Memoria Técnica de la Práctica de Deep Learning - Alvar García

En este documento se plasmará la experiencia de la realización de la práctica Deep Learning. En esta práctica se pretende crear una red neuronal que pueda predecir las métricas de engagement de unos POIs, a través de un dataset de imágenes de esos POIs y otro dataset de metadata sobre estos mismos POIs.

Con este fin, se comienza por la preparación de los datos que se van a usar, con el fin de obtener un conjunto de datos que se puedan pasar por una red neuronal y que ésta los pueda usar para aprender exitosamente a predecir las mencionadas engagement metrics.

1.1. Carga de Datos y paquetes:

En este epígrafe se procedió a la carga de los datos a usar en esta práctica, como también las distintas librerías que se usan en esta práctica. Estas librerías incluyen numpy y pandas para la manipulación de los datos y su análisis, seaborn y matplotlib para la visualización de los mismos, una librería de Deep Learning (he optado por PyTorch), Vision por computador (torchvision,que aparte nos da acceso a modelos preentrenados como ResNet), torch.utils.data para la carga de datos, sklearn para la normalización y evaluación, random para asegurar la reproducibilidad a través de la fijación de seeds, y module_utils, una librería de código con funciones auxiliares que se aporta a través de la carpeta de carga.

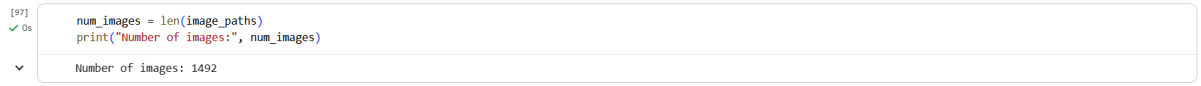
Para la ejecución de esta práctica, se utiliza SEED = 42.

La carga del csv de metadatos no tuvo mayor complicación, aunque para la carga de imágenes se usa un loop para recorrer el archivo donde se encuentran las imágenes, para extraer todas las diferentes imágenes (y posteriormente se transforman para que todas presenten el mismo tamaño).

2. Preparación y análisis de datos:

2.1. Análisis exploratorio del dataset:

Se visiona brevemente un extracto del dataset de imágenes para tener una idea de la calidad y naturaleza de las mismas, cuyo resultado se puede observar a continuación:



Como se puede observar, la calidad de las imágenes varía bastante. También se comprobó el número total de imágenes, el cual es de 1492.
A continuación se procedió a visionar y analizar el dataset de metadatos, para conocer la naturaleza de los datos y las distintas variables presentes en el mismo:

```
poi_df.head()
```

	id	name	shortDescription	categories	tier	locationLon	locationLat	tags	xps	Visits	Likes	Dislikes	Bookmarks	main_image_path
0	4b36a3ed-3b28-4bc7-b975-1d48b586db03	Galería Fran Reus	La Galería Fran Reus es un espacio dedicado a ...	['Escultura', 'Pintura']	1	2.642262	39.572694	[]	500	10009	422	3582	78	data_main/4b36a3ed-3b28-4bc7-b975-1d48b586db03...
1	e32b3603-a94f-49df-8b31-92445a86377c	Convento de San Plácido	El Convento de San Plácido en Madrid, fundado ...	['Patrimonio', 'Historia']	1	-3.704467	40.423037	[]	500	10010	7743	96	2786	data_main/e32b3603-a94f-49df-8b31-92445a86377c...
2	0123a69b-13ac-4b65-a5d5-71a95560cff5	Instituto Geológico y Minero de España	El Instituto Geológico y Minero de España, sit...	['Ciencia', 'Patrimonio']	2	-3.699694	40.442045	[]	250	10015	3154	874	595	data_main/0123a69b-13ac-4b65-a5d5-71a95560cff5...
3	390d7d9e-e972-451c-b5e4-f494af15e788	Margarita Gil Roësset	Margarita Gil Roësset, escultora y poetisa esp...	['Cultura']	1	-3.691228	40.427256	[]	500	10011	8559	79	2358	data_main/390d7d9e-e972-451c-b5e4-f494af15e788...
4	023fc1bf-a1cd-4b9f-af78-48792ab1a294	Museo del Traje. Centro de Investigación del P...	El Museo del Traje de Madrid, fundado en 2004,...	['Patrimonio', 'Cultura']	1	-3.727822	40.439665	[]	500	10020	915	2896	143	data_main/023fc1bf-a1cd-4b9f-af78-48792ab1a294...

```
poi_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1569 entries, 0 to 1568
Data columns (total 14 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   id                1569 non-null   object
 1   name              1569 non-null   object
 2   shortDescription  1569 non-null   object
 3   categories        1569 non-null   object
 4   tier              1569 non-null   int64
 5   locationLon       1569 non-null   float64
 6   locationLat       1569 non-null   float64
 7   tags              1569 non-null   object
 8   xps               1569 non-null   int64
 9   Visits            1569 non-null   int64
 10  Likes             1569 non-null   int64
 11  Dislikes          1569 non-null   int64
 12  Bookmarks         1569 non-null   int64
 13  main_image_path   1569 non-null   object
dtypes: float64(2), int64(6), object(6)
memory usage: 171.7+ KB
```

