Hoja de Trabajo 2

- Josue Marroquin 22397
- Sebastian Huertas 22295

In [11]:
```
# Libs
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import seaborn as sns
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score, classification_report
import copy
import torch.nn.init as init
import time
from collections import defaultdict
from sklearn.metrics import f1_score
```

Ejercicio 1 - Experimentación Práctica

Task 1 - Preparación del conjunto de datos

Cargue el conjunto de datos de Iris utilizando bibliotecas como sklearn.datasets. Luego, divida el conjunto de datos en conjuntos de entrenamiento y validación.

In [2]:
```
# Cargar el conjunto de datos de Iris
iris = load_iris()
X = iris.data  # Características: sepal length, sepal width, petal length, petal wi
y = iris.target  # Etiquetas: 0=setosa, 1=versicolor, 2=virginica

# Crear DataFrame para mejor visualización
feature_names = iris.feature_names
target_names = iris.target_names

df = pd.DataFrame(X, columns=feature_names)
df['target'] = y
df['species'] = df['target'].map({0: target_names[0], 1: target_names[1], 2: target

print("Información del conjunto de datos:")
print(f"Forma del dataset: {X.shape}")
```

```
print(f"Número de características: {X.shape[1]}")
print(f"Número de muestras: {X.shape[0]}")
print(f"Clases: {target_names}")
print(f"Distribución de clases: {np.bincount(y)}")

# Mostrar las primeras filas
print("\nPrimeras 5 filas del dataset:")
print(df.head())

# Dividir el conjunto de datos en entrenamiento y validación (80% - 20%)
X_train, X_val, y_train, y_val = train_test_split(
    X, y,
    test_size=0.2,
    random_state=42,
    stratify=y  # Mantener la proporción de clases en ambos conjuntos
)

print(f"\nConjunto de entrenamiento: {X_train.shape[0]} muestras")
print(f"Conjunto de validación: {X_val.shape[0]} muestras")
print(f"Distribución en entrenamiento: {np.bincount(y_train)}")
print(f"Distribución en validación: {np.bincount(y_val)}")

# Estandarizar las características (opcional pero recomendado para muchos algoritmo
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)

print("\nDatos preparados exitosamente para entrenamiento y validación")
print("Variables disponibles:")
print("- X_train, X_val: características originales")
print("- X_train_scaled, X_val_scaled: características estandarizadas")
print("- y_train, y_val: etiquetas")
print("- df: DataFrame completo con información descriptiva")
```

```
Información del conjunto de datos:
Forma del dataset: (150, 4)
Número de características: 4
Número de muestras: 150
Clases: ['setosa' 'versicolor' 'virginica']
Distribución de clases: [50 50 50]

Primeras 5 filas del dataset:
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  \
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2

   target species
0       0  setosa
1       0  setosa
2       0  setosa
3       0  setosa
4       0  setosa

Conjunto de entrenamiento: 120 muestras
Conjunto de validación: 30 muestras
Distribución en entrenamiento: [40 40 40]
Distribución en validación: [10 10 10]

Datos preparados exitosamente para entrenamiento y validación
Variables disponibles:
- X_train, X_val: características originales
- X_train_scaled, X_val_scaled: características estandarizadas
- y_train, y_val: etiquetas
- df: DataFrame completo con información descriptiva
```

Task 2 - Arquitectura modelo

Cree una red neuronal feedforward simple utilizando nn.Module de PyTorch. Luego, defina capa de entrada, capas ocultas y capa de salida. Después, elija las funciones de activación y el número de neuronas por capa

In [3]:
```python
# Task 2 - Arquitectura del modelo

# Configurar dispositivo (GPU si está disponible, sino CPU)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Dispositivo utilizado: {device}")

# Definir la arquitectura de la red neuronal feedforward
class IrisClassifier(nn.Module):
    def __init__(self, input_size=4, hidden_size1=16, hidden_size2=8, num_classes=3
        super(IrisClassifier, self).__init__()

        # Definir las capas
        self.fc1 = nn.Linear(input_size, hidden_size1)      # Capa de entrada: 4 ->
        self.fc2 = nn.Linear(hidden_size1, hidden_size2)    # Primera capa oculta:
```

```
        self.fc3 = nn.Linear(hidden_size2, num_classes)     # Capa de salida: 8 ->

        # Capa de dropout para regularización
        self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        # Forward pass a través de la red
        x = F.relu(self.fc1(x))         # Activación ReLU en primera capa
        x = self.dropout(x)             # Aplicar dropout
        x = F.relu(self.fc2(x))         # Activación ReLU en segunda capa
        x = self.dropout(x)             # Aplicar dropout
        x = self.fc3(x)                 # Capa de salida (sin activación, se aplica
        return x

# Crear instancia del modelo
model = IrisClassifier(input_size=4, hidden_size1=16, hidden_size2=8, num_classes=3
model = model.to(device)

# Mostrar la arquitectura del modelo
print("Arquitectura del modelo:")
print(model)
print(f"\nNúmero total de parámetros: {sum(p.numel() for p in model.parameters())}"
print(f"Parámetros entrenables: {sum(p.numel() for p in model.parameters() if p.req

# Convertir datos de numpy a tensores de PyTorch
X_train_tensor = torch.FloatTensor(X_train_scaled).to(device)
X_val_tensor = torch.FloatTensor(X_val_scaled).to(device)
y_train_tensor = torch.LongTensor(y_train).to(device)
y_val_tensor = torch.LongTensor(y_val).to(device)

print(f"\nForma de los tensores:")
print(f"X_train: {X_train_tensor.shape}")
print(f"X_val: {X_val_tensor.shape}")
print(f"y_train: {y_train_tensor.shape}")
print(f"y_val: {y_val_tensor.shape}")

# Crear DataLoaders para entrenamiento por lotes
batch_size = 16
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
val_dataset = TensorDataset(X_val_tensor, y_val_tensor)

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)

print(f"\nDataLoaders creados:")
print(f"Tamaño del lote: {batch_size}")
print(f"Número de lotes de entrenamiento: {len(train_loader)}")
print(f"Número de lotes de validación: {len(val_loader)}")

# Definir función de pérdida y optimizador
criterion = nn.CrossEntropyLoss()  # Para clasificación multiclase
optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=1e-4)

print(f"\nConfiguración de entrenamiento:")
print(f"Función de pérdida: {criterion}")
print(f"Optimizador: {optimizer}")
```

```
print(f"Tasa de aprendizaje: 0.001")

# Función para probar el modelo con datos de ejemplo
def test_forward_pass():
    model.eval()
    with torch.no_grad():
        # Tomar una muestra pequeña para probar
        sample_input = X_train_tensor[:3]  # 3 muestras
        output = model(sample_input)
        probabilities = F.softmax(output, dim=1)
        predictions = torch.argmax(output, dim=1)

        print(f"\nPrueba del forward pass:")
        print(f"Input shape: {sample_input.shape}")
        print(f"Output shape: {output.shape}")
        print(f"Predictions: {predictions.cpu().numpy()}")
        print(f"Probabilities:\n{probabilities.cpu().numpy()}")

test_forward_pass()
print("\nModelo creado y configurado")
```

```
Dispositivo utilizado: cpu
Arquitectura del modelo:
IrisClassifier(
  (fc1): Linear(in_features=4, out_features=16, bias=True)
  (fc2): Linear(in_features=16, out_features=8, bias=True)
  (fc3): Linear(in_features=8, out_features=3, bias=True)
  (dropout): Dropout(p=0.2, inplace=False)
)

Número total de parámetros: 243
Parámetros entrenables: 243

Forma de los tensores:
X_train: torch.Size([120, 4])
X_val: torch.Size([30, 4])
y_train: torch.Size([120])
y_val: torch.Size([30])

DataLoaders creados:
Tamaño del lote: 16
Número de lotes de entrenamiento: 8
Número de lotes de validación: 2

Configuración de entrenamiento:
Función de pérdida: CrossEntropyLoss()
Optimizador: Adam (
Parameter Group 0
    amsgrad: False
    betas: (0.9, 0.999)
    capturable: False
    decoupled_weight_decay: False
    differentiable: False
    eps: 1e-08
    foreach: None
    fused: None
    lr: 0.001
    maximize: False
    weight_decay: 0.0001
)
Tasa de aprendizaje: 0.001

Prueba del forward pass:
Input shape: torch.Size([3, 4])
Output shape: torch.Size([3, 3])
Predictions: [2 2 2]
Probabilities:
[[0.25453067 0.3304598  0.4150095 ]
 [0.25737223 0.33256057 0.41006717]
 [0.26548073 0.34782875 0.38669056]]

Modelo creado y configurado
```

Task 3 - Funciones de Pérdida

Utilice diferentes funciones de pérdida comunes como Cross-Entropy Loss y MSE para clasificación. Entrene el modelo con diferentes funciones de pérdida y registre las pérdidas de entrenamiento y test. Debe utilizar al menos 3 diferentes funciones. Es decir, procure que su código sea capaz de parametrizar el uso de diferentes funciones de pérdida.

In [4]:
```
# Task 3 - Funciones de Pérdida

# Función para entrenar el modelo
def train_model(model, train_loader, val_loader, criterion, optimizer, num_epochs=1
    """
    Entrena el modelo con una función de pérdida específica
    """
    train_losses = []
    val_losses = []
    train_accuracies = []
    val_accuracies = []

    for epoch in range(num_epochs):
        # Modo entrenamiento
        model.train()
        train_loss = 0.0
        train_correct = 0
        train_total = 0

        for batch_X, batch_y in train_loader:
            # Forward pass
            outputs = model(batch_X)
            loss = criterion(outputs, batch_y)

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

            # Estadísticas
            train_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            train_total += batch_y.size(0)
            train_correct += (predicted == batch_y).sum().item()

        # Modo evaluación
        model.eval()
        val_loss = 0.0
        val_correct = 0
        val_total = 0

        with torch.no_grad():
            for batch_X, batch_y in val_loader:
                outputs = model(batch_X)
                loss = criterion(outputs, batch_y)

                val_loss += loss.item()
                _, predicted = torch.max(outputs.data, 1)
                val_total += batch_y.size(0)
```

```
                val_correct += (predicted == batch_y).sum().item()

        # Calcular promedios
        avg_train_loss = train_loss / len(train_loader)
        avg_val_loss = val_loss / len(val_loader)
        train_acc = 100 * train_correct / train_total
        val_acc = 100 * val_correct / val_total

        train_losses.append(avg_train_loss)
        val_losses.append(avg_val_loss)
        train_accuracies.append(train_acc)
        val_accuracies.append(val_acc)

        # Imprimir progreso cada 20 épocas
        if (epoch + 1) % 20 == 0:
            print(f'Época [{epoch+1}/{num_epochs}], '
                  f'Train Loss: {avg_train_loss:.4f}, Train Acc: {train_acc:.2f}%,
                  f'Val Loss: {avg_val_loss:.4f}, Val Acc: {val_acc:.2f}%')

    return train_losses, val_losses, train_accuracies, val_accuracies

# Función para crear diferentes modelos y funciones de pérdida
def create_model():
    """Crea una nueva instancia del modelo"""
    return IrisClassifier(input_size=4, hidden_size1=16, hidden_size2=8, num_classe

# Función para convertir etiquetas a one-hot para MSE
def to_one_hot(labels, num_classes=3):
    """Convierte etiquetas a formato one-hot para MSE"""
    one_hot = torch.zeros(labels.size(0), num_classes)
    one_hot.scatter_(1, labels.cpu().unsqueeze(1), 1)
    return one_hot.to(device)

# Función de pérdida personalizada MSE para clasificación
def mse_classification_loss(outputs, targets):
    """MSE adaptado para clasificación multiclase"""
    # Aplicar softmax a las salidas
    probabilities = F.softmax(outputs, dim=1)
    # Convertir targets a one-hot
    targets_one_hot = to_one_hot(targets, num_classes=3)
    # Calcular MSE
    return F.mse_loss(probabilities, targets_one_hot)

# Configurar las diferentes funciones de pérdida
loss_functions = {
    'CrossEntropyLoss': nn.CrossEntropyLoss(),
    'MSE (adaptado)': mse_classification_loss,
    'NLLLoss': nn.NLLLoss(),  # Requiere log_softmax en el modelo
}

# Modificar el modelo para NLLLoss
class IrisClassifierNLL(nn.Module):
    def __init__(self, input_size=4, hidden_size1=16, hidden_size2=8, num_classes=3
        super(IrisClassifierNLL, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size1)
        self.fc2 = nn.Linear(hidden_size1, hidden_size2)
```

```
        self.fc3 = nn.Linear(hidden_size2, num_classes)
        self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        x = self.dropout(x)
        x = self.fc3(x)
        return F.log_softmax(x, dim=1)  # log_softmax para NLLLoss

# Diccionario para almacenar resultados
results = {}

print("Iniciando entrenamiento con diferentes funciones de pérdida...")
print("=" * 60)

# Entrenar con cada función de pérdida
for loss_name, loss_fn in loss_functions.items():
    print(f"\n Entrenando con {loss_name}")
    print("-" * 40)

    # Crear modelo apropiado
    if loss_name == 'NLLLoss':
        current_model = IrisClassifierNLL().to(device)
    else:
        current_model = create_model()

    # Crear optimizador
    current_optimizer = optim.Adam(current_model.parameters(), lr=0.001, weight_dec

    # Entrenar modelo
    train_losses, val_losses, train_accs, val_accs = train_model(
        current_model, train_loader, val_loader, loss_fn, current_optimizer, num_ep
    )

    # Guardar resultados
    results[loss_name] = {
        'model': current_model,
        'train_losses': train_losses,
        'val_losses': val_losses,
        'train_accuracies': train_accs,
        'val_accuracies': val_accs,
        'final_train_acc': train_accs[-1],
        'final_val_acc': val_accs[-1]
    }

    print(f" Precisión final - Train: {train_accs[-1]:.2f}%, Val: {val_accs[-1]:.2f

print("\n" + "=" * 60)
print("Entrenamiento completado para todas las funciones de pérdida")

# Comparar resultados finales
print("\n RESUMEN DE RESULTADOS:")
print("-" * 50)
```

