Guia Completa del Proyecto

Clasificacion de Actividades Humanas (HAR)
con Redes Neuronales Hibridas CNN-1D + MLP

Fundamentos de Tensores en PyTorch

Enero 2026

Índice

1. Introduccion y Objetivo

Este documento conecta los conceptos de **Algebra Lineal** y **Estadistica** con la implementacion practica en PyTorch para resolver el problema de clasificacion de actividades humanas usando datos de sensores inerciales de smartphones.

1.1. El Problema

Dado un conjunto de senales de acelerometro y giroscopio, queremos predecir cual de las 6 actividades esta realizando una persona:

Clase	**Actividad**
1	WALKING (Caminando)
2	WALKING_UPSTAIRS (Subiendo escaleras)
3	WALKING_DOWNSTAIRS (Bajando escaleras)
4	SITTING (Sentado)
5	STANDING (De pie)
6	LAYING (Acostado)

Cuadro 1: Clases del dataset HAR

2. Estructura del Dataset

2.1. Archivos Disponibles

Los datos se encuentran en `AI/data/` con la siguiente estructura:

```
data/
  train/
    X_train.txt          (7352 muestras x 561 features)
    y_train.txt          (7352 etiquetas)
    Inertial Signals/
      body_acc_x_train.txt   (7352 x 128 timesteps)
      body_acc_y_train.txt
      body_acc_z_train.txt
      body_gyro_x_train.txt
      body_gyro_y_train.txt
      body_gyro_z_train.txt
      total_acc_x_train.txt
      total_acc_y_train.txt
      total_acc_z_train.txt
  test/
    (misma estructura con 2947 muestras)
```

2.2. Interpretacion Matematica de los Datos

Representacion Tensorial

Cada muestra de `Inertial Signals` es una **serie temporal multivariada**:

$$\mathbf{X} \in \mathbb{R}^{N \times C \times T}$$

Donde:

- N = numero de muestras (7352 train, 2947 test)
- $C = 9$ canales (3 acc_body + 3 gyro + 3 acc_total)
- $T = 128$ pasos temporales (2.56 segundos a 50Hz)

Conexion con Algebra Lineal: Cada muestra individual es una **matriz** $\mathbf{X}_i \in \mathbb{R}^{C \times T}$ donde cada fila representa una senal del sensor a lo largo del tiempo.

3. Fundamentos Matematicos

3.1. Tensores: Generalizacion de Vectores y Matrices

Definicion Formal

Un **tensor** de orden n es una estructura algebraica que generaliza:

- Orden 0: Escalar $s \in \mathbb{R}$
- Orden 1: Vector $\mathbf{v} \in \mathbb{R}^d$
- Orden 2: Matriz $\mathbf{A} \in \mathbb{R}^{m \times n}$
- Orden 3+: Tensor $\mathcal{T} \in \mathbb{R}^{d_1 \times d_2 \times \cdots \times d_n}$

En PyTorch, el tensor de entrada para este proyecto tiene forma:

```
X = torch.randn(batch_size, 9, 128)  # (B, C, T)
```

3.2. Convolucion 1D: Extraccion de Patrones Temporales

La operacion de **convolucion** es el corazon de las CNN. En 1D:

Convolucion 1D – Definicion Matematica

Sea $\mathbf{x} \in \mathbb{R}^T$ una senal de entrada y $\mathbf{w} \in \mathbb{R}^K$ un kernel (filtro) de tamano K. La convolucion discreta es:

$$(\mathbf{x} * \mathbf{w})[t] = \sum_{k=0}^{K-1} \mathbf{x}[t+k] \cdot \mathbf{w}[k]$$

Interpretacion:

- Es un **producto punto deslizante** entre el kernel y ventanas de la senal.
- El kernel "aprende" a detectar patrones locales (ej. picos de aceleracion).
- Es similar a calcular una **media movil ponderada** donde los pesos se optimizan.

En PyTorch:

```
# Entrada: (batch, canales_in, tiempo)
# Salida:  (batch, canales_out, tiempo_nuevo)
conv = nn.Conv1d(in_channels=9, out_channels=32, kernel_size=5)
```

Formula de dimension de salida:

$$T_{out} = \left\lfloor \frac{T_{in} - K + 2P}{S} \right\rfloor + 1$$

Donde K = kernel_size, P = padding, S = stride.

3.3. Pooling: Reduccion de Dimensionalidad

Max Pooling selecciona el valor maximo en cada ventana:

$$\text{MaxPool}(\mathbf{x}, k)[i] = \max_{j=0}^{k-1} \mathbf{x}[i \cdot k + j]$$

Por que funciona:

- Reduce la dimension temporal (menos parametros).
- Proporciona **invarianza a pequenas traslaciones**.
- Mantiene las caracteristicas mas relevantes (los maximos).