Como se puede observar, el dataset presenta variables numéricas y de tipo ‘object’, entre las cuales se encuentran:
- El id de la imágen (id)
- El nombre (name)
- Una breve descripción del POI (shortDescription)
- Categorías que se le asignan a cada POI (categories).
- Un tier que se le asigna a cada POI según su importancia (tier).
- Un par de variables de latitud y longitud (locationLat y locationLon)
- Tags que se le asignan a cada imagen (tags)
- Una variable de la experiencia que se obtiene al visitar cada POI (xps)

A parte de estas columnas, también se podían encontrar variables de engagement, que se describen a continuación:

- El número de visitas (Visits)
- El número de likes (Likes)
- El número de dislikes (Dislikes)
- El número de veces que el POI se ha guardado como bookmark (Bookmarks)
- El path de la imagen dentro de la carpeta (main_image_path) que coincide con el id.

Además, se identificó que el número de entradas de este dataset de metadata era mayor que el de las imágenes (1569 filas). Esto suscitó sospechas de posible presencia de

duplicados, lo cual se confirmó posteriormente:

```
id
a0f3f39c-fc87-4031-900d-d4776b1f3491    7
61a339d8-d9d6-4abc-9425-455d062e3338    6
060c7567-ef4a-4f09-9607-a4e3c2d4ad5c    6
5b38dcaa-19ac-422d-9c74-5b29e2a19629    6
fde6f250-9196-4ca8-8351-a9e8b15a63b6    6
ffd48a7d-f1f7-425c-9c52-fa534cc82167    5
3f17de6e-21e7-4d3a-b4ec-29c0630aebb4    5
48057ac3-5306-422f-95fa-0ff0937124bb    5
8313dc99-1dd6-4994-976a-505650fb5c7e    5
9d06c579-b645-41ce-b2f4-b57524cc7cea    5
bf345996-7402-47ac-abaa-cbfb19eefb0b    4
fb41c189-6d0a-488b-8c82-95dad97567f7    4
1728c4c3-7655-497c-b4ef-76d430ee2044    4
04be145b-f73e-41aa-9332-d1702684dc53    4
8083365a-4ad8-45a2-b08a-e7ea63bf1db0    4
23869080-e075-47f0-bb14-a07f6c704995    4
af0f8a7b-ca71-469e-a882-f20705acf206    4
31057c38-af33-46cc-95b1-341c037fec50    3
7f6e10d2-ff8d-447a-8f40-28d3c5c4c771    3
651efcc9-29a8-4781-bf73-10dfa65a7d69    3
f67f254e-da95-40ef-820c-602a2d09e979    3
d5c11570-41c3-40d2-bc1a-b3217780cd2c    2
e5427b0f-dd8c-4a31-bbe2-9c31a654a464    2
49b9c79c-9cef-4590-909f-c478ef21a10c    1
89145354-bb06-4fa2-8a67-0f1c905c0b29    1
24549199-144f-4710-8c03-164098bbf7d6    1
c9fa0bb9-71dd-414a-b110-30a2ec78e1b8    1
```

Se confirmó que, quitando los duplicados, los registros restantes eran 1492.

Además, el dataset no presentaba ningún dato faltante, con lo que no se tuvieron que tratar éstos últimos.

2.2. Preprocesamiento de datos:

2.2.1. Preprocesamiento de las imágenes:
2.2.2. Tratamiento de los datos faltantes:

- Ni las imágenes ni los datos faltantes (por ausencia de los mismos) se trataron en este paso.

2.2.3. Preprocesamiento de los metadatos:

Para tratar el problema de los registros duplicados, se crea una función de unificación, y un diccionario de columnas con funciones para conseguir la fusión de registros que más respetase la información de los mismos.

```
    def merge_unique_lists(series):
        # La suma de tags generará una lista con solo valores unicos
        items = []

        for sub in series:
            if pd.isna(sub):
                continue
            elif isinstance(sub, list):
                items.extend(sub)
            else:
                items.append(sub)

        return list(dict.fromkeys(items))
```

```
agg_funcs = {
    "Visits": "sum",      # se suman los valores de las columnas
    "Likes": "sum",
    "Dislikes": "sum",
    "Bookmarks": "sum",
    "categories": merge_unique_lists,
    "tags": merge_unique_lists,
    "name": "first",  # se fija el primer valor como valido
    "shortDescription": "first",
    "tier": "first",
    "locationLon": "first",
    "locationLat": "first",
    "xps": "first",
    "main_image_path": "first"

}

poi_df_final = poi_df.groupby("id", as_index=False).agg(agg_funcs)
```

- Se llevaron a cabo la ordenación de las listas que conforman los registros de ‘tags’ y ‘categories’ para conseguir un conteo más real de los datos de dichas variables
- También se decidió eliminar las variables ‘name’ y ‘shortDescription’, ya que se decidió que no aportan gran valor para la predicción de las métricas de engagement.
- Tras visualizar los datos de la variable ‘tags’, se decidió que ésta sería más útil si se tomase el número de tags, más que el significado semántico de cada tag.

