

Explicacion delCodigo

Proyecto HAR: CNN-1D + MLP

Analisis Linea por Linea del Notebook

Conectando codigo con fundamentos teoricos

Contenido:

Configuracion • Carga de datos • EDA
Dataset y DataLoader • Arquitectura del modelo
Funciones de entrenamiento • Loop de entrenamiento
Evaluacion • Visualizacion

Documento complementario:

conceptos_fundamentales_har.pdf

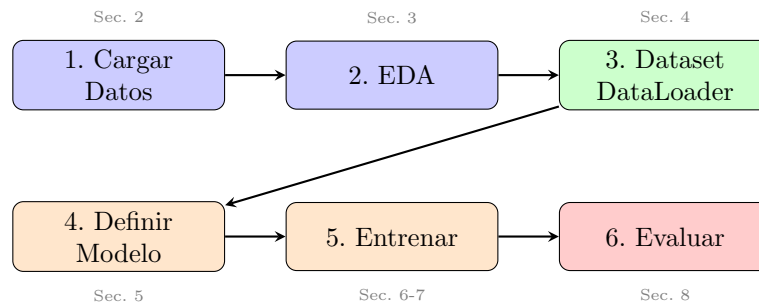
Las referencias **[Concepto N]** apuntan a secciones de ese documento.

Índice

1. Vision General del Pipeline	2
2. Configuracion del Entorno	2
2.1. Imports y Librerias	2
2.2. Configuracion de Graficos	3
2.3. Reproducibilidad y Device	3
3. Carga de Datos Inerciales	4
3.1. Definicion de Rutas y Archivos	4
3.2. Funcion load_signals	4
3.3. Funcion load_labels	6
3.4. Ejecucion de la Carga	6
4. Analisis Exploratorio de Datos (EDA)	7
4.1. Informacion General	7
4.2. Estadisticas por Canal	7
4.3. Distribucion de Clases	8
4.4. Matriz de Correlaciones	9
5. Dataset Personalizado y DataLoaders	9
5.1. Clase HARDataset	9
5.2. Division Train/Validation	10
5.3. Verificacion de Shapes	11
6. Arquitectura CNN-1D + MLP	12
6.1. Definicion de la Clase HARModel	12
6.2. Metodo Forward	13
6.3. Creacion del Modelo	14
7. Funciones de Entrenamiento	15
7.1. Funcion train_epoch	15
7.2. Funcion validate_epoch	17
7.3. Clase EarlyStopping	18
8. Loop de Entrenamiento Principal	19
8.1. Configuracion de Hiperparametros	19
8.2. Loop de Entrenamiento	20
9. Evaluacion en Test Set	22
9.1. Predicciones en Test	22
9.2. Reporte de Clasificacion	23
9.3. Matriz de Confusion	23
10. Visualizacion de Resultados	24
10.1. Curvas de Loss	24
10.2. Curvas de Accuracy	25
11. Resumen: Flujo Completo del Codigo	26

1. Vision General del Pipeline

Antes de analizar el codigo, veamos el flujo completo del proyecto:



Estructura del documento

Cada seccion sigue el mismo patron:

1. **Codigo:** El fragmento del notebook
2. **Explicacion:** Que hace cada linea
3. **Conexion teorica:** Referencia a conceptos fundamentales
4. **Diagrama** (cuando aplica): Visualizacion del proceso

2. Configuracion del Entorno

2.1. Imports y Librerias

Codigo: Imports

```

1  # Librerias principales
2  import torch                    # Framework de deep learning
3  import torch.nn as nn          # Modulos de redes neuronales
4  import torch.nn.functional as F # Funciones (ReLU, softmax, etc.)
5  from torch.utils.data import Dataset, DataLoader, random_split
6  import os                      # Operaciones del sistema
7
8  import numpy as np             # Operaciones numericas
9  import pandas as pd            # Manipulacion de datos
10 import matplotlib.pyplot as plt # Graficos
11 import seaborn as sns          # Graficos estadisticos
12 from sklearn.metrics import confusion_matrix, classification_report
13 import warnings
14 warnings.filterwarnings('ignore') # Ocultar advertencias
  
```

Explicacion linea por linea

<code>torch</code>	Framework principal de PyTorch para tensores y autograd
<code>torch.nn</code>	Contiene capas predefinidas (Conv1d, Linear, etc.)
<code>torch.nn.functional</code>	Funciones sin estado (ReLU, softmax, loss)
<code>Dataset, DataLoader</code>	Clases para manejar datos en batches
<code>random_split</code>	Para dividir train en train+validation
<code>numpy</code>	Operaciones matematicas eficientes con arrays
<code>pandas</code>	DataFrames para analisis exploratorio
<code>sklearn.metrics</code>	Metricas de evaluacion (accuracy, F1, etc.)

2.2. Configuracion de Graficos

Codigo: Estilo de graficos

```

1 # Configuracion de graficos
2 plt.style.use('seaborn-v0_8-whitegrid') # Estilo moderno
3 sns.set_palette("husl")                # Paleta de colores
4 plt.rcParams['figure.figsize'] = (12, 6) # Tamano por defecto
5 plt.rcParams['font.size'] = 11          # Tamano de fuente
6 %matplotlib inline                     # Mostrar en notebook

```

2.3. Reproducibilidad y Device

Codigo: Semillas y GPU

```

1 # Reproducibilidad: fijamos las semillas
2 torch.manual_seed(42)      # Semilla para PyTorch
3 np.random.seed(42)         # Semilla para NumPy
4
5 # Device: usar GPU si esta disponible
6 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

```

Conexion con Conceptos Fundamentales

[Concepto 1] – Por que fijar semillas

La semilla 42 inicializa el generador pseudo-aleatorio. Esto afecta:

- Inicializacion de pesos del modelo
- Orden de shuffle en DataLoader
- Neuronas desactivadas por Dropout
- Division train/validation

Sin semilla fija, cada ejecucion daria resultados diferentes.