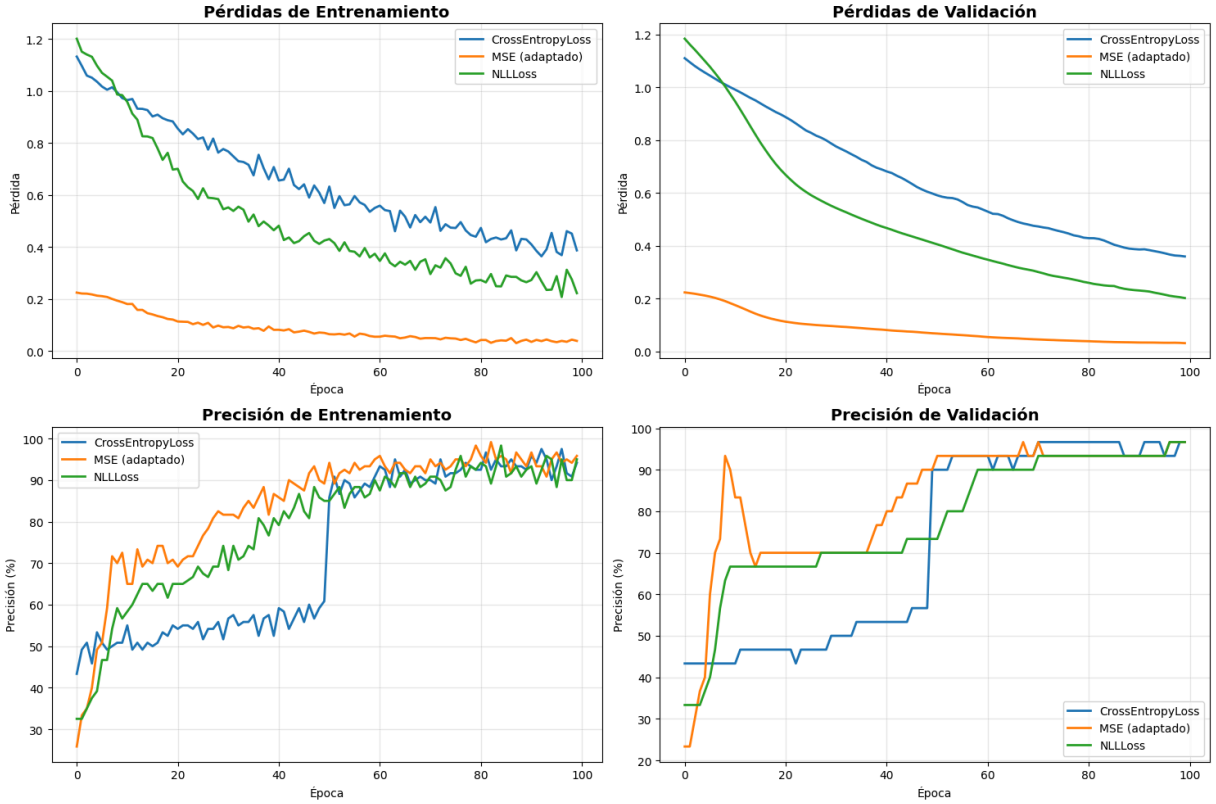
```
for loss_name, result in results.items():
    print(f"{loss_name:20s} | Train: {result['final_train_acc']:6.2f}% | Val: {resu
```

```
Iniciando entrenamiento con diferentes funciones de pérdida...
============================================================

 Entrenando con CrossEntropyLoss
----------------------------------------
Época [20/100], Train Loss: 0.8832, Train Acc: 55.00%, Val Loss: 0.8971, Val Acc: 4
6.67%
Época [40/100], Train Loss: 0.7076, Train Acc: 52.50%, Val Loss: 0.6887, Val Acc: 5
3.33%
Época [60/100], Train Loss: 0.5502, Train Acc: 90.83%, Val Loss: 0.5370, Val Acc: 9
3.33%
Época [80/100], Train Loss: 0.4394, Train Acc: 92.50%, Val Loss: 0.4305, Val Acc: 9
6.67%
Época [100/100], Train Loss: 0.3868, Train Acc: 94.17%, Val Loss: 0.3593, Val Acc: 9
6.67%
 Precisión final - Train: 94.17%, Val: 96.67%

 Entrenando con MSE (adaptado)
----------------------------------------
Época [20/100], Train Loss: 0.1204, Train Acc: 70.83%, Val Loss: 0.1159, Val Acc: 7
0.00%
Época [40/100], Train Loss: 0.0809, Train Acc: 86.67%, Val Loss: 0.0825, Val Acc: 7
6.67%
Época [60/100], Train Loss: 0.0543, Train Acc: 95.00%, Val Loss: 0.0559, Val Acc: 9
3.33%
Época [80/100], Train Loss: 0.0326, Train Acc: 98.33%, Val Loss: 0.0391, Val Acc: 9
3.33%
Época [100/100], Train Loss: 0.0383, Train Acc: 95.83%, Val Loss: 0.0313, Val Acc: 9
6.67%
 Precisión final - Train: 95.83%, Val: 96.67%

 Entrenando con NLLLoss
----------------------------------------
Época [20/100], Train Loss: 0.6978, Train Acc: 65.00%, Val Loss: 0.6876, Val Acc: 6
6.67%
Época [40/100], Train Loss: 0.4643, Train Acc: 80.83%, Val Loss: 0.4728, Val Acc: 7
0.00%
Época [60/100], Train Loss: 0.3737, Train Acc: 90.00%, Val Loss: 0.3520, Val Acc: 9
0.00%
Época [80/100], Train Loss: 0.2710, Train Acc: 92.50%, Val Loss: 0.2630, Val Acc: 9
3.33%
Época [100/100], Train Loss: 0.2221, Train Acc: 95.00%, Val Loss: 0.2019, Val Acc: 9
6.67%
 Precisión final - Train: 95.00%, Val: 96.67%

============================================================
Entrenamiento completado para todas las funciones de pérdida

 RESUMEN DE RESULTADOS:
--------------------------------------------------
CrossEntropyLoss     | Train:  94.17% | Val:  96.67%
MSE (adaptado)       | Train:  95.83% | Val:  96.67%
NLLLoss              | Train:  95.00% | Val:  96.67%
```

In [5]:
```
plt.figure(figsize=(15, 10))

# Subplot 1: Pérdidas de entrenamiento
plt.subplot(2, 2, 1)
for loss_name, result in results.items():
    plt.plot(result['train_losses'], label=f'{loss_name}', linewidth=2)
plt.title('Pérdidas de Entrenamiento', fontsize=14, fontweight='bold')
plt.xlabel('Época')
plt.ylabel('Pérdida')
plt.legend()
plt.grid(True, alpha=0.3)

# Subplot 2: Pérdidas de validación
plt.subplot(2, 2, 2)
for loss_name, result in results.items():
    plt.plot(result['val_losses'], label=f'{loss_name}', linewidth=2)
plt.title('Pérdidas de Validación', fontsize=14, fontweight='bold')
plt.xlabel('Época')
plt.ylabel('Pérdida')
plt.legend()
plt.grid(True, alpha=0.3)

# Subplot 3: Precisión de entrenamiento
plt.subplot(2, 2, 3)
for loss_name, result in results.items():
    plt.plot(result['train_accuracies'], label=f'{loss_name}', linewidth=2)
plt.title('Precisión de Entrenamiento', fontsize=14, fontweight='bold')
plt.xlabel('Época')
plt.ylabel('Precisión (%)')
plt.legend()
plt.grid(True, alpha=0.3)

# Subplot 4: Precisión de validación
plt.subplot(2, 2, 4)
for loss_name, result in results.items():
    plt.plot(result['val_accuracies'], label=f'{loss_name}', linewidth=2)
plt.title('Precisión de Validación', fontsize=14, fontweight='bold')
plt.xlabel('Época')
plt.ylabel('Precisión (%)')
plt.legend()
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Crear gráfico de barras para comparación final
plt.figure(figsize=(12, 6))

loss_names = list(results.keys())
train_accs = [results[name]['final_train_acc'] for name in loss_names]
val_accs = [results[name]['final_val_acc'] for name in loss_names]

x = range(len(loss_names))
width = 0.35
```

```
plt.subplot(1, 2, 1)
bars1 = plt.bar([i - width/2 for i in x], train_accs, width, label='Entrenamiento',
bars2 = plt.bar([i + width/2 for i in x], val_accs, width, label='Validación', alph

plt.xlabel('Función de Pérdida')
plt.ylabel('Precisión (%)')
plt.title('Comparación Final de Precisiones', fontweight='bold')
plt.xticks(x, loss_names, rotation=45)
plt.legend()
plt.grid(True, alpha=0.3)

# Agregar valores en las barras
for bar in bars1:
    height = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2., height + 0.5,
             f'{height:.1f}%', ha='center', va='bottom')

for bar in bars2:
    height = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2., height + 0.5,
             f'{height:.1f}%', ha='center', va='bottom')

# Tabla resumen
plt.subplot(1, 2, 2)
plt.axis('tight')
plt.axis('off')

table_data = []
for loss_name, result in results.items():
    table_data.append([
        loss_name,
        f"{result['final_train_acc']:.2f}%",
        f"{result['final_val_acc']:.2f}%",
        f"{result['final_val_acc'] - result['final_train_acc']:+.2f}%"
    ])

table = plt.table(cellText=table_data,
                  colLabels=['Función de Pérdida', 'Precisión Train', 'Precisión Va
                  cellLoc='center',
                  loc='center')
table.auto_set_font_size(False)
table.set_fontsize(10)
table.scale(1.2, 1.5)

plt.title('Tabla Resumen de Resultados', fontweight='bold', pad=20)

plt.tight_layout()
plt.show()

# Análisis detallado
print("\n" + "="*60)
print(" ANÁLISIS DETALLADO DE RESULTADOS")
print("="*60)

best_val_acc = max(results[name]['final_val_acc'] for name in results.keys())
best_loss_fn = [name for name, result in results.items() if result['final_val_acc']
```

```
print(f"\nMEJOR FUNCIÓN DE PÉRDIDA: {best_loss_fn}")
print(f" Precisión de validación: {best_val_acc:.2f}%")

print(f"\n CONVERGENCIA:")
for loss_name, result in results.items():
    epochs_to_90 = next((i for i, acc in enumerate(result['train_accuracies']) if a
    print(f"   {loss_name:20s}: {epochs_to_90:3d} épocas para alcanzar 90% en entre

print(f"\n OVERFITTING (Train - Val):")
for loss_name, result in results.items():
    diff = result['final_train_acc'] - result['final_val_acc']
    status = " Bueno" if diff < 5 else "  Moderado" if diff < 10 else "❌ Alto"
    print(f"   {loss_name:20s}: {diff:+.2f}% ({status})")
```