2.2.3.1. Tratamiento de variables categóricas:

Tras descartar ‘tags’ como variables categórica habiéndose convertido en una numérica, quedaban las siguientes variables por tratar:

- Tier
- Categories

Tier:

Esta variable se trató como una variable categórica ordinal, y por tanto se proceso como tal. Los presentes originalmente eran 1,2,3,4, siendo 1 el más importante y 4 el menos. Para una correcta lectura de estos valores por parte de la red neuronal, primero se tenía que tener en cuenta la existencia del valor 0 (quedando así los cuatro tiers como 0,1,2 y 3), y segundo, se debía invertir el orden del tier, es decir, convertir todos los tier 0 en tier 3, ya que así quedaría alineado el valor de mayor importancia con el valor más alto. Por último, se normalizan los valores, quedando éstos (por orden ascendente de importancia) como 0, 0.33, 0.66 y 1.

Categorías:

Se concluye que hay 12 categorías distintas. Se decide usar un embedded encoder, donde se crean tensores para cada categoría, donde la proximidad semántica equivale a la cercanía matemática. Esta columna por tanto se convirtió en una lista de tensores.

2.3. Creación de la métrica de engagement:

La métrica de engagement es el valor por el cual se define la interacción que ha tenido el usuario con el POI dado. Para confeccionar la misma desde el punto de partida de nuestras variables (Likes, Dislikes, Visits y Bookmarks). Se decidió la siguiente fórmula:

$$\text{Engagement} = \text{Likes}/(\text{Visits} + e) - \text{Dislikes}/(\text{Visits} + e) + 2*\text{Bookmarks}/(\text{Visits} + e)$$

De esta forma, se aseguraba que cada POI se valoraba en proporción a las visitas que hubiese tenido, y se decidió ponderar los Bookmarks como el doble de valiosos que los likes, ya que se consideró que un usuario es más dado a poner un like que a molestarse en guardar el POI en un bookmark (mostrando así intención de volver a visitarlo potencialmente).

Los valores presentes en esta métrica varían entre -1 y 3 aproximadamente.

2.4. División estratificada del dataset:

Para preparar el dataset para la estratificación, se separa la métrica de engagement (el target) en un dataframe llamado ‘y’.

```
y =engagement_metric[["Engagement_metric"]]
```

```
y.head()
```

	Engagement_metric
0	-0.199600
1	1.278394
2	1.250750
3	1.322574
4	-0.338062

Los target se estratifican usando bins y qcut, asegurando que la distribución de cuantiles en los distintos splits es comparable (y así promoviendo un entrenamiento más preciso).

El dataset de metadatos quedó como se muestra a continuación:

```
X_meta = poi_df_final[["xps","tier_ord","num_tags","locationLon","locationLat","categories_idx","main_image_path"]]
X_meta.head()
```

	xps	tier_ord	num_tags	locationLon	locationLat	categories_idx	main_image_path
0	1000	1.000000	0	-5.776669	40.111279	[4, 11]	data_main/002b3c1f-37f0-4d36-84d4-2d7132f83fe7...
1	250	0.666667	0	-3.714472	40.410457	[10, 6]	data_main/005fe23a-7f9e-4a57-a24a-1e9a6f1c1702...
2	800	1.000000	0	-3.711620	40.406714	[1, 10]	data_main/00761a42-1b02-42f3-b2b2-211ffd985118...
3	150	0.333333	0	-3.714567	40.412984	[3]	data_main/0089bb20-c132-496c-b215-1986fcfa9d4b...
4	600	0.666667	0	-3.652625	40.450756	[10, 3]	data_main/00f483bf-a5a2-4009-bc67-b9f3f2e73488...

Tras llevar a cabo los splits de los metadatos e imágenes, se llevaron a cabo varias transformaciones (se llevan a cabo después de realizar los splits para evitar leakage).

- Se convirtieron las variables de latitud y longitud en coordenadas esféricas (tres variables, x, y, z). Esto se hizo a través de la definición de una función que se aplicó a cada split.

- Se normalizaron las variables numéricas ('xps', 'tier_ord', 'num_tags', 'x', 'y', 'z') usando StandardScaler.

Las imágenes fueron estratificadas teniendo en cuenta que su nombre debía coincidir con el nombre en main_image_path del dataset de metadatos (de esta forma, las distintas partes del dataset se alineaban y quedaban en los mismos splits). Por lo que la estratificación de las imágenes fue una 'imitación' de la estratificación de metadatos.

2.5. Data Augmentation:

Sobre el split 'train' de imágenes, se lleva a cabo un data augmentation, una técnica que genera muestras alteradas a partir de las que ya existen para multiplicar el número de imágenes con las que entrena la red neuronal, mejorando así su capacidad de aprendizaje, especialmente importante dado el bajo número de muestras que presenta el dataset que se está manejando.