Explicacion: Device

```
1 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

Esta linea es un **operador ternario**:

- Si `torch.cuda.is_available()` es `True` → usa `'cuda'` (GPU)
- Si es `False` → usa `'cpu'`

La GPU acelera operaciones de matrices (multiplicaciones en convoluciones y capas lineales).

3. Carga de Datos Inerciales

3.1. Definicion de Rutas y Archivos

Codigo: Configuracion de rutas

```
1 # Ruta base de los datos
2 BASE_PATH = '/content/drive/MyDrive/analitica/data'
3
4 # Nombres de los archivos de senales
5 signal_files = [
6     'body_acc_x', 'body_acc_y', 'body_acc_z',      # Aceleracion del
7     'body_gyro_x', 'body_gyro_y', 'body_gyro_z',   # Velocidad
8     'total_acc_x', 'total_acc_y', 'total_acc_z'    # Aceleracion
9 ]
```

Conexion con Conceptos Fundamentales

[Concepto 2] – Ejes X, Y, Z: Cada archivo contiene mediciones de UN eje espacial.

[Concepto 5] – Tipos de sensores:

- `body_acc`: Aceleracion SIN gravedad (movimiento puro)
- `total_acc`: Aceleracion CON gravedad (orientacion)
- `body_gyro`: Velocidad angular (rotacion)

3.2. Funcion `load_signals`

Codigo: Funcion para cargar senales

```
1 def load_signals(subset):
2     """
3     Carga las 9 senales de un subconjunto (train o test).
4
5     Args:
6         subset: 'train' o 'test'
```

```

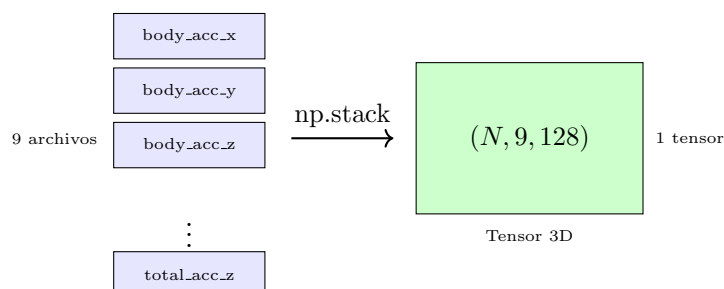
7
8     Returns:
9         X: numpy array de shape (N, 9, 128)
10    """
11    signals = []
12    for signal_name in signal_files:
13        filename = f'{signal_name}_{subset}.txt'
14        filepath = os.path.join(BASE_PATH, subset, 'data_train',
15                                filename)
16        # Cada archivo tiene N filas x 128 columnas (timesteps)
17        signal = np.loadtxt(filepath)
18        signals.append(signal)
19
20    # Apilar en un tensor 3D: (N, 9, 128)
21    X = np.stack(signals, axis=1)
22    return X

```

Explicacion detallada

Linea por linea:

1. `signals = []` – Lista vacia para acumular las 9 senales
2. `for signal_name in signal_files:` – Itera sobre los 9 nombres de archivos
3. `filename = f'{signal_name}_{subset}.txt'` – Construye nombre:
 - Si `signal_name='body_acc_x'` y `subset='train'`
 - Resultado: `'body_acc_x_train.txt'`
4. `np.loadtxt(filepath)` – Lee el archivo de texto:
 - Cada fila = 1 muestra (ventana de 2.56 segundos)
 - Cada columna = 1 timestep (de los 128)
 - Resultado: array ($N, 128$)
5. `signals.append(signal)` – Agrega a la lista (ahora tiene 9 arrays)
6. `np.stack(signals, axis=1)` – Apila los 9 arrays:
 - Cada array tiene shape ($N, 128$)
 - `axis=1` los pone en la dimension de canales
 - Resultado: ($N, 9, 128$)



3.3. Funcion load_labels

Codigo: Funcion para cargar etiquetas

```
1 def load_labels(subset):
2     """
3     Carga las etiquetas de un subconjunto.
4     Las etiquetas originales son 1-6, las convertimos a 0-5.
5     """
6     filepath = os.path.join(BASE_PATH, subset, f'y_{subset}.txt')
7     y = np.loadtxt(filepath, dtype=int)
8     return y - 1 # Convertir a indices 0-5
```

Por que restar 1?

Las etiquetas originales del dataset UCI HAR son 1-6:

1=WALKING, 2=UPSTAIRS, 3=DOWNSTAIRS, 4=SITTING, 5=STANDING,
6=LAYING

PyTorch espera indices desde 0, asi que restamos 1:

0=WALKING, 1=UPSTAIRS, 2=DOWNSTAIRS, 3=SITTING, 4=STANDING,
5=LAYING

3.4. Ejecucion de la Carga

Codigo: Cargar todos los datos