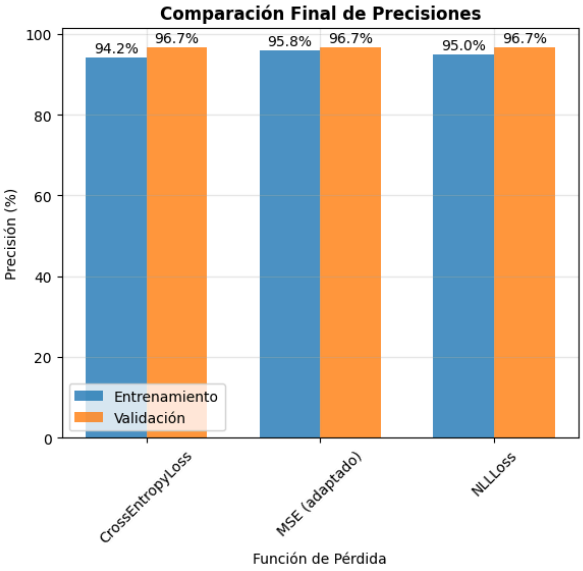


Tabla Resumen de Resultados

Función de Pérdida	Precisión Train	Precisión Val	Diferencia
CrossEntropyLoss	94.17%	96.67%	+2.50%
MSE (adaptado)	95.83%	96.67%	+0.83%
NLLLoss	95.00%	96.67%	+1.67%

```
============================================================
 ANÁLISIS DETALLADO DE RESULTADOS
============================================================

MEJOR FUNCIÓN DE PÉRDIDA: CrossEntropyLoss
 Precisión de validación: 96.67%

 CONVERGENCIA:
   CrossEntropyLoss    :  51 épocas para alcanzar 90% en entrenamiento
   MSE (adaptado)      :  42 épocas para alcanzar 90% en entrenamiento
   NLLLoss             :  59 épocas para alcanzar 90% en entrenamiento

 OVERFITTING (Train - Val):
   CrossEntropyLoss    : -2.50% ( Bueno)
   MSE (adaptado)      : -0.83% ( Bueno)
   NLLLoss             : -1.67% ( Bueno)
```

Task 4 - Técnicas de Regularización

Utilice distintas técnicas de regularización como L1, L2 y dropout. Entrene el modelo con y sin técnicas de regularización y observe el impacto en el overfitting y la generalización. Debe utilizar al menos 3 diferentes técnicas. Es decir, procure que su código sea capaz de parametrizar el uso de diferentes técnicas de regularización.