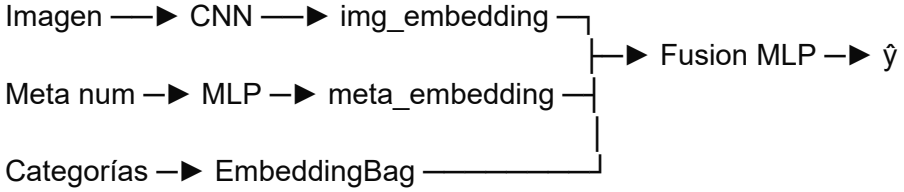
3. Arquitectura del Modelo:

El dataset que se pretende procesar consiste de varias partes:

- el dataset de metadata
- la capa del embeddedEncoder
- el dataset de imágenes

Para intentar obtener el mejor resultado posible, se van a tratar cada una por separado, y cada rama del proceso se va a fusionar al final de la red neuronal.

Esquema de la red neuronal Late Fusion propuesta:



3.1. Metadatos:
Para entrenar los metadatos, se va a utilizar una MLP (multilayer perceptron neural network), ya que son redes neuronales que funcionan bien para datos tabulares con un número alto de parámetros.

La red que se ha diseñado tiene una capa de entrada de 6 neuronas (una por cada variable que entra en la red (x,y,z,tier_ord, xps, num_tags).

La siguiente capa (la capa oculta) consta de 32 neuronas, y tiene una función de activación ReLu, que ayuda a mantener los gradientes, tiene un buen backpropagation, y funciona bien en redes neuronales de varias capas. Además tiene la ventaja añadida que es bastante rápida computacionalmente usando el GPU.

Aparte, se llevan a cabo dos acciones más sobre esta capa:

- BatchNorm: normaliza activaciones y hace las veces de regularizador, reduciendo la inestabilidad de los gradientes. Además reduce el overfitting. Tiene buena sinergia con ReLu porque centra el input de la neurona cerca de 0, manteniéndola activa.

- Dropout: setea la activación de un número aleatorio de neuronas (en mi caso, 0.2) a cero, evitando co-adaptación (que es muy probable en un dataset tan pequeño como el usado.

Con estas dos implementaciones, y manteniendo el número de neuronas de la capa oculta en un número pequeño (32 neuronas) espero evitar grandes problemas de overfitting. Aunque luego se valorará si ha sido efectivo o no.

Rama MLP:

```
[ ]
class MetaEncoder(nn.Module):
    def __init__(self, input_dim=6, output_dim=64):
        super().__init__()

        # Se van a usar capas ocultas de 32 neuronas para evitar overfitting en datasets pequeños
        # Se añade Dropout (0.2) para mejorar la generalización
        self.net = nn.Sequential(
            nn.Linear(input_dim, 32),
            nn.ReLU(),
            nn.BatchNorm1d(32),  # Estabiliza y acelera el entrenamiento normalizando las activaciones.
            nn.Dropout(0.2),     # Diseñado para evitar el overfitting,
                                 # forzando a la red a aprender representaciones redundantes.

            nn.Linear(32, output_dim),
            nn.ReLU()
        )

    def forward(self, x):
        return self.net(x)
```

3.2. Categorías:

Como se ha mencionado antes, el encoder es especialmente útil con categorías con significado semántico porque permite que dos categorías con significados semánticos parecidos acaben en espacios vectoriales cercanos, manteniendo la carga de información semántica.

Ademas, se usa EmbeddingBag porque realiza reducción y embedding en una operación, es más fiable con datasets pequeños, hace uso de los offsets y asi evita el uso de padding, puede aprender de relaciones semánticas, que para esta variable es aconsejable.

Rama de categorías:

```
class CategoryEncoder(nn.Module):
  # Crea un vector con el numero de categorias presentes en cada entrada del dataset
  # y usa el promedio para agregarlos. Este vector mantiene la informacion semántica
  # de las categorias.

    def __init__(self, num_categories, emb_dim=32):
        super().__init__()

        self.embedding = nn.EmbeddingBag(
            num_embeddings=num_categories,
            embedding_dim=emb_dim,
            mode="mean"
        )

    def forward(self, flat_cats, offsets):
        return self.embedding(flat_cats, offsets)
```

+ Code +

3.2. Imágenes:

Para las imágenes, se pretende usar una red neuronal convolucional. A diferencia de redes neuronales lineares (por ejemplo), la red no conecta cada pixel a cada neurona de la red, si no que pasa un filtro de un tamaño determinado por toda la imagen (en pasos o 'strides' determinados por el usuario) y va detectando patrones locales. Este tipo de redes neuronales son mucho más eficaces en imágenes que las MLPs.