3.4. Capa Lineal (Fully Connected): Transformacion Afin

La capa `nn.Linear` implementa:

Transformacion Afin

$$\mathbf{y} = \mathbf{x}\mathbf{W}^T + \mathbf{b}$$

Donde:

- $\mathbf{x} \in \mathbb{R}^{d_{in}}$ es el vector de entrada
- $\mathbf{W} \in \mathbb{R}^{d_{out} \times d_{in}}$ es la matriz de pesos
- $\mathbf{b} \in \mathbb{R}^{d_{out}}$ es el vector de sesgo (bias)
- $\mathbf{y} \in \mathbb{R}^{d_{out}}$ es la salida

Conexion con Algebra Lineal:

- $\mathbf{W}$ actua como una **transformacion lineal** que rota/escala el espacio.
- $\mathbf{b}$ **traslada** el origen.
- Geometricamente: proyecta datos de d_{in} dimensiones a d_{out} dimensiones.

3.5. Funciones de Activacion: No Linealidad

Sin funciones de activacion, multiples capas lineales colapsan en una sola (por asociatividad de matrices):

$$\mathbf{W}_2(\mathbf{W}_1\mathbf{x}) = (\mathbf{W}_2\mathbf{W}_1)\mathbf{x} = \mathbf{W}_{equiv}\mathbf{x}$$

3.5.1. ReLU (Rectified Linear Unit)

$$\text{ReLU}(x) = \max(0, x) = \begin{cases} x & \text{si } x > 0 \\ 0 & \text{si } x \leq 0 \end{cases} \tag{1}$$

Propiedades:

- Introduce no linealidad permitiendo aprender funciones complejas.
- Gradiente: $\frac{d}{dx}\text{ReLU}(x) = \mathbf{1}_{x>0}$ (funcion indicadora).
- Problema: "neuronas muertas" cuando $x \leq 0$ siempre.

3.5.2. Softmax: Probabilidades de Clase

Para clasificacion multiclase con K clases:

$$\text{Softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{2}$$

Propiedades estadisticas:

- $\sum_{i=1}^{K} \text{Softmax}(\mathbf{z})_i = 1$ (suma a 1).
- Cada salida $\in (0, 1)$ (interpretable como probabilidad).
- Convierte "logits" $\mathbf{z}$ en una **distribucion de probabilidad categorica**.

3.6. Regularizacion: Evitando el Sobreajuste

3.6.1. Dropout

Durante el entrenamiento, "apaga" neuronas aleatoriamente con probabilidad p:

$$\tilde{h}_i = \begin{cases} \frac{h_i}{1-p} & \text{con probabilidad } 1-p \\ 0 & \text{con probabilidad } p \end{cases} \tag{3}$$

Interpretacion estadistica: Entrena un **ensemble implicito** de subredes.

3.6.2. Batch Normalization

Normaliza las activaciones dentro de cada mini-batch:

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \tag{4}$$

Donde μ_B y σ_B^2 son la media y varianza del mini-batch.

Conexion con Estadistica: Aplica **estandarizacion (z-score)** dinamicamente durante el entrenamiento.

4. Funcion de Perdida: Cross-Entropy

Para clasificacion multiclase, usamos **Cross-Entropy Loss**:

$$\mathcal{L} = -\sum_{i=1}^{N}\sum_{c=1}^{K} y_{i,c}\log(\hat{p}_{i,c}) \tag{5}$$

Donde:

- $y_{i,c} \in \{0,1\}$ es 1 si la muestra i pertenece a la clase c.
- $\hat{p}_{i,c}$ es la probabilidad predicha para la clase c.

Conexion con Teoria de la Informacion

La Cross-Entropy mide la "distancia" entre la distribucion real P y la predicha Q:

$$H(P,Q) = -\mathbb{E}_P[\log Q] = -\sum_x P(x)\log Q(x)$$

Minimizar Cross-Entropy es equivalente a maximizar la **verosimilitud** (Maximum Likelihood).

En PyTorch:

```
# CrossEntropyLoss espera logits (sin softmax) y etiquetas como indices
criterion = nn.CrossEntropyLoss()
loss = criterion(logits, labels)  # logits: (B, 6), labels: (B,)
```

5. Optimizacion: Descenso de Gradiente

5.1. Gradiente: Direccion de Maximo Crecimiento

El **gradiente** de la funcion de perdida respecto a los parametros es:

$$\nabla_\theta \mathcal{L} = \left[\frac{\partial \mathcal{L}}{\partial \theta_1}, \frac{\partial \mathcal{L}}{\partial \theta_2}, \ldots, \frac{\partial \mathcal{L}}{\partial \theta_n}\right] \tag{6}$$

5.2. Regla de Actualizacion (SGD)

Descenso de Gradiente Estocastico

$$\theta_{t+1} = \theta_t - \eta\nabla_\theta \mathcal{L}(\theta_t)$$

Donde η es el **learning rate** (tasa de aprendizaje).