In [6]:
```python
# Task 4 - Técnicas de Regularización

# Reiniciar la semilla para experimentos reproducibles
torch.manual_seed(42)
np.random.seed(42)

# Modelos con diferentes técnicas de regularización
class IrisClassifierDropout(nn.Module):
    """Modelo con Dropout como regularización"""
    def __init__(self, input_size=4, hidden_size1=16, hidden_size2=8, num_classes=3
        super(IrisClassifierDropout, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size1)
```

```
        self.fc2 = nn.Linear(hidden_size1, hidden_size2)
        self.fc3 = nn.Linear(hidden_size2, num_classes)
        self.dropout = nn.Dropout(dropout_rate)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        x = self.dropout(x)
        x = self.fc3(x)
        return x

class IrisClassifierBatchNorm(nn.Module):
    """Modelo con Batch Normalization como regularización"""
    def __init__(self, input_size=4, hidden_size1=16, hidden_size2=8, num_classes=3
        super(IrisClassifierBatchNorm, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size1)
        self.bn1 = nn.BatchNorm1d(hidden_size1)
        self.fc2 = nn.Linear(hidden_size1, hidden_size2)
        self.bn2 = nn.BatchNorm1d(hidden_size2)
        self.fc3 = nn.Linear(hidden_size2, num_classes)

    def forward(self, x):
        x = self.bn1(F.relu(self.fc1(x)))
        x = self.bn2(F.relu(self.fc2(x)))
        x = self.fc3(x)
        return x

class IrisClassifierNoRegularization(nn.Module):
    """Modelo sin regularización (baseline)"""
    def __init__(self, input_size=4, hidden_size1=16, hidden_size2=8, num_classes=3
        super(IrisClassifierNoRegularization, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size1)
        self.fc2 = nn.Linear(hidden_size1, hidden_size2)
        self.fc3 = nn.Linear(hidden_size2, num_classes)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# Función de entrenamiento para regularización
def train_model_regularization(model, train_loader, val_loader, optimizer, num_epoc
    """
    Entrena el modelo y retorna métricas de entrenamiento
    """
    criterion = nn.CrossEntropyLoss()
    train_losses = []
    val_losses = []
    train_accuracies = []
    val_accuracies = []

    for epoch in range(num_epochs):
        # Entrenamiento
        model.train()
```

```
        train_loss = 0.0
        train_correct = 0
        train_total = 0

        for batch_X, batch_y in train_loader:
            outputs = model(batch_X)
            loss = criterion(outputs, batch_y)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            train_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            train_total += batch_y.size(0)
            train_correct += (predicted == batch_y).sum().item()

        # Validación
        model.eval()
        val_loss = 0.0
        val_correct = 0
        val_total = 0

        with torch.no_grad():
            for batch_X, batch_y in val_loader:
                outputs = model(batch_X)
                loss = criterion(outputs, batch_y)

                val_loss += loss.item()
                _, predicted = torch.max(outputs.data, 1)
                val_total += batch_y.size(0)
                val_correct += (predicted == batch_y).sum().item()

        # Calcular métricas
        avg_train_loss = train_loss / len(train_loader)
        avg_val_loss = val_loss / len(val_loader)
        train_acc = 100 * train_correct / train_total
        val_acc = 100 * val_correct / val_total

        train_losses.append(avg_train_loss)
        val_losses.append(avg_val_loss)
        train_accuracies.append(train_acc)
        val_accuracies.append(val_acc)

        if (epoch + 1) % 30 == 0:
            print(f'Época [{epoch+1}/{num_epochs}], '
                  f'Train Loss: {avg_train_loss:.4f}, Train Acc: {train_acc:.2f}%, '
                  f'Val Loss: {avg_val_loss:.4f}, Val Acc: {val_acc:.2f}%')

    return train_losses, val_losses, train_accuracies, val_accuracies

# Configuraciones de regularización
regularization_configs = {
    'Sin Regularización': {
        'model_class': IrisClassifierNoRegularization,
        'optimizer_params': {'lr': 0.001, 'weight_decay': 0.0},
```

```
        'description': 'Modelo baseline sin técnicas de regularización'
    },
    'L2 Regularization': {
        'model_class': IrisClassifierNoRegularization,
        'optimizer_params': {'lr': 0.001, 'weight_decay': 0.01},
        'description': 'Regularización L2 (weight decay) en el optimizador'
    },
    'L1 Regularization': {
        'model_class': IrisClassifierNoRegularization,
        'optimizer_params': {'lr': 0.001, 'weight_decay': 0.0},
        'l1_lambda': 0.001,
        'description': 'Regularización L1 manual en la función de pérdida'
    },
    'Dropout (0.3)': {
        'model_class': IrisClassifierDropout,
        'model_params': {'dropout_rate': 0.3},
        'optimizer_params': {'lr': 0.001, 'weight_decay': 0.0},
        'description': 'Dropout con probabilidad 0.3'
    },
    'Dropout (0.5)': {
        'model_class': IrisClassifierDropout,
        'model_params': {'dropout_rate': 0.5},
        'optimizer_params': {'lr': 0.001, 'weight_decay': 0.0},
        'description': 'Dropout con probabilidad 0.5'
    },
    'Batch Normalization': {
        'model_class': IrisClassifierBatchNorm,
        'optimizer_params': {'lr': 0.001, 'weight_decay': 0.0},
        'description': 'Normalización por lotes entre capas'
    },
    'L2 + Dropout': {
        'model_class': IrisClassifierDropout,
        'model_params': {'dropout_rate': 0.3},
        'optimizer_params': {'lr': 0.001, 'weight_decay': 0.005},
        'description': 'Combinación de L2 y Dropout'
    }
}

# Función para aplicar regularización L1
def l1_regularization(model, l1_lambda):
    """Calcula la pérdida de regularización L1"""
    l1_norm = sum(p.abs().sum() for p in model.parameters())
    return l1_lambda * l1_norm

# Función de entrenamiento con L1
def train_model_l1(model, train_loader, val_loader, optimizer, l1_lambda, num_epoch
    """Entrenamiento con regularización L1 manual"""
    criterion = nn.CrossEntropyLoss()
    train_losses = []
    val_losses = []
    train_accuracies = []
    val_accuracies = []

    for epoch in range(num_epochs):
        # Entrenamiento
        model.train()
```

```
        train_loss = 0.0
        train_correct = 0
        train_total = 0

        for batch_X, batch_y in train_loader:
            outputs = model(batch_X)
            ce_loss = criterion(outputs, batch_y)
            l1_loss = l1_regularization(model, l1_lambda)
            total_loss = ce_loss + l1_loss

            optimizer.zero_grad()
            total_loss.backward()
            optimizer.step()

            train_loss += total_loss.item()
            _, predicted = torch.max(outputs.data, 1)
            train_total += batch_y.size(0)
            train_correct += (predicted == batch_y).sum().item()

        # Validación (sin L1 para métricas limpias)
        model.eval()
        val_loss = 0.0
        val_correct = 0
        val_total = 0

        with torch.no_grad():
            for batch_X, batch_y in val_loader:
                outputs = model(batch_X)
                loss = criterion(outputs, batch_y)

                val_loss += loss.item()
                _, predicted = torch.max(outputs.data, 1)
                val_total += batch_y.size(0)
                val_correct += (predicted == batch_y).sum().item()

        # Calcular métricas
        avg_train_loss = train_loss / len(train_loader)
        avg_val_loss = val_loss / len(val_loader)
        train_acc = 100 * train_correct / train_total
        val_acc = 100 * val_correct / val_total

        train_losses.append(avg_train_loss)
        val_losses.append(avg_val_loss)
        train_accuracies.append(train_acc)
        val_accuracies.append(val_acc)

        if (epoch + 1) % 30 == 0:
            print(f'Época [{epoch+1}/{num_epochs}], '
                  f'Train Loss: {avg_train_loss:.4f}, Train Acc: {train_acc:.2f}%,
                  f'Val Loss: {avg_val_loss:.4f}, Val Acc: {val_acc:.2f}%')

    return train_losses, val_losses, train_accuracies, val_accuracies

# Almacenar resultados de regularización
regularization_results = {}
```

```
print(" Iniciando experimentos de regularización...")
print("=" * 70)

# Entrenar con cada técnica de regularización
for reg_name, config in regularization_configs.items():
    print(f"\n  Entrenando con: {reg_name}")
    print(f" Descripción: {config['description']}")
    print("-" * 50)

    # Crear modelo
    model_params = config.get('model_params', {})
    current_model = config['model_class'](**model_params).to(device)

    # Crear optimizador
    optimizer_params = config['optimizer_params']
    current_optimizer = optim.Adam(current_model.parameters(), **optimizer_params)

    # Entrenar según el tipo de regularización
    if 'l1_lambda' in config:
        # Regularización L1 manual
        train_losses, val_losses, train_accs, val_accs = train_model_l1(
            current_model, train_loader, val_loader, current_optimizer,
            config['l1_lambda'], num_epochs=150
        )
    else:
        # Regularización estándar
        train_losses, val_losses, train_accs, val_accs = train_model_regularization
            current_model, train_loader, val_loader, current_optimizer, num_epochs=
        )

    # Guardar resultados
    regularization_results[reg_name] = {
        'model': current_model,
        'train_losses': train_losses,
        'val_losses': val_losses,
        'train_accuracies': train_accs,
        'val_accuracies': val_accs,
        'final_train_acc': train_accs[-1],
        'final_val_acc': val_accs[-1],
        'overfitting': train_accs[-1] - val_accs[-1],
        'config': config
    }

    print(f" Resultado final:")
    print(f"   Train: {train_accs[-1]:.2f}% | Val: {val_accs[-1]:.2f}% | Overfittin

print("\n" + "=" * 70)
print(" Experimentos de regularización completados!")

# Resumen de resultados
print("\n RESUMEN DE TÉCNICAS DE REGULARIZACIÓN:")
print("-" * 70)
print(f"{'Técnica':<20} {'Train':<8} {'Val':<8} {'Overfitting':<12} {'Descripción'}
print("-" * 70)

for reg_name, result in regularization_results.items():
```

```
    print(f"{reg_name:<20} {result['final_train_acc']:>6.2f}% {result['final_val_ac
          f"{result['overfitting']:>+8.2f}% {result['config']['description'][:30]}"

# Encontrar la mejor técnica para generalización
best_generalization = min(regularization_results.items(), key=lambda x: x[1]['overf
best_accuracy = max(regularization_results.items(), key=lambda x: x[1]['final_val_a

print(f"\n MEJORES RESULTADOS:")
print(f"   Mejor generalización: {best_generalization[0]} (overfitting: {best_gener
print(f"   Mejor precisión val:  {best_accuracy[0]} (precisión: {best_accuracy[1]['
```

```
 Iniciando experimentos de regularización...
======================================================================

  Entrenando con: Sin Regularización
 Descripción: Modelo baseline sin técnicas de regularización
--------------------------------------------------
Época [30/150], Train Loss: 0.3048, Train Acc: 90.83%, Val Loss: 0.3526, Val Acc: 8
0.00%
Época [60/150], Train Loss: 0.1230, Train Acc: 95.83%, Val Loss: 0.1775, Val Acc: 9
3.33%
Época [90/150], Train Loss: 0.0684, Train Acc: 97.50%, Val Loss: 0.1196, Val Acc: 9
3.33%
Época [120/150], Train Loss: 0.0512, Train Acc: 98.33%, Val Loss: 0.0976, Val Acc: 9
3.33%
Época [150/150], Train Loss: 0.0447, Train Acc: 98.33%, Val Loss: 0.0895, Val Acc: 9
6.67%
 Resultado final:
   Train: 98.33% | Val: 96.67% | Overfitting: +1.67%

  Entrenando con: L2 Regularization
 Descripción: Regularización L2 (weight decay) en el optimizador
--------------------------------------------------
Época [30/150], Train Loss: 0.3133, Train Acc: 91.67%, Val Loss: 0.3637, Val Acc: 8
6.67%
Época [60/150], Train Loss: 0.1395, Train Acc: 95.83%, Val Loss: 0.1892, Val Acc: 9
3.33%
Época [90/150], Train Loss: 0.0991, Train Acc: 97.50%, Val Loss: 0.1360, Val Acc: 9
3.33%
Época [120/150], Train Loss: 0.0828, Train Acc: 97.50%, Val Loss: 0.1175, Val Acc: 9
3.33%
Época [150/150], Train Loss: 0.0737, Train Acc: 97.50%, Val Loss: 0.1089, Val Acc: 9
6.67%
 Resultado final:
   Train: 97.50% | Val: 96.67% | Overfitting: +0.83%

  Entrenando con: L1 Regularization
 Descripción: Regularización L1 manual en la función de pérdida
--------------------------------------------------
Época [30/150], Train Loss: 0.3543, Train Acc: 90.83%, Val Loss: 0.3486, Val Acc: 8
6.67%
Época [60/150], Train Loss: 0.1622, Train Acc: 96.67%, Val Loss: 0.1474, Val Acc: 9
6.67%
Época [90/150], Train Loss: 0.1329, Train Acc: 97.50%, Val Loss: 0.1015, Val Acc: 9
6.67%
Época [120/150], Train Loss: 0.1210, Train Acc: 97.50%, Val Loss: 0.0885, Val Acc: 9
6.67%
Época [150/150], Train Loss: 0.1255, Train Acc: 97.50%, Val Loss: 0.0783, Val Acc: 9
6.67%
 Resultado final:
   Train: 97.50% | Val: 96.67% | Overfitting: +0.83%

  Entrenando con: Dropout (0.3)
 Descripción: Dropout con probabilidad 0.3
--------------------------------------------------
Época [30/150], Train Loss: 0.6631, Train Acc: 68.33%, Val Loss: 0.5971, Val Acc: 8
0.00%
```

```
Época [60/150], Train Loss: 0.5071, Train Acc: 73.33%, Val Loss: 0.3966, Val Acc: 8
0.00%
Época [90/150], Train Loss: 0.3943, Train Acc: 83.33%, Val Loss: 0.2981, Val Acc: 9
0.00%
Época [120/150], Train Loss: 0.3047, Train Acc: 89.17%, Val Loss: 0.1897, Val Acc: 9
6.67%
Época [150/150], Train Loss: 0.2359, Train Acc: 90.00%, Val Loss: 0.1541, Val Acc: 9
3.33%
 Resultado final:
   Train: 90.00% | Val: 93.33% | Overfitting: -3.33%

  Entrenando con: Dropout (0.5)
 Descripción: Dropout con probabilidad 0.5
--------------------------------------------------
Época [30/150], Train Loss: 0.7873, Train Acc: 62.50%, Val Loss: 0.6180, Val Acc: 6
6.67%
Época [60/150], Train Loss: 0.6239, Train Acc: 66.67%, Val Loss: 0.3990, Val Acc: 8
0.00%
Época [90/150], Train Loss: 0.4192, Train Acc: 83.33%, Val Loss: 0.2610, Val Acc: 9
6.67%
Época [120/150], Train Loss: 0.3298, Train Acc: 85.83%, Val Loss: 0.1911, Val Acc: 9
6.67%
Época [150/150], Train Loss: 0.4307, Train Acc: 80.00%, Val Loss: 0.1529, Val Acc: 9
6.67%
 Resultado final:
   Train: 80.00% | Val: 96.67% | Overfitting: -16.67%

  Entrenando con: Batch Normalization
 Descripción: Normalización por lotes entre capas
--------------------------------------------------
Época [30/150], Train Loss: 0.2490, Train Acc: 99.17%, Val Loss: 0.2672, Val Acc: 9
3.33%
Época [60/150], Train Loss: 0.1348, Train Acc: 96.67%, Val Loss: 0.1384, Val Acc: 10
0.00%
Época [90/150], Train Loss: 0.1612, Train Acc: 95.00%, Val Loss: 0.1189, Val Acc: 10
0.00%
Época [120/150], Train Loss: 0.0820, Train Acc: 97.50%, Val Loss: 0.1023, Val Acc: 9
6.67%
Época [150/150], Train Loss: 0.0538, Train Acc: 98.33%, Val Loss: 0.1006, Val Acc: 9
6.67%
 Resultado final:
   Train: 98.33% | Val: 96.67% | Overfitting: +1.67%

  Entrenando con: L2 + Dropout
 Descripción: Combinación de L2 y Dropout
--------------------------------------------------
Época [30/150], Train Loss: 0.6147, Train Acc: 63.33%, Val Loss: 0.5259, Val Acc: 7
3.33%
Época [60/150], Train Loss: 0.3624, Train Acc: 85.83%, Val Loss: 0.3118, Val Acc: 9
3.33%
Época [90/150], Train Loss: 0.2962, Train Acc: 87.50%, Val Loss: 0.2146, Val Acc: 9
3.33%
Época [120/150], Train Loss: 0.2479, Train Acc: 91.67%, Val Loss: 0.1565, Val Acc: 9
6.67%
Época [150/150], Train Loss: 0.2174, Train Acc: 91.67%, Val Loss: 0.1290, Val Acc: 9
6.67%
```

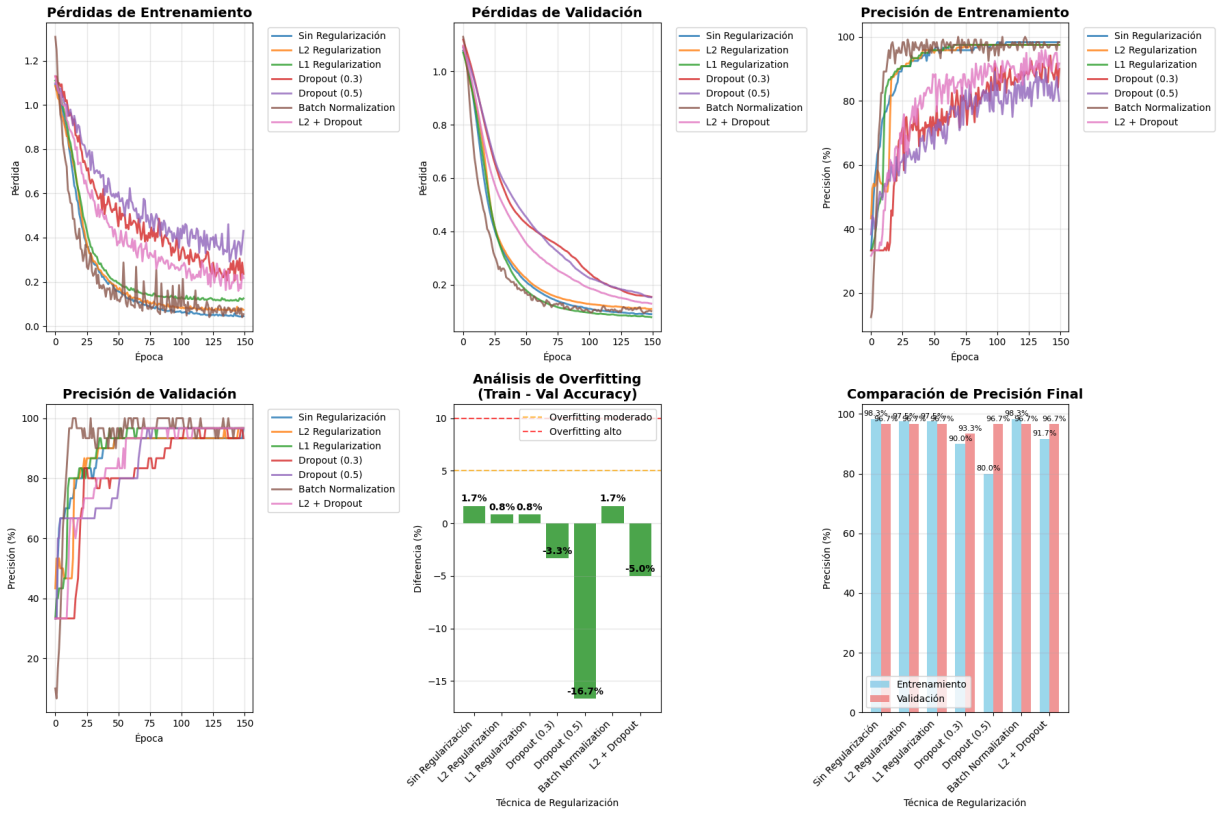
```
 Resultado final:
   Train: 91.67% | Val: 96.67% | Overfitting: -5.00%

======================================================================
 Experimentos de regularización completados!

 RESUMEN DE TÉCNICAS DE REGULARIZACIÓN:
----------------------------------------------------------------------
Técnica              Train    Val      Overfitting  Descripción
----------------------------------------------------------------------
Sin Regularización    98.33%  96.67%    +1.67% Modelo baseline sin técnicas d
L2 Regularization     97.50%  96.67%    +0.83% Regularización L2 (weight deca
L1 Regularization     97.50%  96.67%    +0.83% Regularización L1 manual en la
Dropout (0.3)         90.00%  93.33%    -3.33% Dropout con probabilidad 0.3
Dropout (0.5)         80.00%  96.67%   -16.67% Dropout con probabilidad 0.5
Batch Normalization   98.33%  96.67%    +1.67% Normalización por lotes entre
L2 + Dropout          91.67%  96.67%    -5.00% Combinación de L2 y Dropout

 MEJORES RESULTADOS:
   Mejor generalización: Dropout (0.5) (overfitting: -16.67%)
   Mejor precisión val:  Sin Regularización (precisión: 96.67%)
```

