Práctica 6.b. Evaluando un algoritmo de aprendizaje

Dataset

En este ejercicio los datos de entrenamiento a considerar serán generados de manera artificial mediante la función ***sklearn.datasets.make_blobs***. Se ha creado el método ***generate_data()*** para poder generar estos datos

```
def generate_data():
    classes = 6
    m = 800
    std = 0.4
    centers = np.array([[-1, 0], [1, 0], [0, 1], [0, -1], [-2, 1], [-2, -1]])
    X, y = make_blobs(n_samples=m, centers=centers, cluster_std=std,
    random_state=2, n_features=2)

    return X, y
```

Se dividirán los diferentes datos entre los datos de entrenamiento, los de validación cruzada y el conjunto de datos de prueba:

```
X_train, X_, y_train, y_ = train_test_split(X, y, test_size=0.5, random_state=1)
X_cv, X_test, y_cv, y_test = train_test_split(X_, y_, test_size=0.2, random_state=1)
```

Complejidad del modelo

Se construirán dos diferentes modelos de redes neuronales, un modelo simple y un modelo complejo, y se evaluarán para determinar cuál de ellos puede ajustarse más apropiadamente al dataset.

Modelo complejo

El modelo complejo tendrá las siguientes características:

- Una capa densa con 120 unidades y activación **relu**.
- Una capa densa con 40 unidades y activación **relu**.
- Una capa densa con 6 unidades y activación **lineal**.

Se entrenará utilizando:

- La función de pérdida ***torch.nn.CrossEntropyLoss*.**
- El optimizador ***torch.optim.Adam*** con una tasa de aprendizaje de 0.001.
- 1000 épocas.

Se ha creado una clase denominada ***ComplexModel.***

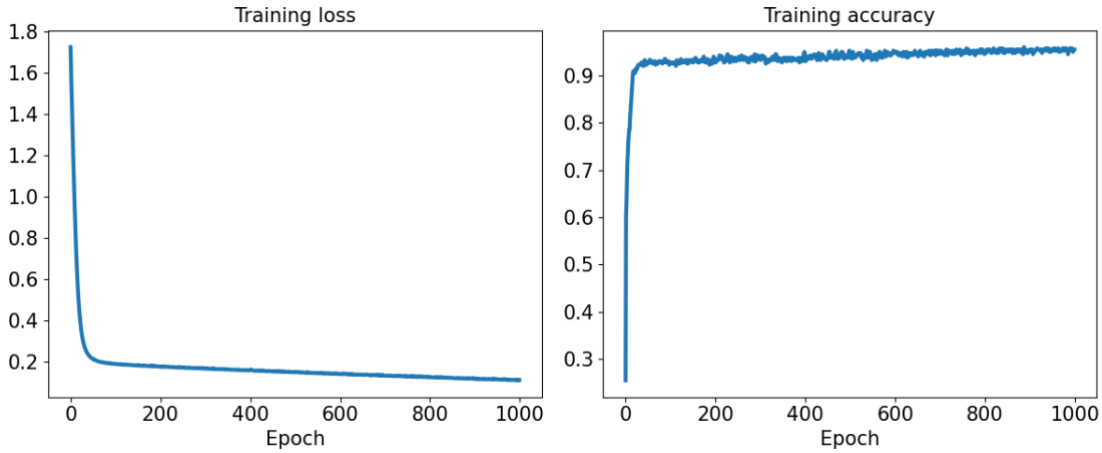
```
class ComplexModel(nn.Module):
    def __init__(self):
        super(ComplexModel, self).__init__()
        self.fc1 = nn.Linear(2, 120)
        self.fc2 = nn.Linear(120, 40)
        self.fc3 = nn.Linear(40, 6)

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

Y para la implementación del entrenamiento se ha desarrollado un método denominado ***train_complex_model()***:

```python
def train_complex_model():
    X, y = generate_data()
    X_train, X_, y_train, y_ = train_test_split(X, y, test_size=0.5, random_state=1)
    X_cv, X_test, y_cv, y_test = train_test_split(X_, y_, test_size=0.2, random_state=1)

    display_data(X_train, X_test, y_train, y_test)