```
1 # Cargar datos
2 print("Cargando datos de entrenamiento...")
3 X_train = load_signals('train')
4 y_train = load_labels('train')
5
6 print("Cargando datos de test...")
7 X_test = load_signals('test')
8 y_test = load_labels('test')
9
10 print(f"X_train shape: {X_train.shape}") # (7352, 9, 128)
11 print(f"y_train shape: {y_train.shape}") # (7352,)
12 print(f"X_test shape: {X_test.shape}") # (2947, 9, 128)
13 print(f"y_test shape: {y_test.shape}") # (2947,)
```

Conexion con Conceptos Fundamentales

[Concepto 3] – 50 Hz y 128 timesteps

Cada muestra tiene 128 valores porque:

$$\text{Duracion} = \frac{128 \text{ muestras}}{50 \text{ Hz}} = 2.56 \text{ segundos}$$

[Concepto 4] – X e y en Machine Learning

- **X_train**: Features de entrada, shape $(N, C, T) = (7352, 9, 128)$
- **y_train**: Etiquetas de salida, shape $(N,) = (7352,)$

[Concepto 7] – Division train/test por sujetos

Los 7352 samples de train vienen de 21 personas diferentes a los 2947 de test (9 personas). Esto evita data leakage.

4. Analisis Exploratorio de Datos (EDA)

4.1. Informacion General

Codigo: Estadisticas basicas

```
1 print(f"Muestras de entrenamiento: {X_train.shape[0]:,}") # 7,352
2 print(f"Muestras de test: {X_test.shape[0]:,}") # 2,947
3 print(f"Canales (senales): {X_train.shape[1]:,}") # 9
4 print(f"Timesteps por muestra: {X_train.shape[2]:,}") # 128
5
6 print(f"X_train dtype: {X_train.dtype}") # float64
7 print(f"X_train: min={X_train.min():.4f}, max={X_train.max():.4f}")
8
9 print(f"X_train: {X_train.nbytes / 1024 / 1024:.2f} MB") # ~67 MB
```

Que nos dice cada linea

- **shape[0]** = numero de muestras (primera dimension)
- **shape[1]** = numero de canales (9 senales de sensores)
- **shape[2]** = numero de timesteps (128 puntos temporales)
- **dtype** = tipo de dato (float64 = numeros de punto flotante de 64 bits)
- **nbytes** = memoria ocupada en bytes

4.2. Estadisticas por Canal

Codigo: Calcular estadisticas

```
1 channel_names = [
2     'body_acc_x', 'body_acc_y', 'body_acc_z',
3     'body_gyro_x', 'body_gyro_y', 'body_gyro_z',
4     'total_acc_x', 'total_acc_y', 'total_acc_z'
5 ]
```

```

6
7 stats_data = []
8 for i, name in enumerate(channel_names):
9     channel_data = X_train[:, i, :].flatten()  # Todos los valores
        del canal
10     stats_data.append({
11         'Canal': name,
12         'Media': channel_data.mean(),
13         'Std': channel_data.std(),
14         'Min': channel_data.min(),
15         'Max': channel_data.max()
16     })
17
18 stats_df = pd.DataFrame(stats_data)
19 print(stats_df)

```

Explicacion: Indexacion y flatten

`X_train[:, i, :]`:

- : en posicion 0 = todas las muestras
- i en posicion 1 = canal especifico (0-8)
- : en posicion 2 = todos los timesteps

Resultado: array de shape (7352,128)

`.flatten()` convierte a 1D: $(7352 \times 128,) = (941,056,)$

Esto permite calcular estadisticas sobre TODOS los valores del canal.

4.3. Distribucion de Clases

Codigo: Contar clases

```

1 activity_names = ['WALKING', 'WALKING_UPSTAIRS', '
    WALKING_DOWNSTAIRS',
2                 'SITTING', 'STANDING', 'LAYING']
3
4 # np.unique retorna valores unicos y sus conteos
5 unique_train, counts_train = np.unique(y_train, return_counts=True)
6
7 for i, (label, count) in enumerate(zip(activity_names, counts_train
8 )):
9     percentage = (count / len(y_train)) * 100
10    print(f"{i}: {label:20s} | {count:4d} ({percentage:5.2f}%)")

```

Conexion con Conceptos Fundamentales

[Concepto 8] – Accuracy y balance de clases

Este analisis verifica que las clases estan balanceadas (13-19% cada una). Esto significa que el accuracy es una metrica confiable para evaluar el modelo.

4.4. Matriz de Correlaciones

Codigo: Calcular correlaciones

```

1 # Promediar cada canal sobre el tiempo
2 channel_means = np.mean(X_train, axis=2) # (N, 9) -> (7352, 9)
3
4 # Crear DataFrame y calcular correlaciones
5 corr_df = pd.DataFrame(channel_means, columns=channel_names)
6 correlation_matrix = corr_df.corr()
7
8 # Mostrar correlaciones fuertes
9 for i in range(len(channel_names)):
10     for j in range(i+1, len(channel_names)):
11         corr = correlation_matrix.iloc[i, j]
12         if abs(corr) > 0.5:
13             print(f"{channel_names[i]} <-> {channel_names[j]}: r = {corr:+.3f}")

```

Conexion con Conceptos Fundamentales

[Concepto 9] – Coeficiente de correlacion r

$|r| > 0.5$ indica correlacion fuerte. Esto nos dice que canales estan relacionados (por ejemplo, body_acc y total_acc difieren solo por la gravedad).

5. Dataset Personalizado y DataLoaders

5.1. Clase HARDataset

Codigo: Definicion del Dataset