Interpretacion geometrica: Nos movemos en direccion **opuesta** al gradiente para descender hacia el minimo.

5.3. Adam: Optimizador Adaptativo

Adam combina momentum y tasas de aprendizaje adaptativas:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \quad \text{(momento de primer orden)} \tag{7}$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \quad \text{(momento de segundo orden)} \tag{8}$$
$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{9}$$

Donde $\hat{m}_t$ y $\hat{v}_t$ son versiones corregidas por sesgo.
En PyTorch:

```
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-3, weight_decay
    =1e-2)
```

6. Backpropagation: Regla de la Cadena

Para calcular gradientes en redes profundas, usamos la **regla de la cadena**:

Regla de la Cadena

Si $z = f(y)$ y $y = g(x)$, entonces:

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

Para una red $\mathcal{L} = f_3(f_2(f_1(\mathbf{x})))$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_1} = \frac{\partial \mathcal{L}}{\partial f_3} \cdot \frac{\partial f_3}{\partial f_2} \cdot \frac{\partial f_2}{\partial f_1} \cdot \frac{\partial f_1}{\partial \mathbf{W}_1}$$

En PyTorch:

```
loss.backward()  # Calcula todos los gradientes automaticamente (
    autograd)
```

7. Arquitectura Propuesta: CNN-1D + MLP

7.1. Diagrama de Flujo

```
Entrada: (B, 9, 128)
    |
    v
[Conv1d + BatchNorm + ReLU + MaxPool] x N bloques
    |
    v
Flatten: (B, features)
    |
    v
[Linear + ReLU + Dropout] x M capas
    |
```

```
    v
Linear(features, 6)
    |
    v
Salida: (B, 6) logits -> Softmax -> Probabilidades
```

7.2. Codigo de Referencia

```python
class HARModel(nn.Module):
    def __init__(self, n_channels=9, n_classes=6):
        super().__init__()

        # Bloque CNN-1D
        self.conv1 = nn.Conv1d(n_channels, 64, kernel_size=5, padding=2)
        self.bn1 = nn.BatchNorm1d(64)
        self.conv2 = nn.Conv1d(64, 128, kernel_size=5, padding=2)
        self.bn2 = nn.BatchNorm1d(128)
        self.pool = nn.MaxPool1d(kernel_size=2)
        self.dropout = nn.Dropout(0.3)

        # Calcular dimension despues de convs
        # 128 -> pool -> 64 -> pool -> 32
        self.fc1 = nn.Linear(128 * 32, 256)
        self.fc2 = nn.Linear(256, n_classes)

    def forward(self, x):
        # x: (B, 9, 128)
        x = self.pool(F.relu(self.bn1(self.conv1(x))))  # (B, 64, 64)
        x = self.pool(F.relu(self.bn2(self.conv2(x))))  # (B, 128, 32)
        x = x.view(x.size(0), -1)  # Flatten: (B, 128*32)
        x = self.dropout(F.relu(self.fc1(x)))
        x = self.fc2(x)  # Logits: (B, 6)
        return x
```

8. Pasos para Completar la Tarea

8.1. Paso 1: Cargar los Datos

```python
import numpy as np
import torch
from torch.utils.data import Dataset, DataLoader

def load_inertial_signals(folder, subset):
    """Carga los 9 canales de senales inerciales."""
    signals = []
    files = ['body_acc_x', 'body_acc_y', 'body_acc_z',
             'body_gyro_x', 'body_gyro_y', 'body_gyro_z',
             'total_acc_x', 'total_acc_y', 'total_acc_z']

    for f in files:
        path = f"{folder}/{subset}/Inertial Signals/{f}_{subset}.txt"
        data = np.loadtxt(path)  # (N, 128)
        signals.append(data)

```

```
    # Stack: (N, 9, 128)
    return np.stack(signals, axis=1)

# Cargar datos
X_train = load_inertial_signals('data', 'train')  # (7352, 9, 128)
y_train = np.loadtxt('data/train/y_train.txt') - 1  # Clases 0-5
```