En vez de usar una CNN no entrenada, se elige usar una red ResNet pre-entrenada, ya que goza de las siguientes ventajas:

En esta rama se escoge trabajar con una red convolucional pre entrenada por tener esta ventajas sobre una CNN sin pre entrenamiento. A continuación explico algunas de estas ventajas:

- ResNet viene con pesos optimizados pre entrenados con un altisimo volumen de imagenes de ImageNet.
- Es más indicado para datasets pequeños (como es el caso de este dataset, que solo consiste de 1500 entradas).
- El tiempo de entrenamiento es menor que en un CNN sin entrenar (este modelo puede llegar a tener un buen nivel de precisión a los 5-10 epochs, mientras que un modelo no entrenado puede tardar al rededor de los 100 epochs en conseguir los mismos niveles.
- En conjunto con el dropout que se implementamos en otras capas, una red pre entrenada es menos propensa a memorizar el ruido de nuestras imágenes.
- Tiene también menor riesgo de overfitting, y una precisión final en principio superior a una CNN sin entrenar.

Rama CNN (ResNet):

```
class ImageEncoder(nn.Module):
    def __init__(self, output_dim=256):
        super().__init__()

        backbone = models.resnet18(pretrained=True)
        self.feature_extractor = nn.Sequential(
            *list(backbone.children())[:-1]  # quitamos la capa fc
        )

        self.fc = nn.Linear(512, output_dim)

    def forward(self, x):
    # se define el forward pass
        x = self.feature_extractor(x)   # [batch, 512, 1, 1]
        x = x.view(x.size(0), -1)       # [batch, 512]
        return self.fc(x)
```

3.3. Capa de fusión:

La capa de fusión sirve como punto de convergencia de las tres partes mencionadas anteriormente. Nótese que se aplica la función de activación ReLu con un Dropout de 0.3 en este caso.

```
class FusionRegressor(nn.Module):
    def __init__(self, input_dim):
        super().__init__()

        self.net = nn.Sequential(
            nn.Linear(input_dim, 128),
            nn.ReLU(),
            nn.Dropout(0.3),   # Aseguramos que el regresor no se vuelva perezoso
            nn.Linear(128, 1) # Devuelve un valor
        )

    def forward(self, x):
        return self.net(x).squeeze(1)
```

A continuación se presenta el modelo final ensamblado:

[ ]

```python
class LateFusionModel(nn.Module):
    def __init__(self, num_categories):
        super().__init__()
    # Las diferentes ramas del modelo
        self.img_enc = ImageEncoder(256)
        self.meta_enc = MetaEncoder(6, 64)
        self.cat_enc = CategoryEncoder(num_categories, 32)
    # La capa de fusión
        self.regressor = FusionRegressor(256 + 64 + 32)

    def forward(self, images, meta, flat_cats, offsets):
        img_feat = self.img_enc(images)
        meta_feat = self.meta_enc(meta)
        cat_feat = self.cat_enc(flat_cats, offsets)

        fused = torch.cat([img_feat, meta_feat, cat_feat], dim=1)

        return self.regressor(fused)
```

4. Entrenamiento y Optimización:

Para este punto, primero se definió el pipeline de entrenamiento:

[ ]

```python
# Configuración del dispositivo (GPU si está disponible)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Usando dispositivo: {device}")

# Inicializar el modelo
model = LateFusionModel(num_categories=12).to(device)

# Definir función de pérdida y optimizador
criterion = nn.MSELoss()
optimizer = optim.AdamW(model.parameters(), lr=0.001, weight_decay=1e-4)

# Scheduler de learning rate - reduce el LR cuando val loss se estanque
scheduler = optim.lr_scheduler.ReduceLROnPlateau(
    optimizer, mode='min', factor=0.5, patience=3
)

# Configuración del early stop
class EarlyStopping:
    def __init__(self, patience=7, min_delta=0):
        self.patience = patience
        self.min_delta = min_delta
        self.counter = 0
        self.best_loss = None
        self.early_stop = False
        self.best_model = None
```

```
        def __call__(self, val_loss, model):
            if self.best_loss is None:
                self.best_loss = val_loss
                self.best_model = model.state_dict().copy()
            elif val_loss > self.best_loss - self.min_delta:
                self.counter += 1
                print(f'EarlyStopping counter: {self.counter} out of {self.patience}')
                if self.counter >= self.patience:
                    self.early_stop = True
            else:
                self.best_loss = val_loss
                self.best_model = model.state_dict().copy()
                self.counter = 0

    early_stopping = EarlyStopping(patience=7, min_delta=0.001)

    print("Pipeline de entrenamiento configurado correctamente.")
```

```
Usando dispositivo: cpu
Pipeline de entrenamiento configurado correctamente.
```

En esta celda, se definen las funciones de pérdida y el optimizador, el scheduler de learning rate (usamos reduceLROnPlateau, que como su propio nombre indica, reduce el LR a la mitad cuando se encuentra que validation loss no mejora en un epoch.

También se define el Earlystop, que en este caso finaliza el entrenamiento si este deja de mejorar después de 7 intentos.