```

1 class HARDataset(Dataset):
2     """
3     Dataset personalizado para HAR.
4
5     Hereda de torch.utils.data.Dataset y debe implementar:
6     - __len__: retorna el numero de muestras
7     - __getitem__: retorna una muestra y su etiqueta
8     """
9     def __init__(self, X, y):
10         self.X = torch.FloatTensor(X) # Convertir a tensor float32
11         self.y = torch.LongTensor(y) # Convertir a tensor int64
12
13     def __len__(self):
14         return len(self.X)
15
16     def __getitem__(self, idx):
17         return self.X[idx], self.y[idx]

```

Explicacion de cada metodo

`__init__(self, X, y):`

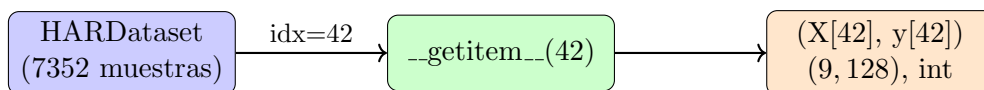
- Recibe arrays de NumPy y los convierte a tensores de PyTorch
- `FloatTensor`: para features (numeros decimales)
- `LongTensor`: para etiquetas (enteros para `CrossEntropyLoss`)

`__len__(self):`

- Retorna el numero total de muestras
- Usado por `DataLoader` para saber cuantos batches crear

`__getitem__(self, idx):`

- Recibe un indice y retorna la muestra correspondiente
- Retorna tupla: (features, etiqueta)
- `DataLoader` llama esto repetidamente para formar batches



5.2. Division Train/Validation

Codigo: Split y DataLoaders

```

1  # Crear datasets completos
2  train_full_dataset = HARDataset(X_train, y_train)
3  test_dataset = HARDataset(X_test, y_test)
4
5  # Split train -> train + validation (80/20)
6  train_size = int(0.8 * len(train_full_dataset)) # 5881
7  val_size = len(train_full_dataset) - train_size # 1471
8  train_dataset, val_dataset = random_split(
9      train_full_dataset,
10     [train_size, val_size]
11 )
12
13 # Crear DataLoaders
14 batch_size = 64
15
16 train_loader = DataLoader(train_dataset, batch_size=batch_size,
17                             shuffle=True)
18 val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle
19                             =False)
20 test_loader = DataLoader(test_dataset, batch_size=batch_size,
21                             shuffle=False)
  
```

Explicacion de parametros

random_split:

- Divide el dataset en partes de tamanos especificados
- La division es aleatoria (afectada por la semilla)
- 80 % para entrenar, 20 % para validacion

DataLoader:

- `batch_size=64`: Agrupa 64 muestras por batch
- `shuffle=True`: Mezcla los datos cada epoca (solo en train)
- `shuffle=False`: Mantiene orden fijo (en val/test)

Conexion con Conceptos Fundamentales

[Concepto 10] – Batch y Batch Size

Con `batch_size=64` y 5881 muestras de train:

$$\text{Batches por epoca} = \lceil 5881/64 \rceil = 92 \text{ batches}$$

Cada batch tiene shape (64, 9, 128) para X y (64,) para y.

5.3. Verificacion de Shapes

Codigo: Verificar un batch

```

1 # Obtener un batch de ejemplo
2 X_batch, y_batch = next(iter(train_loader))
3
4 print(f"X_batch: {X_batch.shape}") # torch.Size([64, 9, 128])
5 print(f"y_batch: {y_batch.shape}") # torch.Size([64])

```

Que hace `next(iter(...))`

1. `iter(train_loader)`: Crea un iterador sobre el DataLoader
2. `next(...)`: Obtiene el primer elemento (primer batch)

Es equivalente a:

```

1 for X_batch, y_batch in train_loader:
2     break # Solo tomar el primero

```

6. Arquitectura CNN-1D + MLP

6.1. Definicion de la Clase HARModel

Codigo: Arquitectura completa

```
1 class HARModel(nn.Module):
2     """
3     Arquitectura Hibrida CNN-1D + MLP para HAR
4     """
5     def __init__(self, n_channels=9, n_classes=6):
6         super(HARModel, self).__init__()
7
8         # BLOQUE CNN-1D
9         self.conv1 = nn.Conv1d(n_channels, 64, kernel_size=5,
10                                padding=2)
11         self.bn1 = nn.BatchNorm1d(64)
12
13         self.conv2 = nn.Conv1d(64, 128, kernel_size=5, padding=2)
14         self.bn2 = nn.BatchNorm1d(128)
15
16         self.pool = nn.MaxPool1d(kernel_size=2)
17         self.dropout = nn.Dropout(0.3)
18
19         # BLOQUE MLP
20         self.fc1 = nn.Linear(128 * 32, 256)
21         self.fc2 = nn.Linear(256, n_classes)
```

Explicacion de cada capa

nn.Conv1d(9, 64, kernel_size=5, padding=2):

- Entrada: 9 canales
- Salida: 64 filtros (canales de salida)
- Kernel: ventana de 5 timesteps
- Padding=2: agrega 2 ceros a cada lado (mantiene dimension)

nn.BatchNorm1d(64):

- Normaliza las activaciones por batch
- Estabiliza el entrenamiento

nn.MaxPool1d(kernel_size=2):

- Reduce dimension temporal a la mitad
- Toma el maximo de cada par de valores

nn.Dropout(0.3):

- Apaga 30 % de neuronas aleatoriamente
- Solo activo durante entrenamiento

nn.Linear(4096, 256):

- Capa fully connected
- Entrada: $128 \times 32 = 4096$ (canales \times timesteps)
- Salida: 256 neuronas

6.2. Metodo Forward

Codigo: Forward pass