    X_train_norm = (X_train - np.mean(X_train)) / np.std(X_train)
    X_train_norm = torch.from_numpy(X_train_norm).float()
    y_train = torch.from_numpy(y_train)

    train_ds = TensorDataset(X_train_norm, y_train)
    torch.manual_seed(1)
    batch_size = 120
    train_dl = DataLoader(train_ds, batch_size, shuffle=True)

    input_size = X_train_norm.shape[1]
    hidden_size = 40
    output_size = 6

    model = ComplexModel()

    learning_rate = 0.001
    loss_fn = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

    num_epochs = 1000
    log_epochs = num_epochs / 10
    loss_hist = [0] * num_epochs
    accuracy_hist = [0] * num_epochs

    for epoch in range(num_epochs):
        for x_batch, y_batch in train_dl:
            pred = model(x_batch)
            loss = loss_fn(pred, y_batch.long())

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

            loss_hist[epoch] += loss.item() * y_batch.size(0)
            is_correct = (torch.argmax(pred, dim = 1) == y_batch).float()
            accuracy_hist[epoch] += is_correct.sum()

        loss_hist[epoch] /= len(train_dl.dataset)
        accuracy_hist[epoch] /= len(train_dl.dataset)
        if epoch % log_epochs == 0:
            print(f'Epoch {epoch} loss {loss_hist[epoch]}')

    display_accuracy_loss(accuracy_hist, loss_hist)

    X_test_norm = (X_test - np.mean(X_train)) / np.std(X_train)
    X_test_norm = torch.from_numpy(X_test_norm).float()
    y_test = torch.from_numpy(y_test)

    pred_test = model(X_test_norm)

    correct = (torch.argmax(pred_test, dim = 1) == y_test).float()
    accuracy = correct.mean()

