelo, aunque en mi caso, la base de datos de "Marvel_Movies" no es tan grande como lo esperaría, se podría considerar agregar otros lanzamientos como las series o películas más recientes que no se han incluido dentro del archivo, con el fin de experimentar un poco más con este modelo.

En resumen, el modelo funciona de manera razonable con los hiperparámetros seleccionados, y existe margen para mejoras adicionales a través de experimentación y posiblemente la adquisición de más datos.