```

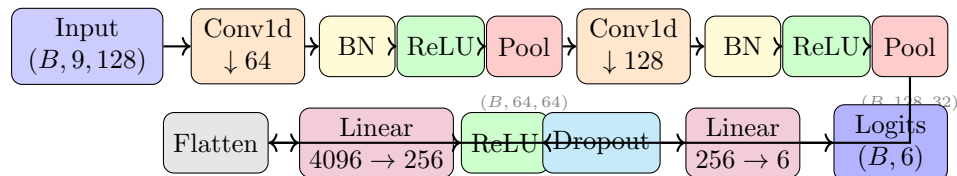
1 def forward(self, x):
2     # x shape: (B, 9, 128)
3
4     # Bloque Conv 1
5     x = self.conv1(x)           # (B, 64, 128)
6     x = self.bn1(x)             # Normalizacion
7     x = F.relu(x)               # Activacion
8     x = self.pool(x)            # (B, 64, 64)
9
10    # Bloque Conv 2
11    x = self.conv2(x)            # (B, 128, 64)
12    x = self.bn2(x)              # Normalizacion
13    x = F.relu(x)                # Activacion
14    x = self.pool(x)             # (B, 128, 32)
15
16    # Flatten: convertir tensor 3D a 2D
17    x = x.view(x.size(0), -1)    # (B, 4096)

```

```

18
19 # MLP
20 x = self.fc1(x)           # (B, 256)
21 x = F.relu(x)            # Activacion
22 x = self.dropout(x)      # Regularizacion
23
24 x = self.fc2(x)          # (B, 6) - Logits
25
26 return x

```



Conexion con Conceptos Fundamentales

[Concepto 11] – Ejemplos numericos de operaciones

Cada operacion tiene una formula:

- **Conv1d:** $(x * w)[t] = \sum_k x[t + k] \cdot w[k]$
- **MaxPool:** $\text{Pool}(x)[i] = \max(x[2i], x[2i + 1])$
- **ReLU:** $\text{ReLU}(x) = \max(0, x)$
- **Linear:** $y = Wx + b$

[Concepto 15] – Calculo de parametros

Total: 1,094,790 parametros (ver seccion 15 del documento de conceptos).

Explicacion de `x.view(x.size(0), -1)`

`view` cambia la forma del tensor sin copiar datos:

- `x.size(0)` = tamaño del batch (64)
- `-1` = “calcula automaticamente esta dimension”

Ejemplo:

```

1 x.shape # torch.Size([64, 128, 32])
2 x.view(64, -1).shape # torch.Size([64, 4096])
3 # Porque: 128 * 32 = 4096

```

6.3. Creacion del Modelo

Codigo: Instanciar y mover a GPU

```

1 # Crear modelo
2 model = HARModel(n_channels=9, n_classes=6).to(device)
3
4 # Contar parametros

```

```

5 total_params = sum(p.numel() for p in model.parameters())
6 trainable_params = sum(p.numel() for p in model.parameters() if p.
    requires_grad)
7
8 print(f"Total:          {total_params:,}")          # 1,094,790
9 print(f"Entrenables: {trainable_params:,}")          # 1,094,790

```

Explicacion del conteo de parametros

`model.parameters()` retorna un generador con todos los tensores de pesos.

`p.numel()` retorna el numero de elementos (Number of Elements).

Ejemplo para `Conv1d(9, 64, kernel=5)`:

```

1 # Pesos: 64 filtros x (9 canales x 5 kernel)
2 weights = 64 * 9 * 5 = 2880
3 # Bias: 1 por filtro
4 bias = 64
5 # Total: 2880 + 64 = 2944

```

7. Funciones de Entrenamiento

7.1. Funcion train_epoch

Codigo: Entrenamiento de una epoca

```

1 def train_epoch(model, loader, criterion, optimizer, device):
2     """
3     Entrena el modelo por una epoca.
4
5     Returns:
6         (avg_loss, accuracy)
7     """
8     model.train() # Modo entrenamiento
9     total_loss = 0
10    correct = 0
11    total = 0
12
13    for X_batch, y_batch in loader:
14        X_batch = X_batch.to(device)
15        y_batch = y_batch.to(device)
16
17        # Forward pass
18        outputs = model(X_batch)          # (B, 6) logits
19        loss = criterion(outputs, y_batch) # Cross-entropy
20
21        # Backward pass (backpropagation)
22        optimizer.zero_grad() # Limpiar gradientes anteriores
23        loss.backward()        # Calcular gradientes
24        optimizer.step()       # Actualizar pesos
25
26        # Metrics
27        total_loss += loss.item() * X_batch.size(0)
28        predictions = outputs.argmax(dim=1)

```

```

29         correct += (predictions == y_batch).sum().item()
30         total += y_batch.size(0)
31
32     avg_loss = total_loss / total
33     accuracy = correct / total
34     return avg_loss, accuracy

```

Explicacion del ciclo de entrenamiento

model.train():

- Activa Dropout (apaga neuronas)
- BatchNorm usa estadísticas del batch actual

.to(device):

- Mueve tensores a GPU (si disponible)
- Necesario para que modelo y datos esten en el mismo dispositivo

optimizer.zero_grad():

- PyTorch acumula gradientes por defecto
- Debemos limpiarlos antes de cada batch

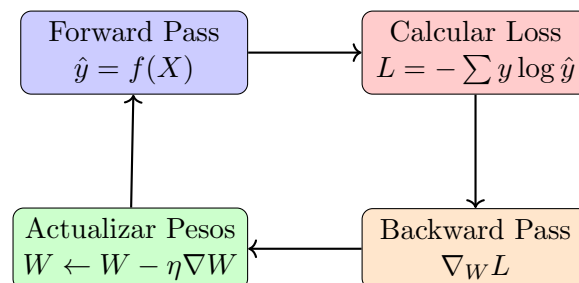
loss.backward():

- Calcula $\frac{\partial L}{\partial W}$ para todos los pesos
- Usa la regla de la cadena (backpropagation)

optimizer.step():

- Actualiza pesos: $W = W - \eta \cdot \nabla W$
- η = learning rate

Para cada batch



Conexion con Conceptos Fundamentales

[Concepto 13] – Funciones `train_epoch` y `validate_epoch`

La diferencia clave entre `train` y `validate`:

- **Train:** Calcula gradientes y actualiza pesos
- **Validate:** Solo evalua, no modifica el modelo

7.2. Funcion `validate_epoch`

Codigo: Validacion de una epoca