A continuación se definen las funciones de entrenamiento y validación:

```
# Función de entrenamiento
def train_epoch(model, loader, criterion, optimizer, device):
    model.train()
    total_loss = 0

    for images, meta, flat_cats, offsets, y in tqdm(loader, desc="Training"):
        # Mover datos al dispositivo
        images = images.to(device)
        meta = meta.to(device)
        flat_cats = flat_cats.to(device)
        offsets = offsets.to(device)
        y = y.to(device)

        # Forward pass
        y_pred = model(images, meta, flat_cats, offsets)
        loss = criterion(y_pred, y)

        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    return total_loss / len(loader)
```

```
    return total_loss / len(loader)

# Función de validación
def validate_epoch(model, loader, criterion, device):
    model.eval()
    total_loss = 0
    all_preds = []
    all_targets = []

    with torch.no_grad():
        for images, meta, flat_cats, offsets, y in tqdm(loader, desc="Validation"):
            # Mover datos al dispositivo
            images = images.to(device)
            meta = meta.to(device)
            flat_cats = flat_cats.to(device)
            offsets = offsets.to(device)
            y = y.to(device)

            # Forward pass
            y_pred = model(images, meta, flat_cats, offsets)
            loss = criterion(y_pred, y)

            total_loss += loss.item()
            all_preds.extend(y_pred.cpu().numpy())
            all_targets.extend(y.cpu().numpy())

    return total_loss / len(loader), np.array(all_preds), np.array(all_targets)
```

y el loop principal de entrenamiento, con su parte informativa a traves de prints de datos clave:

```
# Loop principal de entrenamiento
num_epochs = 30
train_losses = []
val_losses = []
learning_rates = []

print("Iniciando entrenamiento...\n")

for epoch in range(num_epochs):
    print(f"\n{'='*60}")
    print(f"Epoch {epoch + 1}/{num_epochs}")
    print(f"{'='*60}")

    # Entrenar
    train_loss = train_epoch(model, train_loader, criterion, optimizer, device)
    train_losses.append(train_loss)

    # Validar
    val_loss, val_preds, val_targets = validate_epoch(model, val_loader, criterion, device)
    val_losses.append(val_loss)

    # Calcular métricas adicionales
    mae = np.mean(np.abs(val_preds - val_targets))
    rmse = np.sqrt(val_loss)

    # Calcular R²
    ss_res = np.sum((val_targets - val_preds) ** 2)
```

```
        # Calcular métricas adicionales
        mae = np.mean(np.abs(val_preds - val_targets))
        rmse = np.sqrt(val_loss)

        # Calcular R²
        ss_res = np.sum((val_targets - val_preds) ** 2)
        ss_tot = np.sum((val_targets - np.mean(val_targets)) ** 2)
        r2 = 1 - (ss_res / ss_tot)

        # Guardar learning rate actual
        current_lr = optimizer.param_groups[0]['lr']
        learning_rates.append(current_lr)

        # Imprimir resultados
        print(f"\nResultados del Epoch {epoch + 1}:")
        print(f"  Train Loss: {train_loss:.4f}")
        print(f"  Val Loss:   {val_loss:.4f}")
        print(f"  Val MAE:    {mae:.4f}")
        print(f"  Val RMSE:   {rmse:.4f}")
        print(f"  Val R²:     {r2:.4f}")
        print(f"  Learning Rate: {current_lr:.6f}")

        # Actualizar learning rate scheduler
        scheduler.step(val_loss)

        # Early stopping check
        early_stopping(val_loss, model)
```

```
    print(f"  Learning Rate: {current_lr:.6f}")

    # Actualizar learning rate scheduler
    scheduler.step(val_loss)

    # Early stopping check
    early_stopping(val_loss, model)
    if early_stopping.early_stop:
        print("\n⚠️  Early stopping activado!")
        print(f"Mejor validación loss: {early_stopping.best_loss:.4f}")
        # Restaurar el mejor modelo
        model.load_state_dict(early_stopping.best_model)
        break

print("\n" + "="*60)
print("Entrenamiento completado!")
print("="*60)

# Guardar el modelo final
torch.save({
    'epoch': epoch,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'train_losses': train_losses,
    'val_losses': val_losses,
}, '/content/drive/MyDrive/practicas_DL/late_fusion_model.pth')

print("\n✅ Modelo guardado exitosamente.")
```

Este entrenamiento dio los siguientes resultados (muestro solo el último epoch):

```
============================================================
Epoch 11/30
============================================================
Training: 100%|██████████| 33/33 [04:05<00:00,  7.44s/it]
Validation: 100%|██████████| 7/7 [00:18<00:00,  2.64s/it]

Resultados del Epoch 11:
  Train Loss: 0.4160
  Val Loss:   0.4785
  Val MAE:    0.4696
  Val RMSE:   0.6917
  Val R²:     0.4184
  Learning Rate: 0.000500
EarlyStopping counter: 7 out of 7

⚠️  Early stopping activado!
Mejor validación loss: 0.4535

============================================================
Entrenamiento completado!
```

Como se puede observar, el modelo se paró usando el earlyStop, con lo que el modelo empeoró en 7 epochs. Vemos también que el val loss es más de 10% mayor que el train los, lo que podría estar indicando un sobreajuste en el modelo.