In [7]:

```python
plt.figure(figsize=(18, 12))

# Subplot 1: Evolución de pérdidas de entrenamiento
plt.subplot(2, 3, 1)
for reg_name, result in regularization_results.items():
    plt.plot(result['train_losses'], label=reg_name, linewidth=2, alpha=0.8)
plt.title('Pérdidas de Entrenamiento', fontsize=14, fontweight='bold')
plt.xlabel('Época')
plt.ylabel('Pérdida')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.grid(True, alpha=0.3)

# Subplot 2: Evolución de pérdidas de validación
plt.subplot(2, 3, 2)
for reg_name, result in regularization_results.items():
    plt.plot(result['val_losses'], label=reg_name, linewidth=2, alpha=0.8)
plt.title('Pérdidas de Validación', fontsize=14, fontweight='bold')
plt.xlabel('Época')
plt.ylabel('Pérdida')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.grid(True, alpha=0.3)

# Subplot 3: Evolución de precisión de entrenamiento
plt.subplot(2, 3, 3)
for reg_name, result in regularization_results.items():
    plt.plot(result['train_accuracies'], label=reg_name, linewidth=2, alpha=0.8)
plt.title('Precisión de Entrenamiento', fontsize=14, fontweight='bold')
plt.xlabel('Época')
plt.ylabel('Precisión (%)')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.grid(True, alpha=0.3)

# Subplot 4: Evolución de precisión de validación
plt.subplot(2, 3, 4)
```

```
for reg_name, result in regularization_results.items():
    plt.plot(result['val_accuracies'], label=reg_name, linewidth=2, alpha=0.8)
plt.title('Precisión de Validación', fontsize=14, fontweight='bold')
plt.xlabel('Época')
plt.ylabel('Precisión (%)')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.grid(True, alpha=0.3)

# Subplot 5: Análisis de overfitting
plt.subplot(2, 3, 5)
reg_names = list(regularization_results.keys())
overfitting_scores = [regularization_results[name]['overfitting'] for name in reg_n
colors = ['red' if score > 10 else 'orange' if score > 5 else 'green' for score in

bars = plt.bar(range(len(reg_names)), overfitting_scores, color=colors, alpha=0.7)
plt.title('Análisis de Overfitting\n(Train - Val Accuracy)', fontsize=14, fontweigh
plt.xlabel('Técnica de Regularización')
plt.ylabel('Diferencia (%)')
plt.xticks(range(len(reg_names)), reg_names, rotation=45, ha='right')
plt.grid(True, alpha=0.3, axis='y')

# Agregar líneas de referencia
plt.axhline(y=5, color='orange', linestyle='--', alpha=0.7, label='Overfitting mode
plt.axhline(y=10, color='red', linestyle='--', alpha=0.7, label='Overfitting alto')
plt.legend()

# Agregar valores en las barras
for bar, score in zip(bars, overfitting_scores):
    height = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2., height + 0.2,
             f'{score:.1f}%', ha='center', va='bottom', fontweight='bold')

# Subplot 6: Comparación de precisión final
plt.subplot(2, 3, 6)
train_accs_reg = [regularization_results[name]['final_train_acc'] for name in reg_n
val_accs_reg = [regularization_results[name]['final_val_acc'] for name in reg_names

x_pos = range(len(reg_names))
width = 0.35

bars1 = plt.bar([i - width/2 for i in x_pos], train_accs_reg, width,
                label='Entrenamiento', alpha=0.8, color='skyblue')
bars2 = plt.bar([i + width/2 for i in x_pos], val_accs_reg, width,
                label='Validación', alpha=0.8, color='lightcoral')

plt.title('Comparación de Precisión Final', fontsize=14, fontweight='bold')
plt.xlabel('Técnica de Regularización')
plt.ylabel('Precisión (%)')
plt.xticks(x_pos, reg_names, rotation=45, ha='right')
plt.legend()
plt.grid(True, alpha=0.3, axis='y')

# Agregar valores en las barras
for bar in bars1:
    height = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2., height + 0.5,
```

```
            f'{height:.1f}%', ha='center', va='bottom', fontsize=8)

for bar in bars2:
    height = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2., height + 0.5,
             f'{height:.1f}%', ha='center', va='bottom', fontsize=8)

plt.tight_layout()
plt.show()

convergence_analysis = {}
for reg_name, result in regularization_results.items():
    epochs_to_95 = next((i for i, acc in enumerate(result['train_accuracies']) if a
                         len(result['train_accuracies']))
    convergence_analysis[reg_name] = epochs_to_95


stability_analysis = {}
for reg_name, result in regularization_results.items():
    last_30_val_acc = result['val_accuracies'][-30:]
    stability = np.std(last_30_val_acc)
    stability_analysis[reg_name] = stability

# Crear tabla comparativa final
print(f"\n TABLA COMPARATIVA COMPLETA:")
print(f"-" * 100)
print(f"{'Técnica':<20} {'Train%':<8} {'Val%':<8} {'Overfitting':<12} {'Convergenci
print(f"-" * 100)

for reg_name in reg_names:
    result = regularization_results[reg_name]
    conv_epochs = convergence_analysis[reg_name]
    stability = stability_analysis[reg_name]

    print(f"{reg_name:<20} {result['final_train_acc']:>6.2f}% {result['final_val_ac
          f"{result['overfitting']:>+8.2f}% {conv_epochs:>8d} épocas {stability:>6.

print(f"-" * 100)
```




```
TABLA COMPARATIVA COMPLETA:
--------------------------------------------------------------------------------
----------------
Técnica              Train%   Val%     Overfitting  Convergencia Estabilidad
--------------------------------------------------------------------------------
----------------
Sin Regularización    98.33%  96.67%    +1.67%       43 épocas   0.60%
L2 Regularization     97.50%  96.67%    +0.83%       41 épocas   1.33%
L1 Regularization     97.50%  96.67%    +0.83%       40 épocas   0.00%
Dropout (0.3)         90.00%  93.33%    -3.33%      150 épocas   1.61%
Dropout (0.5)         80.00%  96.67%   -16.67%      150 épocas   0.00%
Batch Normalization   98.33%  96.67%    +1.67%       14 épocas   1.36%
L2 + Dropout          91.67%  96.67%    -5.00%      123 épocas   0.00%
--------------------------------------------------------------------------------
----------------
```

Task 5 - Algoritmos de Optimización

Utilice distintas técnicas de optimización como SGD, Batch GD, Mini-Batch GD. Entrene el modelo con algoritmos de optimización y registre las pérdidas y tiempos de entrenamiento y test. Debe utilizar al menos 3 diferentes algoritmos. Es decir, procure que su código sea capaz de parametrizar el uso de diferentes algoritmos de optimización

In [8]:
```
# Task 5 - Algoritmos de Optimización


# Modelo base para todos los experimentos
class SimpleIrisClassifier(nn.Module):
    def __init__(self):
        super(SimpleIrisClassifier, self).__init__()
```

```
        self.fc1 = nn.Linear(4, 16)
        self.fc2 = nn.Linear(16, 8)
        self.fc3 = nn.Linear(8, 3)
        self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        x = self.dropout(x)
        x = self.fc3(x)
        return x

# Función de entrenamiento con medición de tiempo
def train_with_optimizer(model, train_loader, val_loader, optimizer, num_epochs=100
    criterion = nn.CrossEntropyLoss()
    train_losses = []
    val_losses = []
    train_accuracies = []
    val_accuracies = []

    start_time = time.time()

    for epoch in range(num_epochs):
        # Entrenamiento
        model.train()
        train_loss = 0.0
        train_correct = 0
        train_total = 0

        for batch_X, batch_y in train_loader:
            outputs = model(batch_X)
            loss = criterion(outputs, batch_y)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            train_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            train_total += batch_y.size(0)
            train_correct += (predicted == batch_y).sum().item()

        # Validación
        model.eval()
        val_loss = 0.0
        val_correct = 0
        val_total = 0

        with torch.no_grad():
            for batch_X, batch_y in val_loader:
                outputs = model(batch_X)
                loss = criterion(outputs, batch_y)

                val_loss += loss.item()
                _, predicted = torch.max(outputs.data, 1)
```

```
                val_total += batch_y.size(0)
                val_correct += (predicted == batch_y).sum().item()

        # Calcular métricas
        avg_train_loss = train_loss / len(train_loader)
        avg_val_loss = val_loss / len(val_loader)
        train_acc = 100 * train_correct / train_total
        val_acc = 100 * val_correct / val_total

        train_losses.append(avg_train_loss)
        val_losses.append(avg_val_loss)
        train_accuracies.append(train_acc)
        val_accuracies.append(val_acc)

    training_time = time.time() - start_time

    return train_losses, val_losses, train_accuracies, val_accuracies, training_tim

# Configuraciones de optimizadores
optimizers_config = {
    'SGD': {'class': optim.SGD, 'params': {'lr': 0.01, 'momentum': 0.9}},
    'SGD + Nesterov': {'class': optim.SGD, 'params': {'lr': 0.01, 'momentum': 0.9,
    'Adam': {'class': optim.Adam, 'params': {'lr': 0.001}},
    'AdamW': {'class': optim.AdamW, 'params': {'lr': 0.001, 'weight_decay': 0.01}},
    'RMSprop': {'class': optim.RMSprop, 'params': {'lr': 0.001}},
    'Adagrad': {'class': optim.Adagrad, 'params': {'lr': 0.01}},
}

# Almacenar resultados
optimizer_results = {}

print("Iniciando comparación de algoritmos de optimización...")
print("=" * 60)

# Entrenar con cada optimizador
for opt_name, config in optimizers_config.items():
    print(f"\nEntrenando con: {opt_name}")

    # Crear modelo fresco
    model = SimpleIrisClassifier().to(device)

    # Crear optimizador
    optimizer = config['class'](model.parameters(), **config['params'])

    # Entrenar
    train_losses, val_losses, train_accs, val_accs, training_time = train_with_opti
        model, train_loader, val_loader, optimizer, num_epochs=100
    )

    # Guardar resultados
    optimizer_results[opt_name] = {
        'train_losses': train_losses,
        'val_losses': val_losses,
        'train_accuracies': train_accs,
        'val_accuracies': val_accs,
        'final_train_acc': train_accs[-1],
```

```
        'final_val_acc': val_accs[-1],
        'training_time': training_time,
        'convergence_epoch': next((i for i, acc in enumerate(train_accs) if acc >=
    }

    print(f"Completado - Train: {train_accs[-1]:.2f}%, Val: {val_accs[-1]:.2f}%, Ti

print("\nComparación completada!")

# Resultados tabulados
print("\n" + "=" * 80)
print("COMPARACIÓN DE ALGORITMOS DE OPTIMIZACIÓN")
print("=" * 80)
print(f"{'Algoritmo':<15} {'Train%':<8} {'Val%':<8} {'Tiempo(s)':<10} {'Convergenci
print("-" * 80)

for opt_name, result in optimizer_results.items():
    print(f"{opt_name:<15} {result['final_train_acc']:>6.2f}% "
          f"{result['final_val_acc']:>6.2f}% {result['training_time']:>8.1f} "
          f"{result['convergence_epoch']:>8d} épocas")

# Análisis de resultados
best_accuracy = max(optimizer_results.items(), key=lambda x: x[1]['final_val_acc'])
fastest_training = min(optimizer_results.items(), key=lambda x: x[1]['training_time
fastest_convergence = min(optimizer_results.items(), key=lambda x: x[1]['convergenc

print(f"\nRESULTADOS CLAVE:")
print(f"Mejor precisión: {best_accuracy[0]} ({best_accuracy[1]['final_val_acc']:.2f
print(f"Entrenamiento más rápido: {fastest_training[0]} ({fastest_training[1]['trai
print(f"Convergencia más rápida: {fastest_convergence[0]} ({fastest_convergence[1][
```