```
1 def validate_epoch(model, loader, criterion, device):
2     """
3     Evalua el modelo en el conjunto de validacion.
4     """
5     model.eval() # Modo evaluacion
6     total_loss = 0
7     correct = 0
8     total = 0
9
10    with torch.no_grad(): # NO calcular gradientes
11        for X_batch, y_batch in loader:
12            X_batch = X_batch.to(device)
13            y_batch = y_batch.to(device)
14
15            outputs = model(X_batch)
16            loss = criterion(outputs, y_batch)
17
18            total_loss += loss.item() * X_batch.size(0)
19            predictions = outputs.argmax(dim=1)
20            correct += (predictions == y_batch).sum().item()
21            total += y_batch.size(0)
22
23    avg_loss = total_loss / total
24    accuracy = correct / total
25    return avg_loss, accuracy
```

Diferencias con train_epoch

Aspecto	train_epoch	validate_epoch
Modo	model.train()	model.eval()
Dropout	Activo (30 %)	Desactivado
torch.no_grad()	No	Si
zero_grad()	Si	No
backward()	Si	No
step()	Si	No

torch.no_grad():

- Desactiva el calculo de gradientes
- Ahorra memoria (no guarda activaciones)
- Acelera la inferencia

Conexion con Conceptos Fundamentales

[Concepto 12] – Test Forward Pass

Durante evaluacion:

- model.eval() desactiva comportamiento estocastico
- torch.no_grad() evita overhead de autograd
- El modelo es deterministico

7.3. Clase EarlyStopping

Codigo: Early Stopping

```

1 class EarlyStopping:
2     """
3     Detiene el entrenamiento si el loss de validacion no mejora.
4     """
5     def __init__(self, patience=15, min_delta=0):
6         self.patience = patience          # Epocas a esperar
7         self.min_delta = min_delta         # Mejora minima requerida
8         self.counter = 0                   # Epocas sin mejora
9         self.best_loss = float('inf')     # Mejor loss hasta ahora
10        self.best_state = None              # Pesos del mejor modelo
11        self.early_stop = False            # Flag de parada
12
13    def __call__(self, val_loss, model):
14        if val_loss < self.best_loss - self.min_delta:
15            # Mejora: guardar estado del modelo
16            self.best_loss = val_loss
17            self.best_state = {k: v.cpu().clone()
18                               for k, v in model.state_dict().items()
19                               }
19            self.counter = 0
20        else:
21            # No mejora: incrementar contador

```

```

22         self.counter += 1
23         if self.counter >= self.patience:
24             self.early_stop = True
25
26         return self.early_stop

```

Como funciona EarlyStopping

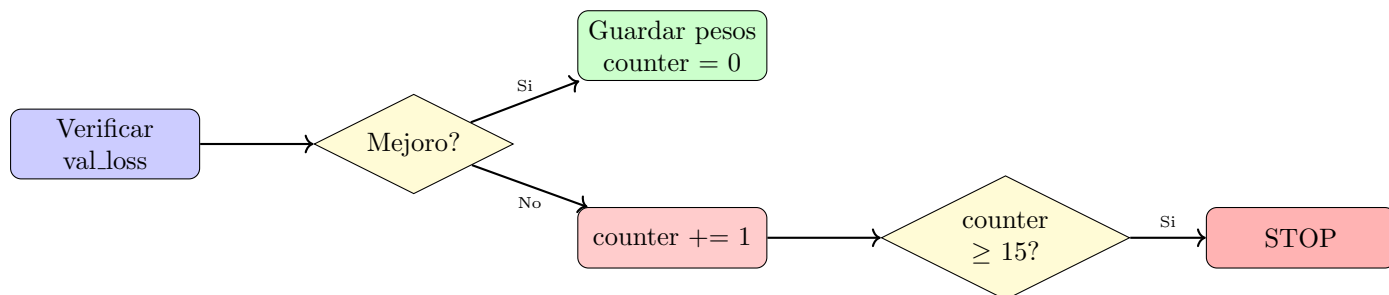
1. Cada epoca, compara `val_loss` con `best_loss`
2. Si mejora: guarda los pesos y resetea contador
3. Si NO mejora: incrementa contador
4. Si contador \geq patience: activa `early_stop`

`model.state_dict():`

- Diccionario con todos los pesos del modelo
- Claves: nombres de capas ('conv1.weight', 'fc1.bias', etc.)
- Valores: tensores con los pesos

`.cpu().clone():`

- Copia los tensores a CPU
- Evita que se modifiquen cuando el modelo sigue entrenando



8. Loop de Entrenamiento Principal

8.1. Configuracion de Hiperparametros

Codigo: Hiperparametros

```

1  # Hiperparametros
2  learning_rate = 1e-3      # 0.001
3  weight_decay = 1e-2      # 0.01 (regularizacion L2)
4  num_epochs = 100         # Maximo de epocas
5
6  # Definir loss y optimizador
7  criterion = nn.CrossEntropyLoss()
8  optimizer = torch.optim.AdamW(

```