8.2. Paso 2: Crear Dataset y DataLoader

```
class HARDataset(Dataset):
    def __init__(self, X, y, transform=None):
        self.X = torch.FloatTensor(X)
        self.y = torch.LongTensor(y)
        self.transform = transform

    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        x = self.X[idx]
        if self.transform:
            x = self.transform(x)
        return x, self.y[idx]

# Crear datasets
train_dataset = HARDataset(X_train, y_train)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
```

8.3. Paso 3: Definir el Loop de Entrenamiento

```
def train_epoch(model, loader, criterion, optimizer, device):
    model.train()
    total_loss, correct = 0, 0

    for X_batch, y_batch in loader:
        X_batch, y_batch = X_batch.to(device), y_batch.to(device)

        optimizer.zero_grad()
        outputs = model(X_batch)
        loss = criterion(outputs, y_batch)
        loss.backward()
        optimizer.step()

        total_loss += loss.item() * X_batch.size(0)
        correct += (outputs.argmax(1) == y_batch).sum().item()

    return total_loss / len(loader.dataset), correct / len(loader.
dataset)
```

8.4. Paso 4: Implementar Early Stopping

```
class EarlyStopping:
    def __init__(self, patience=10):
        self.patience = patience
```

```
        self.best_loss = float('inf')
        self.counter = 0
        self.best_state = None

    def step(self, val_loss, model):
        if val_loss < self.best_loss:
            self.best_loss = val_loss
            self.best_state = {k: v.cpu().clone()
                                for k, v in model.state_dict().items()}
            self.counter = 0
            return False
        self.counter += 1
        return self.counter >= self.patience
```

8.5. Paso 5: Graficar Resultados

```
import matplotlib.pyplot as plt

def plot_training(history):
    fig, axes = plt.subplots(1, 2, figsize=(12, 4))

    axes[0].plot(history['train_loss'], label='Train')
    axes[0].plot(history['val_loss'], label='Validation')
    axes[0].set_title('Loss vs Epoch')
    axes[0].legend()

    axes[1].plot(history['train_acc'], label='Train')
    axes[1].plot(history['val_acc'], label='Validation')
    axes[1].set_title('Accuracy vs Epoch')
    axes[1].legend()

    plt.savefig('training_curves.png')
    plt.show()
```

8.6. Paso 6: Guardar el Modelo

```
# Guardar
torch.save(model.state_dict(), 'har_model.pth')

# Cargar
model = HARModel()
model.load_state_dict(torch.load('har_model.pth'))
```

9. Metricas de Evaluacion

9.1. Accuracy (Exactitud)

$$\text{Accuracy} = \frac{\text{Predicciones Correctas}}{\text{Total de Predicciones}} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}[\hat{y}_i = y_i] \tag{10}$$

9.2. Precision, Recall y F1-Score

Para cada clase c:

$$\text{Precision}_c = \frac{TP_c}{TP_c + FP_c} \tag{11}$$

$$\text{Recall}_c = \frac{TP_c}{TP_c + FN_c} \tag{12}$$

$$\text{F1}_c = 2 \cdot \frac{\text{Precision}_c \cdot \text{Recall}_c}{\text{Precision}_c + \text{Recall}_c} \tag{13}$$

10. Rubrica de Evaluacion

Criterio	**Puntos**
Arquitectura CNN-1D + MLP (creatividad)	25
Entrenamiento correcto (loop, loss, accuracy)	25
Regularizacion (Dropout + BatchNorm)	20
Graficas (loss y accuracy vs epoch)	15
Guardado y carga del modelo	15
Total	**100**

Nota importante: Debes poder explicar el **que**, **como** y **por que** de cada decision de diseno.

11. Resumen de Conexiones Matematicas

Concepto PyTorch	**Matematica**	**Por que importa**
`torch.Tensor`	Tensor de orden n	Estructura de datos fundamental
`nn.Conv1d`	Convolucion discreta	Detecta patrones temporales locales
`nn.Linear`	Transformacion afin $\mathbf{W}\mathbf{x} + \mathbf{b}$	Proyecta a nuevo espacio
`F.relu`	$\max(0, x)$	Introduce no linealidad
`F.softmax`	$\frac{e^{z_i}}{\sum e^{z_j}}$	Convierte logits a probabilidades
`CrossEntropyLoss`	$-\sum y_c \log \hat{p}_c$	Mide error de clasificacion
`loss.backward()`	Regla de la cadena	Calcula gradientes
`optimizer.step()`	$\theta \leftarrow \theta - \eta \nabla \mathcal{L}$	Actualiza pesos
`nn.Dropout`	Mascara aleatoria	Regularizacion
`nn.BatchNorm1d`	Estandarizacion z-score	Estabiliza entrenamiento

Cuadro 2: Mapeo de PyTorch a Matematicas