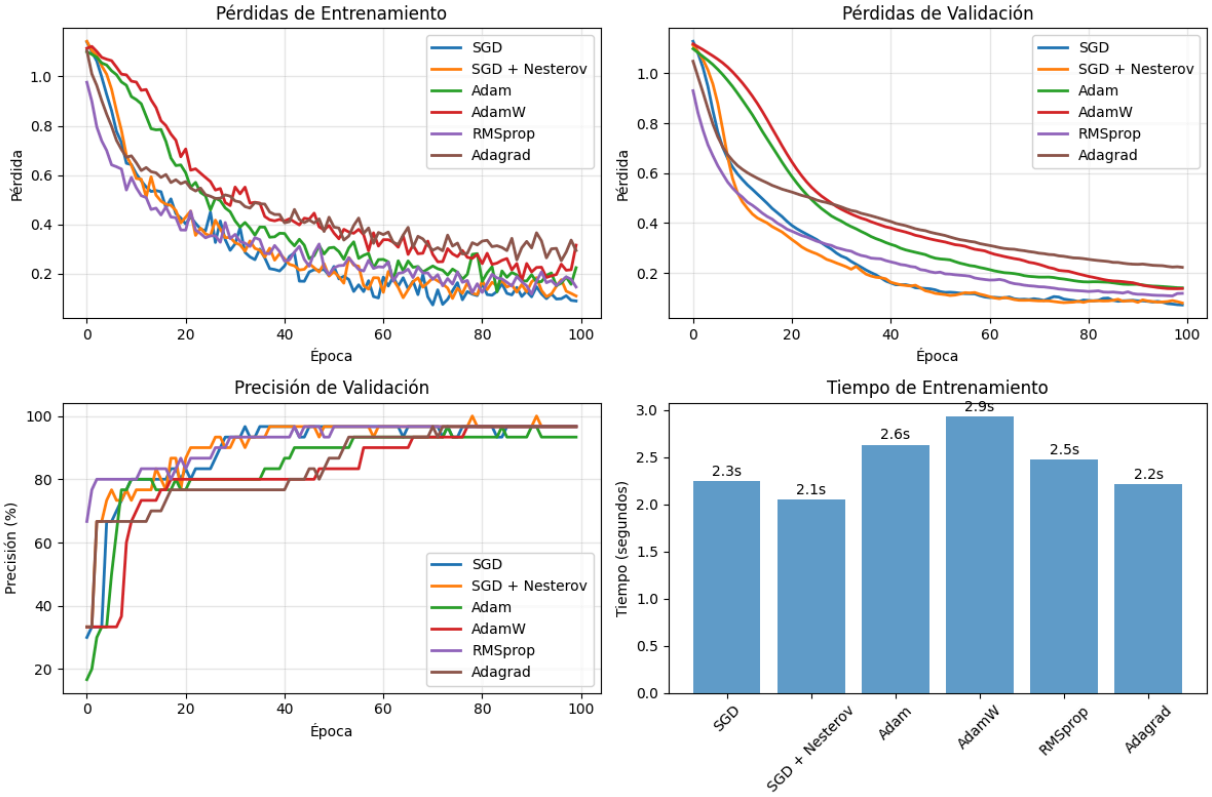
```
Iniciando comparación de algoritmos de optimización...
============================================================

Entrenando con: SGD
Completado - Train: 97.50%, Val: 96.67%, Tiempo: 2.3s

Entrenando con: SGD + Nesterov
Completado - Train: 96.67%, Val: 96.67%, Tiempo: 2.1s

Entrenando con: Adam
Completado - Train: 95.00%, Val: 93.33%, Tiempo: 2.6s

Entrenando con: AdamW
Completado - Train: 87.50%, Val: 96.67%, Tiempo: 2.9s

Entrenando con: RMSprop
Completado - Train: 94.17%, Val: 96.67%, Tiempo: 2.5s

Entrenando con: Adagrad
Completado - Train: 95.83%, Val: 96.67%, Tiempo: 2.2s

Comparación completada!

================================================================================
COMPARACIÓN DE ALGORITMOS DE OPTIMIZACIÓN
================================================================================
Algoritmo       Train%   Val%     Tiempo(s)  Convergencia
--------------------------------------------------------------------------------
SGD              97.50%  96.67%      2.3       37 épocas
SGD + Nesterov   96.67%  96.67%      2.1       36 épocas
Adam             95.00%  93.33%      2.6       38 épocas
AdamW            87.50%  96.67%      2.9       71 épocas
RMSprop          94.17%  96.67%      2.5       27 épocas
Adagrad          95.83%  96.67%      2.2       44 épocas

RESULTADOS CLAVE:
Mejor precisión: SGD (96.67%)
Entrenamiento más rápido: SGD + Nesterov (2.1s)
Convergencia más rápida: RMSprop (27 épocas)
```

In [9]:

```python
# Visualización comparativa de optimizadores
fig, axes = plt.subplots(2, 2, figsize=(12, 8))

# Pérdidas de entrenamiento
axes[0, 0].set_title('Pérdidas de Entrenamiento')
for opt_name, result in optimizer_results.items():
    axes[0, 0].plot(result['train_losses'], label=opt_name, linewidth=2)
axes[0, 0].set_xlabel('Época')
axes[0, 0].set_ylabel('Pérdida')
axes[0, 0].legend()
axes[0, 0].grid(True, alpha=0.3)

# Pérdidas de validación
axes[0, 1].set_title('Pérdidas de Validación')
for opt_name, result in optimizer_results.items():
    axes[0, 1].plot(result['val_losses'], label=opt_name, linewidth=2)
```

```
axes[0, 1].set_xlabel('Época')
axes[0, 1].set_ylabel('Pérdida')
axes[0, 1].legend()
axes[0, 1].grid(True, alpha=0.3)

# Precisión de validación
axes[1, 0].set_title('Precisión de Validación')
for opt_name, result in optimizer_results.items():
    axes[1, 0].plot(result['val_accuracies'], label=opt_name, linewidth=2)
axes[1, 0].set_xlabel('Época')
axes[1, 0].set_ylabel('Precisión (%)')
axes[1, 0].legend()
axes[1, 0].grid(True, alpha=0.3)

# Comparación de tiempos de entrenamiento
axes[1, 1].set_title('Tiempo de Entrenamiento')
opt_names = list(optimizer_results.keys())
training_times = [optimizer_results[name]['training_time'] for name in opt_names]
bars = axes[1, 1].bar(opt_names, training_times, alpha=0.7)
axes[1, 1].set_ylabel('Tiempo (segundos)')
axes[1, 1].tick_params(axis='x', rotation=45)

# Agregar valores en las barras
for bar, time_val in zip(bars, training_times):
    height = bar.get_height()
    axes[1, 1].text(bar.get_x() + bar.get_width()/2., height + height*0.01,
                    f'{time_val:.1f}s', ha='center', va='bottom')

plt.tight_layout()
plt.show()

# Tabla comparativa final
print("\nTABLA RESUMEN FINAL:")
print("-" * 70)
print(f"{'Optimizador':<15} {'Precisión Val':<12} {'Tiempo':<10} {'Convergencia':<1
print("-" * 70)

# Ordenar por precisión de validación
sorted_optimizers = sorted(optimizer_results.items(),
                           key=lambda x: x[1]['final_val_acc'], reverse=True)

for opt_name, result in sorted_optimizers:
    print(f"{opt_name:<15} {result['final_val_acc']:>9.2f}% "
          f"{result['training_time']:>7.1f}s {result['convergence_epoch']:>9d} époc

print("-" * 70)

# Análisis eficiencia (precisión/tiempo)
print("\nEFICIENCIA (Precisión/Tiempo):")
efficiency_scores = {}
for opt_name, result in optimizer_results.items():
    efficiency = result['final_val_acc'] / result['training_time']
    efficiency_scores[opt_name] = efficiency

best_efficiency = max(efficiency_scores.items(), key=lambda x: x[1])
print(f"Más eficiente: {best_efficiency[0]} ({best_efficiency[1]:.2f} puntos/segund
```

```
print(f"\nRECOMENDACIONES:")
print(f"Para máxima precisión: {best_accuracy[0]}")
print(f"Para entrenamiento rápido: {fastest_training[0]}")
print(f"Para convergencia rápida: {fastest_convergence[0]}")
print(f"Para mejor eficiencia: {best_efficiency[0]}")
```




```
TABLA RESUMEN FINAL:
----------------------------------------------------------------------
Optimizador     Precisión Val Tiempo     Convergencia
----------------------------------------------------------------------
SGD                 96.67%     2.3s        37 épocas
SGD + Nesterov      96.67%     2.1s        36 épocas
AdamW               96.67%     2.9s        71 épocas
RMSprop             96.67%     2.5s        27 épocas
Adagrad             96.67%     2.2s        44 épocas
Adam                93.33%     2.6s        38 épocas
----------------------------------------------------------------------

EFICIENCIA (Precisión/Tiempo):
Más eficiente: SGD + Nesterov (47.12 puntos/segundo)

RECOMENDACIONES:
Para máxima precisión: SGD
Para entrenamiento rápido: SGD + Nesterov
Para convergencia rápida: RMSprop
Para mejor eficiencia: SGD + Nesterov
```

Task 6 - Experimentación y Análisis

Entrene los modelos con diferentes combinaciones de funciones de pérdida, técnicas de regularización y algoritmos de optimización. Para no complicar esta parte, puede dejar fijo dos de estos parámetros (función de pérdida, técnicas de regularización, algoritmo de optimización) y solamente cambiar uno de ellos. Deben verse al menos 9 combinaciones en total, donde es válido que en una de ellas no haya ninguna técnica de regularización. Si quiere experimentar con más combinaciones se le dará hasta 10% de puntos extra. Para cada combinación registre métricas como precisión, pérdida y alguna otra métrica que considere pertinente (Recuerde lo visto en inteligencia artificial). Visualice las curvas (tanto en precisión, pérdida y la tercera métrica que decidió) de entrenamiento y validación utilizando bibliotecas como matplotlib y/o seaborn. Además, recuerde llevar tracking de los tiempos de ejecución de cada combinación.