```

9     model.parameters(),
10    lr=learning_rate,
11    weight_decay=weight_decay
12 )
13
14 # Early stopping
15 early_stopping = EarlyStopping(patience=15)

```

Conexion con Conceptos Fundamentales

[Concepto 14] – Hiperparametros y sus valores

Parametro	Valor	Razon
Learning rate	10^{-3}	Estandar para Adam
Weight decay	10^{-2}	Regularizacion L2 moderada
Patience	15	Esperar mejoras lentas

CrossEntropyLoss explicado

`nn.CrossEntropyLoss()` combina:

1. **Softmax**: Convierte logits a probabilidades
2. **Negative Log-Likelihood**: Mide error de clasificacion

Formula:

$$L = -\log \left(\frac{e^{z_y}}{\sum_j e^{z_j}} \right)$$

Donde z_y es el logit de la clase correcta.

AdamW:

- Adam + decoupled weight decay
- Mejor que Adam vanilla para regularizacion

8.2. Loop de Entrenamiento

Codigo: Loop principal

```

1  # Historial para graficos
2  history = {
3      'train_loss': [], 'train_acc': [],
4      'val_loss': [], 'val_acc': []
5  }
6
7  # Entrenamiento
8  for epoch in range(num_epochs):
9      # Entrenar una epoca
10     train_loss, train_acc = train_epoch(
11         model, train_loader, criterion, optimizer, device
12     )
13

```

```
14     # Validar
15     val_loss, val_acc = validate_epoch(
16         model, val_loader, criterion, device
17     )
18
19     # Guardar historial
20     history['train_loss'].append(train_loss)
21     history['train_acc'].append(train_acc)
22     history['val_loss'].append(val_loss)
23     history['val_acc'].append(val_acc)
24
25     # Imprimir progreso cada 5 epocas
26     if (epoch + 1) % 5 == 0:
27         print(f"Epoch {epoch+1:3d}/{num_epochs} | "
28               f"Train Loss: {train_loss:.4f}, Acc: {train_acc:.4f}
29               | "
30               f"Val Loss: {val_loss:.4f}, Acc: {val_acc:.4f}")
31
32     # Early stopping
33     if early_stopping(val_loss, model):
34         print(f"Early stopping en epoca {epoch+1}")
35         break
36
37     # Cargar mejor modelo
38     if early_stopping.best_state is not None:
39         model.load_state_dict(early_stopping.best_state)
```

Flujo del loop

1. Para cada epoca (hasta 100 o early stop):

- Entrenar con train_loader → (loss, acc)
- Evaluar con val_loader → (loss, acc)
- Guardar metricas en history
- Verificar early stopping

2. Si early stopping activa:

- Romper el loop
- Restaurar pesos del mejor modelo

Por que restaurar el mejor modelo?

El modelo al final del entrenamiento NO es necesariamente el mejor. Early stopping guarda los pesos cuando `val_loss` era minimo. Al final, restauramos esos pesos para usar el modelo optimo.

9. Evaluacion en Test Set

9.1. Predicciones en Test

Codigo: Evaluar en test

```
1 # Evaluar en test set
2 model.eval()
3 all_predictions = []
4 all_labels = []
5
6 with torch.no_grad():
7     for X_batch, y_batch in test_loader:
8         X_batch = X_batch.to(device)
9         outputs = model(X_batch)
10        predictions = outputs.argmax(dim=1).cpu().numpy()
11
12        all_predictions.extend(predictions)
13        all_labels.extend(y_batch.numpy())
14
15 all_predictions = np.array(all_predictions)
16 all_labels = np.array(all_labels)
17
18 # Calcular accuracy global
19 test_accuracy = (all_predictions == all_labels).mean()
20 print(f"Test Accuracy: {test_accuracy:.4f}") # 0.9564
```

Explicacion paso a paso

outputs.argmax(dim=1):

- outputs tiene shape $(B, 6)$
- `argmax(dim=1)` encuentra el indice del maximo en cada fila
- Resultado: tensor de shape $(B,)$ con predicciones 0-5

.cpu().numpy():

- Mueve tensor de GPU a CPU
- Convierte a array de NumPy (para sklearn)

.extend():

- Agrega elementos de un iterable a la lista
- Diferente de `.append()` que agrega el objeto completo

9.2. Reporte de Clasificacion

Codigo: Metricas por clase

```
1 from sklearn.metrics import classification_report
2
3 report = classification_report(
4     all_labels,
5     all_predictions,
6     target_names=activity_names,
7     digits=4
8 )
9 print(report)
```

Conexion con Conceptos Fundamentales

[Concepto 8] – Metricas de evaluacion

El reporte incluye para cada clase:

- **Precision:** $\frac{TP}{TP+FP}$
- **Recall:** $\frac{TP}{TP+FN}$
- **F1-Score:** $2 \cdot \frac{P \cdot R}{P+R}$
- **Support:** Numero de muestras reales de esa clase

[Concepto 14] – El F1-score es la media armonica de Precision y Recall.

9.3. Matriz de Confusion

Codigo: Calcular y visualizar

```
1 from sklearn.metrics import confusion_matrix
2
3 cm = confusion_matrix(all_labels, all_predictions)
4
5 # Visualizar
6 fig, ax = plt.subplots(figsize=(10, 8))
7 sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
8             xticklabels=activity_names,
9             yticklabels=activity_names, ax=ax)
10 ax.set_xlabel('Prediccion')
11 ax.set_ylabel('Real')
12 ax.set_title('Matriz de Confusion')
13 plt.show()
```

Interpretacion de la matriz

- **Diagonal:** Predicciones correctas
- **Fuera de diagonal:** Errores (confusiones)
- Fila i , columna j : Cuantas veces se predijo j cuando era i

Observacion tipica: SITTING y STANDING se confunden porque ambas son estaticas ($\text{body_acc} \approx 0$).

10. Visualizacion de Resultados

10.1. Curvas de Loss

Codigo: Graficar loss