El valor de R^2 nos indica que el modelo predice bien aproximadamente el 42% de la varianza del engagement en los conjuntos train y val, lo que deja bastante espacio para mejorar.

También se puede observar que el RMSE es 1.47 veces mayor que el MAE, lo cual es aceptable, pero también se podría mejorar.

Se intuye que una de los desafíos de este modelo es su bajo número de registros, haciendo el entrenamiento más difícil al tener pocos datos de los que aprender.

Con el fin de intentar mejorar el modelo, se aplicarán cambios que combatan el sobreajuste y así conseguir predicciones más precisas.

4.1. Implementación básica de técnicas anti-overfitting y Optimización de hiperparámetros:

Ya se explicaron en el apartado de arquitectura del modelo, con lo cual se procederá a modificar hiperparámetros para intentar conseguir un mejor modelo.

Como el problema está principalmente en el overfitting, vamos a buscar acentuar el efecto de los factores que protegen nuestro modelo del mismo.

- Para empezar, vamos a normalizar el target (no se había hecho antes por que no se consideró necesario, pero quizás ayude con el entrenamiento):

```python
# Se normaliza el target
y_scaler = StandardScaler()
y_train_scaled = y_scaler.fit_transform(y_train.values.reshape(-1, 1)).flatten()
y_val_scaled = y_scaler.transform(y_val.values.reshape(-1, 1)).flatten()
y_test_scaled = y_scaler.transform(y_test.values.reshape(-1, 1)).flatten()
```

- Se aumenta también la complejidad de la capa de fusión, aplicando aquí también Batchnorm y un Dropout más agresivo (de 0.2 a 0.5).
- Igualmente, se aumenta el Dropout en la rama MLP de 0.2 a 0.4.
- Se reduce el LR y se aumenta el weight decay, para más regularización.
- Por último, se añade la línea mostrada en la imagen al backwards pass en el modelo de entrenamiento:

```python
loss = criterion(y_pred, y)

# Backward pass
optimizer.zero_grad()
loss.backward()
#clip_grad_norm añadido después del primer entrenamiento
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
optimizer.step()

total_loss += loss.item()
```

Esto tiene como objetivo explosiones de gradiente que desestabilizan el entrenamiento correcto de la red.

Tras estas modificaciones, se volvió a entrenar la red con el conjunto de datos train/val, y los resultados fueron los siguientes:

```
  Val R²:     0.4995
  Learning Rate: 0.000250
EarlyStopping counter: 6 out of 7

============================================================
Epoch 11/30
============================================================
Training: 100%|██████████| 33/33 [00:19<00:00,  1.67it/s]
Validation: 100%|██████████| 7/7 [00:02<00:00,  2.71it/s]

Resultados del Epoch 11:
  Train Loss: 0.3809
  Val Loss:   0.4983
  Val MAE:    0.4480
  Val RMSE:   0.7059
  Val R²:     0.5145
  Learning Rate: 0.000250
EarlyStopping counter: 7 out of 7
```

Como se puede observar, el R^2 ha mejorado, prediciendo exitosamente el 51% de la varianza. Se observa una diferencia proporcional mayor entre MAE y RMSE, lo que indica