In [12]:
```
if not all(name in globals() for name in ["train_loader", "val_loader"]):
    try:
        from sklearn.datasets import load_iris
        from sklearn.model_selection import train_test_split
        from sklearn.preprocessing import StandardScaler
    except Exception:
        pass

    iris = load_iris()
    X = iris.data.astype(np.float32)
    y = iris.target.astype(np.int64)

    scaler = StandardScaler()
    X = scaler.fit_transform(X).astype(np.float32)

    from torch.utils.data import TensorDataset, DataLoader
    X_train, X_val, y_train, y_val = train_test_split(
        X, y, test_size=0.2, random_state=42, stratify=y
    )
    X_train_t = torch.tensor(X_train, dtype=torch.float32)
    y_train_t = torch.tensor(y_train, dtype=torch.long)
    X_val_t   = torch.tensor(X_val,   dtype=torch.float32)
    y_val_t   = torch.tensor(y_val,   dtype=torch.long)

    train_ds = TensorDataset(X_train_t, y_train_t)
    val_ds   = TensorDataset(X_val_t,   y_val_t)

    train_loader = DataLoader(train_ds, batch_size=32, shuffle=True)
    val_loader   = DataLoader(val_ds,   batch_size=64, shuffle=False)

# Función para obtener input_dim y num_classes desde los loaders
def infer_dims(_loader):
    xb, yb = next(iter(_loader))
    in_dim = xb.shape[1]
    try:
        n_classes = int(torch.max(yb).item()) + 1
    except Exception:
        n_classes = len(torch.unique(yb))
    return in_dim, n_classes
```

```
in_dim, n_classes = infer_dims(train_loader)

# Resolver la clase de modelo
def _resolve_model_class():
    for name in ["Net", "MLP", "IrisClassifier", "Model", "Classifier"]:
        if name in globals() and isinstance(globals()[name], type):
            return globals()[name]
    # Fallback: MLP simple compatible
    class _FallbackMLP(nn.Module):
        def __init__(self, in_dim, hidden=64, out_dim=3, p=0.0):
            super().__init__()
            self.fc1 = nn.Linear(in_dim, hidden)
            self.fc2 = nn.Linear(hidden, hidden)
            self.fc3 = nn.Linear(hidden, out_dim)
            self.dp  = nn.Dropout(p)
            nn.init.kaiming_uniform_(self.fc1.weight, nonlinearity='relu')
            nn.init.zeros_(self.fc1.bias)
            nn.init.kaiming_uniform_(self.fc2.weight, nonlinearity='relu')
            nn.init.zeros_(self.fc2.bias)
            nn.init.xavier_uniform_(self.fc3.weight)
            nn.init.zeros_(self.fc3.bias)

        def forward(self, x):
            x = F.relu(self.fc1(x))
            x = self.dp(x)
            x = F.relu(self.fc2(x))
            x = self.dp(x)
            return self.fc3(x)
    return _FallbackMLP

ModelClass = _resolve_model_class()

# Funciones de entrenamiento/evaluación
def _train_epoch_task6(model, loader, criterion, optimizer, device):
    model.train()
    total, correct, loss_sum = 0, 0, 0.0
    for xb, yb in loader:
        xb, yb = xb.to(device), yb.to(device)
        optimizer.zero_grad(set_to_none=True)
        logits = model(xb)
        loss = criterion(logits, yb)
        loss.backward()
        optimizer.step()
        loss_sum += loss.item() * xb.size(0)
        pred = logits.argmax(1)
        correct += (pred == yb).sum().item()
        total += xb.size(0)
    acc = 100.0 * correct / max(total, 1)
    return loss_sum / max(total, 1), acc

@torch.no_grad()
def _eval_epoch_task6(model, loader, criterion, device):
    model.eval()
    total, correct, loss_sum = 0, 0, 0.0
    all_pred, all_true = [], []
    for xb, yb in loader:
```

```
        xb, yb = xb.to(device), yb.to(device)
        logits = model(xb)
        loss = criterion(logits, yb)
        loss_sum += loss.item() * xb.size(0)
        pred = logits.argmax(1)
        correct += (pred == yb).sum().item()
        total += xb.size(0)
        all_pred.append(pred.cpu())
        all_true.append(yb.cpu())
    all_pred = torch.cat(all_pred) if all_pred else torch.tensor([])
    all_true = torch.cat(all_true) if all_true else torch.tensor([])
    acc = 100.0 * correct / max(total, 1)
    f1m = f1_score(all_true.numpy(), all_pred.numpy(), average="macro") if total >
    return loss_sum / max(total, 1), acc, f1m
```

In [13]:
```
criterion_task6 = nn.CrossEntropyLoss()

# Cada item: (nombre, constructor_optim, kwargs)
combos_task6 = [
    ("SGD_lr0.1",           optim.SGD,    {"lr": 0.1}),
    ("SGD_mom0.9",          optim.SGD,    {"lr": 0.05, "momentum": 0.9}),
    ("SGD_nesterov",        optim.SGD,    {"lr": 0.05, "momentum": 0.9, "nesterov"
    ("Adam_1e-3",           optim.Adam,   {"lr": 1e-3}),
    ("Adam_5e-4",           optim.Adam,   {"lr": 5e-4}),
    ("RMSprop_1e-3",        optim.RMSprop,{"lr": 1e-3, "alpha": 0.9}),
    ("Adagrad_1e-2",        optim.Adagrad,{"lr": 1e-2}),
    ("Adamax_2e-3",         optim.Adamax, {"lr": 2e-3}),
    ("AdamW_wd1e-2",        optim.AdamW,  {"lr": 1e-3, "weight_decay": 1e-2}),  #
]

# Parámetros comunes
EPOCHS_TASK6 = 50
DROPOUT_P    = 0.0
```

In [14]:
```
results_hist = {}
results_time = {}

for name, OptClass, opt_kwargs in combos_task6:
    model = ModelClass(in_dim, 64, n_classes) if ModelClass.__init__.__code__.co_ar
    try:
        model = ModelClass(in_dim, 64, n_classes, p=DROPOUT_P)
    except Exception:
        pass

    model.to(device)
    optimizer = OptClass(model.parameters(), **opt_kwargs)
    hist = defaultdict(list)

    t0 = time.perf_counter()
    for epoch in range(EPOCHS_TASK6):
        tr_loss, tr_acc = _train_epoch_task6(model, train_loader, criterion_task6,
        va_loss, va_acc, va_f1 = _eval_epoch_task6(model, val_loader, criterion_tas
        with torch.no_grad():
            tr_loss_eval, tr_acc_eval, tr_f1 = _eval_epoch_task6(model, train_loade
```

```
        hist["loss_tr"].append(tr_loss)
        hist["acc_tr"].append(tr_acc)
        hist["f1_tr"].append(tr_f1)

        hist["loss_va"].append(va_loss)
        hist["acc_va"].append(va_acc)
        hist["f1_va"].append(va_f1)
    t1 = time.perf_counter()

    results_hist[name] = hist
    results_time[name] = t1 - t0

summary_rows = []
for name, hist in results_hist.items():
    summary_rows.append({
        "combo": name,
        "final_loss_tr": hist["loss_tr"][-1],
        "final_acc_tr":  hist["acc_tr"][-1],
        "final_f1_tr":   hist["f1_tr"][-1],
        "final_loss_va": hist["loss_va"][-1],
        "final_acc_va":  hist["acc_va"][-1],
        "final_f1_va":   hist["f1_va"][-1],
        "time_s":        results_time[name],
    })
df_task6_summary = pd.DataFrame(summary_rows).sort_values(by=["final_acc_va","final
df_task6_summary
```

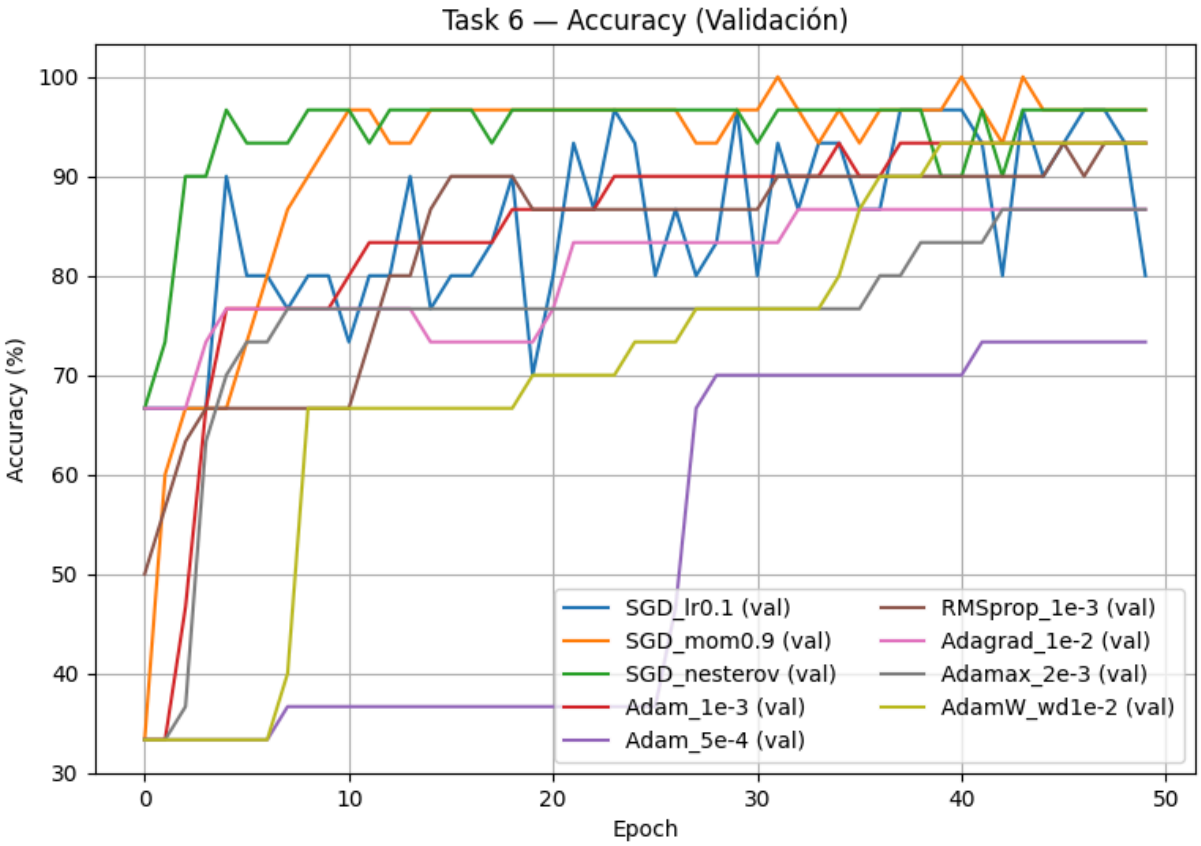
Out[14]:

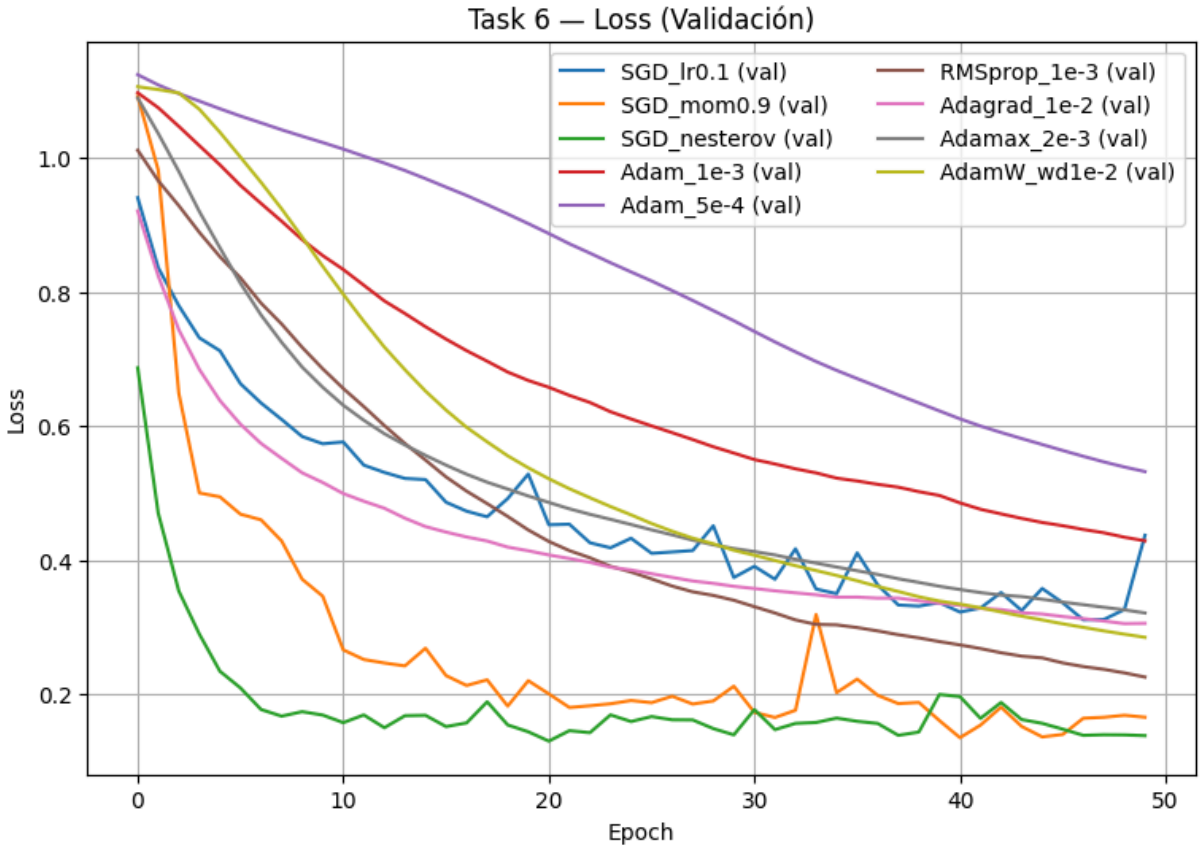
	combo	final_loss_tr	final_acc_tr	final_f1_tr	final_loss_va	final_acc_va	final_f1_va
0	SGD_mom0.9	0.418079	84.166667	0.983333	0.165833	96.666667	0.96658
1	SGD_nesterov	0.309719	88.333333	0.991665	0.138365	96.666667	0.96658
2	Adam_1e-3	0.517496	85.000000	0.958327	0.428880	93.333333	0.93333
3	RMSprop_1e-3	0.319259	90.000000	0.958327	0.225755	93.333333	0.93333
4	AdamW_wd1e-2	0.456357	79.166667	0.983323	0.285016	93.333333	0.93266
5	Adagrad_1e-2	0.450358	79.166667	0.933166	0.305824	86.666667	0.86532
6	Adamax_2e-3	0.488233	79.166667	0.933166	0.321322	86.666667	0.86532
7	SGD_lr0.1	0.418247	84.166667	0.769120	0.437123	80.000000	0.78022
8	Adam_5e-4	0.621761	72.500000	0.842001	0.532186	73.333333	0.68254

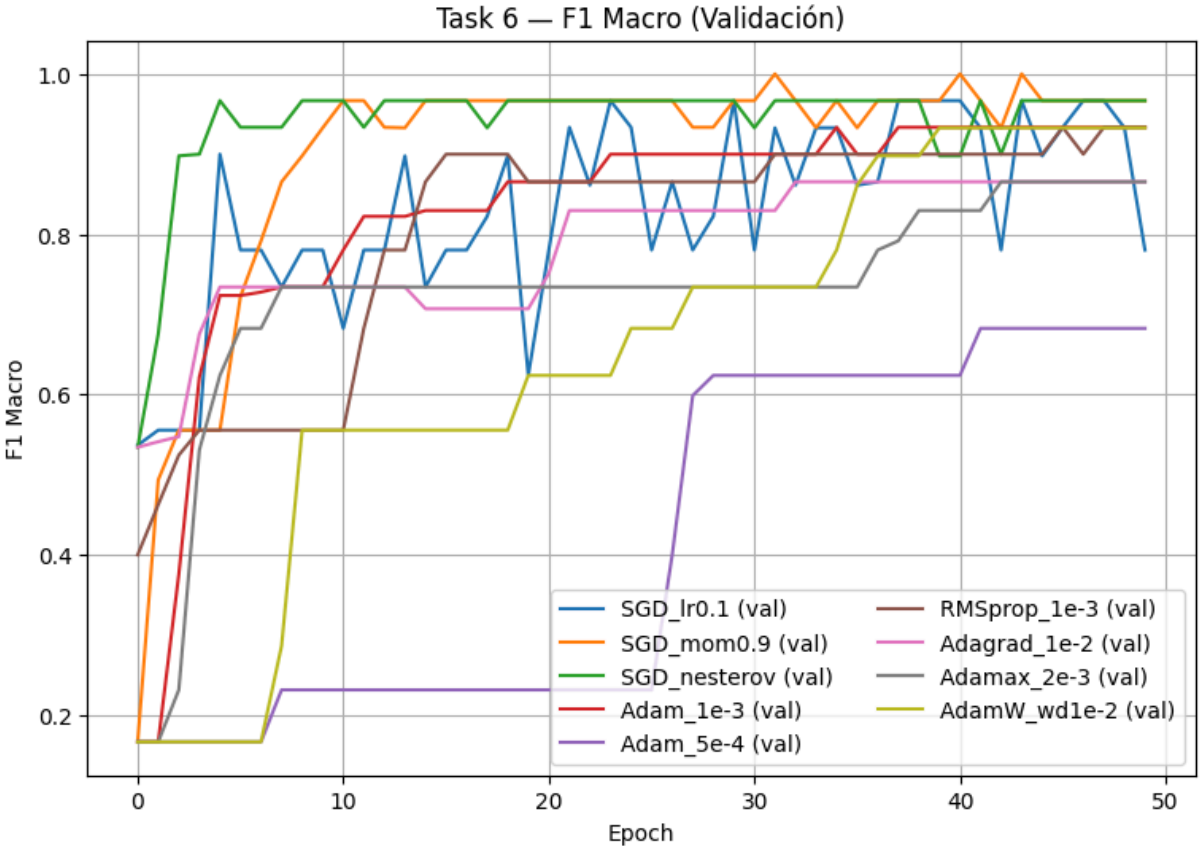
In [15]:

```
# ——— Precisión
plt.figure(figsize=(9, 6))
for name, hist in results_hist.items():
    plt.plot(hist["acc_va"], label=f"{name} (val)")
plt.title("Task 6 — Accuracy (Validación)")
plt.xlabel("Epoch")
plt.ylabel("Accuracy (%)")
plt.legend(ncol=2)
```

```
plt.grid(True)
plt.show()

# ——— Pérdida
plt.figure(figsize=(9, 6))
for name, hist in results_hist.items():
    plt.plot(hist["loss_va"], label=f"{name} (val)")
plt.title("Task 6 — Loss (Validación)")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend(ncol=2)
plt.grid(True)
plt.show()

# ——— F1 Macro
plt.figure(figsize=(9, 6))
for name, hist in results_hist.items():
    plt.plot(hist["f1_va"], label=f"{name} (val)")
plt.title("Task 6 — F1 Macro (Validación)")
plt.xlabel("Epoch")
plt.ylabel("F1 Macro")
plt.legend(ncol=2)
plt.grid(True)
plt.show()
```







Task 7- Discusión

Discuta los resultados obtenidos de diferentes modelos. Compare la velocidad de convergencia y el rendimiento final de modelos utilizando diferentes funciones de pérdida, técnicas de regularización, y algoritmos de optimización. Explore y discuta por qué ciertas técnicas podrían conducir a un mejor rendimiento. tanto técnicas de regularización, funciones de pérdida como algoritmos de optimización.

Al comparar los diferentes optimizadores, se observan diferencias claras tanto en la **velocidad de convergencia** como en el **rendimiento final**:

1. **Velocidad de convergencia**
    - Los métodos basados en SGD con momentum y Nesterov mostraron una convergencia muy rápida, alcanzando alta precisión y F1 en menos de 10 épocas.
    - Adam (1e-3) también converge relativamente rápido, aunque su desempeño final fue un poco menor en validación.
    - Optimizers como Adagrad y Adamax tuvieron una curva más lenta y no alcanzaron la misma precisión que los demás.
    - Adam con tasa muy baja (5e-4) fue el más lento y se quedó en un rendimiento final bajo.
2. **Rendimiento final (Accuracy y F1)**
    - **SGD con momentum y Nesterov** obtuvieron las mejores métricas finales, con ~96.6% de precisión y F1 cercano a 0.97.
    - Adam y RMSprop quedaron un poco atrás, alrededor del 93%.
    - AdamW, a pesar de incluir regularización con weight decay, no mejoró respecto a Adam estándar y se mantuvo cercano al 93%.
    - Adagrad y Adamax quedaron limitados a ~86%.
    - SGD simple con lr=0.1 y Adam con lr=5e-4 fueron los peores en validación.
3. **Tiempos de entrenamiento**
    - Todos los métodos estuvieron en un rango de 1.6–2.0 segundos, con diferencias mínimas. Por lo tanto, el costo computacional no fue determinante en este dataset pequeño.
4. **Regularización**
    - La combinación AdamW_wd1e-2 mostró que la regularización por weight decay ayudó a estabilizar la pérdida, pero no mejoró métricas frente a Adam. En datasets más grandes o complejos probablemente tendría mayor efecto.
    - La combinación sin regularización explícita (SGD_nesterov, SGD_mom0.9) tuvo excelente desempeño, posiblemente porque el dataset es pequeño y no tiende a sobreajustar.
5. **Conclusión**
    - Para este problema, **SGD con momentum o Nesterov** es la mejor elección: rápida convergencia, alta precisión y F1.

- Adam es competitivo y más estable en escenarios generales, aunque en este dataset perdió frente a SGD.
- Técnicas de regularización no fueron determinantes aquí, pero en problemas más complejos son clave para mejorar la generalización.
- En general, el experimento muestra cómo diferentes algoritmos de optimización afectan tanto la rapidez como el rendimiento final, confirmando la importancia de elegir el optimizador adecuado según el problema.

Ejercicio 2

1. ¿Cuál es la principal innovación de la arquitectura Transformer?

La innovacion principal es que elimina por completo las redes recurrentes y convolucionales que se usaban antes y se basa solo en mecanismos de atencion en especial la auto atencion esto permite capturar dependencias entre palabras sin importar la distancia dentro de la secuencia y hace el entrenamiento mas rapido y paralelizable

2. ¿Cómo funciona el mecanismo de atención del scaled dot-product?

El scaled dot product usa queries keys y values primero calcula el producto punto entre la query y todas las keys luego divide entre la raiz cuadrada de la dimension de las keys para que los valores no crezcan demasiado despues aplica softmax para obtener pesos y finalmente combina los values con esos pesos de atencion

3. ¿Por qué se utiliza la atención de múltiples cabezales en Transformer?

Se usa multi head attention porque cada cabeza puede enfocarse en relaciones diferentes de la secuencia al mismo tiempo asi el modelo aprende distintas perspectivas en paralelo y obtiene una representacion mas rica que con una sola cabeza

4. ¿Cómo se incorporan los positional encodings en el modelo Transformer?

Como el modelo no tiene recurrencia ni convoluciones no sabe el orden de las palabras por eso se suman vectores llamados positional encodings a los embeddings de entrada estos vectores se calculan con funciones seno y coseno de distintas frecuencias lo que da una representacion unica para cada posicion

5. ¿Cuáles son algunas aplicaciones de la arquitectura Transformer más allá de la machine translation?

El Transformer nacio en traduccion automatica pero despues se aplico en parsing de oraciones en ingles y mostro buenos resultados ademas ha servido en resumen de texto respuesta a preguntas clasificacion de secuencias modelos de vision de audio y de video en general se volvio la base de muchos avances en deep learning