```

1 epochs_range = range(1, len(history['train_loss']) + 1)
2 best_epoch = np.argmin(history['val_loss']) + 1
3
4 fig, ax = plt.subplots(figsize=(12, 5))
5
6 ax.plot(epochs_range, history['train_loss'], 'b-', label='Train
7         Loss')
8 ax.plot(epochs_range, history['val_loss'], 'r-', label='Validation
9         Loss')
10 ax.axvline(best_epoch, color='green', linestyle='--',
11            label=f'Mejor epoca ({best_epoch})')
12
13 ax.set_title('Loss vs Epoca')
14 ax.set_xlabel('Epoca')
15 ax.set_ylabel('Cross-Entropy Loss')
16 ax.legend()
17 ax.grid(True, alpha=0.3)
18 plt.show()

```

Interpretacion de curvas

- **Train loss baja, val loss baja:** Modelo aprendiendo bien
- **Train loss baja, val loss sube:** Overfitting
- **Gap pequeno:** Buena generalizacion
- **Gap grande:** Posible overfitting

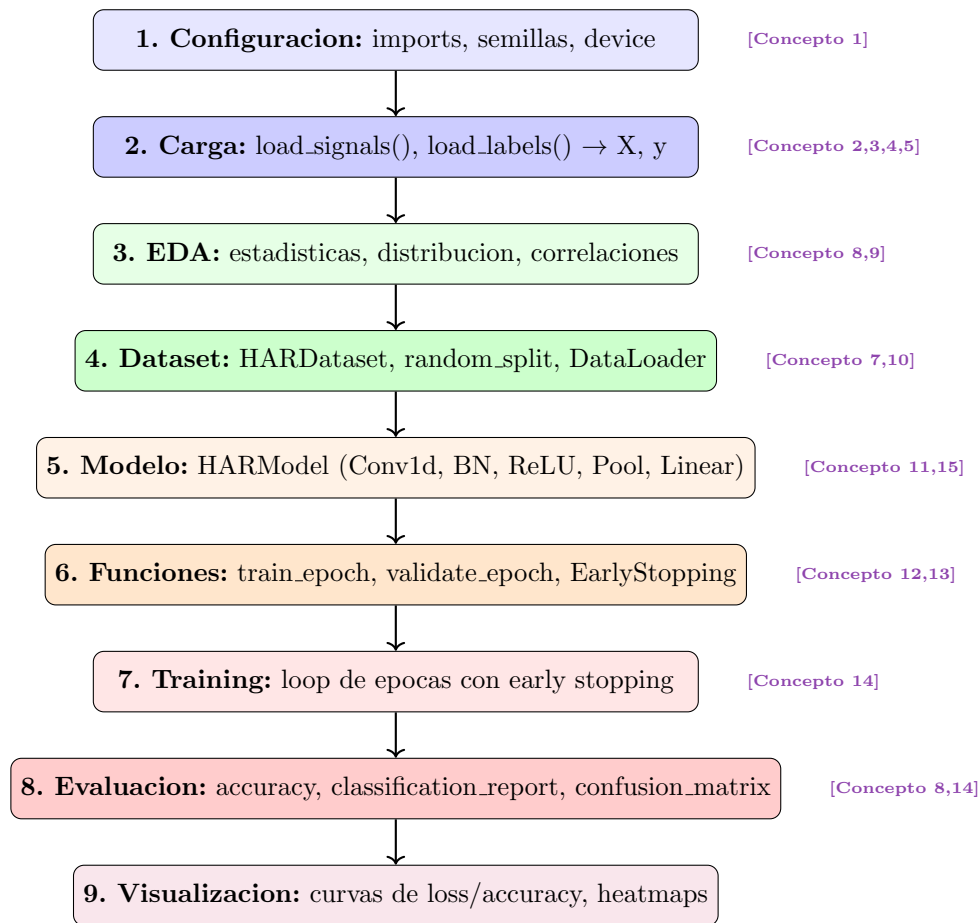
`np.argmin()`: Retorna el indice del valor minimo. Sumamos 1 porque las epocas empiezan en 1.

10.2. Curvas de Accuracy

Codigo: Graficar accuracy

```
1 fig, ax = plt.subplots(figsize=(12, 5))
2
3 ax.plot(epochs_range, history['train_acc'], 'b-', label='Train
  Accuracy')
4 ax.plot(epochs_range, history['val_acc'], 'r-', label='Validation
  Accuracy')
5 ax.axvline(best_epoch, color='green', linestyle='--')
6
7 ax.set_title('Accuracy vs Epoca')
8 ax.set_xlabel('Epoca')
9 ax.set_ylabel('Accuracy')
10 ax.legend()
11 ax.grid(True, alpha=0.3)
12 plt.show()
```

11. Resumen: Flujo Completo del Codigo



Resultado Final

Test Accuracy: 95.64 %

El modelo clasifica correctamente 2818 de 2947 muestras de test, distinguiendo entre 6 actividades humanas usando solo datos de sensores inerciales.

Puntos clave del código:

- Arquitectura CNN-1D extrae patrones temporales automáticamente
- BatchNorm y Dropout previenen overfitting
- Early Stopping detiene el entrenamiento en el punto óptimo
- Division por sujetos garantiza evaluación realista