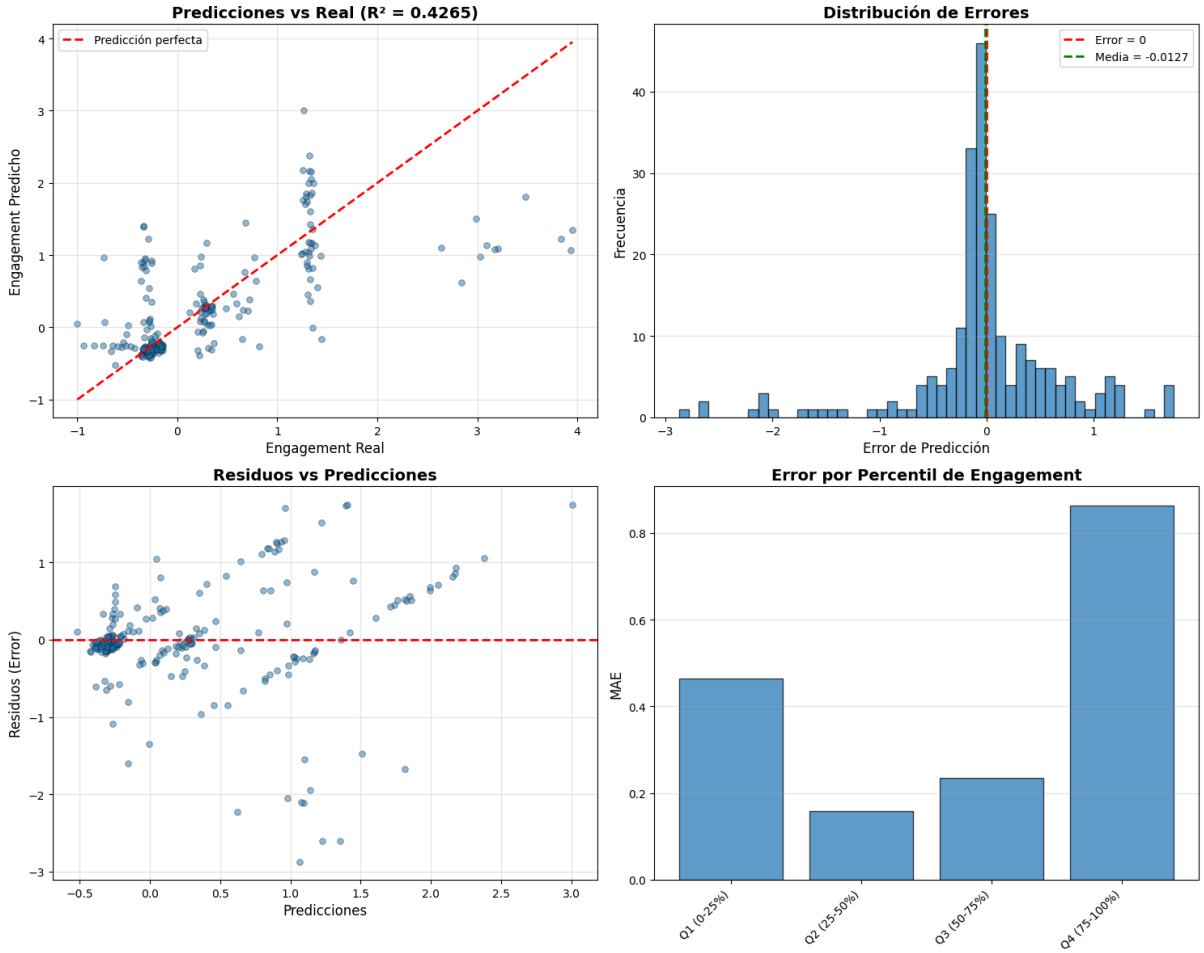
que el modelo parece tener problemas tratando outliers. La diferencia de train loss y val loss también es mayor, lo indica un overfitting en el modelo. Esto puede indicar que el modelo puede tener demasiada capacidad para este dataset.

5. Evaluación y Análisis

Tras estas modificaciones, se procede a evaluar el modelo sobre el conjunto test. Los resultados se visualizan a continuación:

```
================================================================================
📊 RESULTADOS EN TEST
================================================================================

MAE:  0.4304
RMSE: 0.6997
R²:   0.4265
================================================================================
```



📋 EJEMPLOS DE PREDICCIONES:
==========

✅ 5 MEJORES PREDICCIONES:

Real	Predicho	Error
0.2776	0.2785	0.0009
0.2738	0.2710	-0.0028
1.3548	1.3585	0.0037
-0.3459	-0.3497	-0.0038
-0.3043	-0.3102	-0.0059

❌ 5 PEORES PREDICCIONES:

Real	Predicho	Error
3.9374	1.0641	-2.8733
3.8352	1.2275	-2.6077
3.9523	1.3516	-2.6007
2.8455	0.6237	-2.2219
3.2025	1.0932	-2.1093

==========

🔍 ANÁLISIS DE ERRORES POR CARACTERÍSTICAS
==========

📊 Error Medio Absoluto (MAE) por TIER:

Tier_ord -3.8192531766310007: MAE = 0.8003 (n=1)
Tier_ord -2.2451117173326334: MAE = 0.5013 (n=15)
Tier_ord -0.6709702580342659: MAE = 0.3203 (n=93)
Tier_ord 0.9031712012641018: MAE = 0.5069 (n=115)

📊 Error Medio Absoluto (MAE) por NÚMERO DE TAGS:

Pocos (0-3): MAE = 0.3942 (n=156)

📊 Error por CATEGORÍAS más comunes:

Interpretación de los resultados:

El modelo propuesto alcanza en el conjunto de test los siguientes resultados:

- MAE: 0.4304
- RMSE: 0.6997
- R^2: 0.4265

Estos valores indican que el sistema es capaz de explicar aproximadamente el 43% de la variabilidad del engagement metric, lo que supone un rendimiento moderado teniendo en cuenta el tamaño reducido del dataset (aproximadamente 1500 muestras),

La diferencia entre RMSE y MAE (0.70 vs 0.43) sugiere la existencia de errores puntuales de gran magnitud, lo que indica que la distribución de los residuos no es homogénea y que el modelo presenta dificultades para predecir correctamente ciertos casos extremos.

En conclusión, el modelo claramente tiene claro espacio para mejorar. Convendría estudiar una forma de mejorar la precisión para valores extremos, intentando reducir más aún el overfitting, aumentando más el dropout y los weight decays con más detenimiento para optimizar más esta parte del modelo, pero por restricciones temporales, escapa de las posibilidades de esta experiencia. Se pone en evidencia la importancia del data augmentation para aumentar el volumen del dataset, con lo que se mejora bastante el entrenamiento.