    print(f'Test accuracy: {accuracy:.4f}')
```

1. **Preparación de los datos:** La función de entrenamiento comienza dividiendo el conjunto de datos en conjuntos de entrenamiento, validación y prueba utilizando la función ***train_test_split*** de ***scikit-learn***. Luego, normaliza los datos de entrenamiento restando la media y dividiendo por la desviación estándar.
2. **Construcción del DataLoader:** Se crea un objeto ***DataLoader*** utilizando ***TensorDataset*** para convertir los datos en tensores de PyTorch y agruparlos en lotes para el entrenamiento. Esto permite iterar de manera eficiente sobre los datos durante el entrenamiento.
3. **Definición del modelo y configuración del optimizador y la función de pérdida:** Se instancia el modelo de red neuronal ***ComplexModel***. Se define la función de pérdida como la entropía cruzada categórica (CrossEntropyLoss), que es adecuada para problemas de clasificación. Se utiliza el optimizador Adam para actualizar los pesos del modelo durante el entrenamiento.
4. **Bucle de entrenamiento:** El bucle exterior itera sobre un número fijo de épocas (en este caso, 1000). Dentro de este bucle, se itera sobre los lotes de datos de entrenamiento utilizando el DataLoader. Para cada lote, se realizan los siguientes pasos:
    - Se pasa el lote a través del modelo para obtener las predicciones.
    - Se calcula la pérdida entre las predicciones y las etiquetas reales utilizando la función de pérdida definida anteriormente.
    - Se realiza la retropropagación del gradiente (backward) para calcular los gradientes de la función de pérdida con respecto a los parámetros del modelo.
    - Se aplica un paso de optimización (step) para actualizar los parámetros del modelo utilizando el optimizador.
    - Se establecen los gradientes en cero (zero_grad) para evitar acumulaciones en futuros pasos de retropropagación.
5. **Seguimiento del historial de pérdida y precisión:** Durante el entrenamiento, se registra la pérdida y la precisión en cada época. Esto permite realizar un seguimiento del rendimiento del modelo a lo largo del tiempo y diagnosticar problemas como el sobreajuste.
6. **Evaluación en el conjunto de prueba:** Una vez completado el entrenamiento, el modelo se evalúa en el conjunto de prueba para determinar su precisión final en datos no vistos.

En la última época se ha presentado una pérdida de 0.1128479778766632 y cuya gráfica es:



La precisión es 0.8375.

Modelo simple

El modelo simple tendrá las siguientes características
:

- Una capa densa con 6 unidades y activación **relu**.
- Una capa densa con 6 unidades y activación **lineal**.

Se entrenará utilizando:

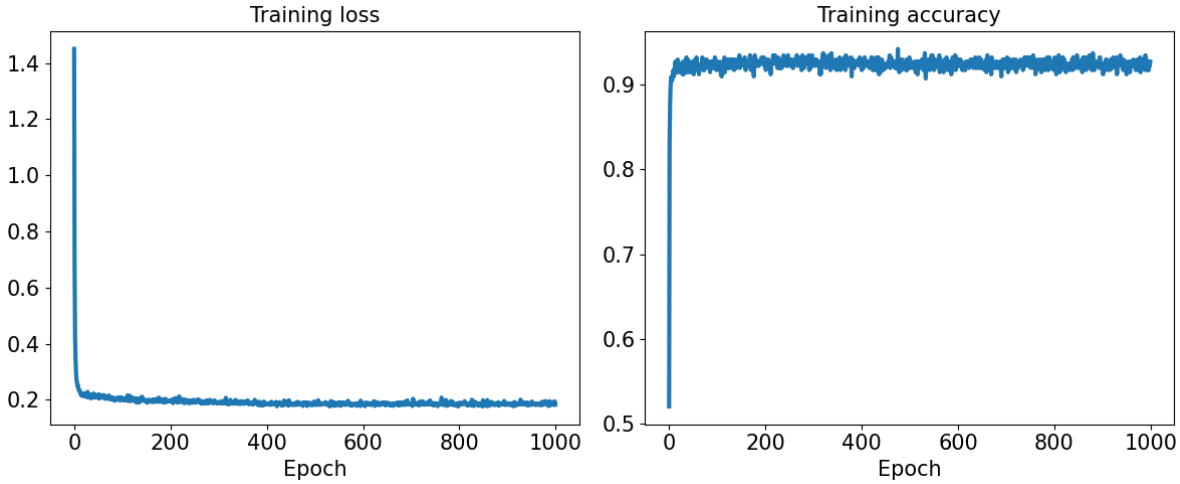
- La función de pérdida ***torch.nn.CrossEntropyLoss*.**
- El optimizador ***torch.optim.Adam*** con una tasa de aprendizaje de 0.01.
- 1000 épocas.

Se ha creado una clase denominada ***SimplexModel.***

```
class SimplexModel(nn.Module):
    def __init__(self):
        super(SimplexModel, self).__init__()
        self.fc1 = nn.Linear(2, 6)
        self.fc2 = nn.Linear(6, 6)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Y para la implementación del entrenamiento se ha desarrollado un método denominado ***train_simplex_model()***:

El funcionamiento de la función de entrenamiento es idéntico a la anterior, si se ejecuta se obtiene en la última época un error de 0.1825 con una gráfica:



Y una precisión en los datos de prueba del 0.8375.

Regularización

Se puede aplicar la regularización para moderar el impacto del modelo complejo. Para ello se ha creado un nuevo modelo complejo que tendrá las siguientes características:

- Una capa densa con 120 unidades y activación relu.
- Una capa densa con 40 unidades y activación relu.
- Una capa densa con 6 unidades y activación lineal.

Se entrenará utilizando:

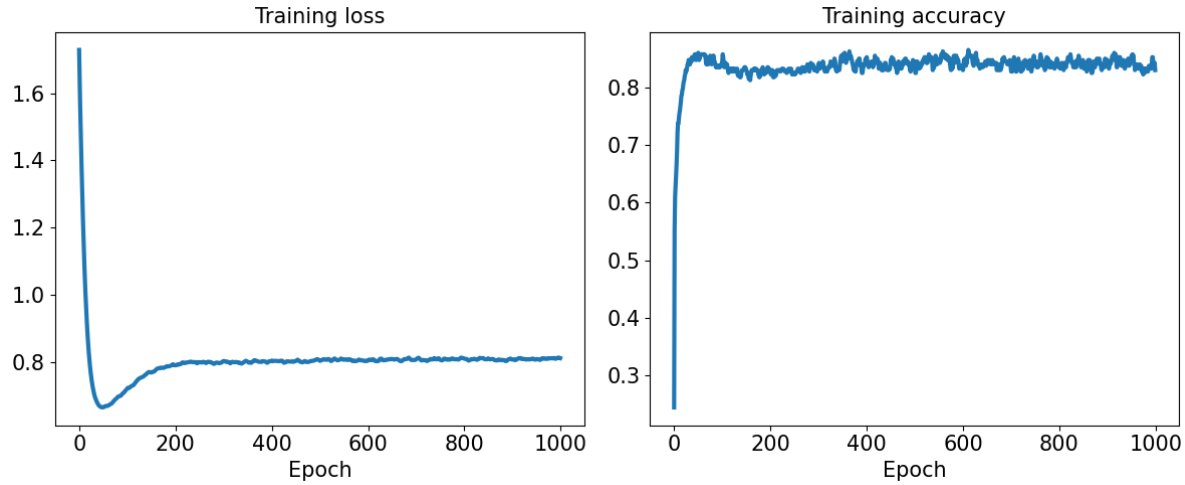
- La función de pérdida torch.nn.CrossEntropyLoss.
- El optimizador torch.optim.Adam con una tasa de aprendizaje de 0.001 y una regularización del 0.1.
- 1000 épocas.

Se ha creado la siguiente clase ***RegularizedComplexModel***

```python
class RegularizedComplexModel(nn.Module):

    def __init__(self):
        super(RegularizedComplexModel, self).__init__()
        self.fc1 = nn.Linear(2, 120)
        self.fc2 = nn.Linear(120, 40)
        self.fc3 = nn.Linear(40, 6)

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

Y para la implementación del entrenamiento se ha desarrollado un método denominado ***train_regularized_complex_model()***:

El funcionamiento del método es casi idéntico a los anteriores con la diferencia de la creación del optimizador que ahora se le pasa por parámetro la regularización especificada. Si se ejecuta se obtiene en la última época un error de 0.8127055406570435 con una gráfica:



Y una precisión en los datos de prueba del 0.8125

Iterar para encontrar el mejor valor de optimización

Para poder encontrar el mejor valor de optimización se ha creado el siguiente código:

```python
regularization_values = [0.0, 0.001, 0.01, 0.05, 0.1, 0.2, 0.3]
best_accuracy = 0.0
best_regularization_value = None

for reg_value in regularization_values:
    val_accuracy = train_regularized_complex_model_with_reg(reg_value)
    print(f"Regularization value: {reg_value}, Validation Accuracy: {val_accuracy}")
    if val_accuracy > best_accuracy:
        best_accuracy = val_accuracy
        best_regularization_value = reg_value

print(f"Best regularization value: {best_regularization_value}, Best validation accuracy: {best_accuracy}")
```

El cual consiste en iterar sobre un conjunto de valores de regularización y realizar una llamada al método ***train_regularized_complex_model_with_reg()*** (detallado más abajo) para encontrar el mejor valor de regularización basándose en la precisión.

```python
def train_regularized_complex_model_with_reg(regularization_value):
    X, y = generate_data()
    X_train, X_, y_train, y_ = train_test_split(X, y, test_size=0.5, random_state=1)
    X_cv, X_test, y_cv, y_test = train_test_split(X_, y_, test_size=0.2, random_state=1)

    X_train_norm = (X_train - np.mean(X_train)) / np.std(X_train)
    X_train_norm = torch.from_numpy(X_train_norm).float()
    y_train = torch.from_numpy(y_train)

    X_val_norm = (X_cv - np.mean(X_train)) / np.std(X_train)
    X_val_norm = torch.from_numpy(X_val_norm).float()
    y_val = torch.from_numpy(y_cv)

    train_ds = TensorDataset(X_train_norm, y_train)
    val_ds = TensorDataset(X_val_norm, y_val)

    batch_size = 120
    train_dl = DataLoader(train_ds, batch_size, shuffle=True)
    val_dl = DataLoader(val_ds, batch_size=batch_size)

    model = RegularizedComplexModel()

    learning_rate = 0.001
    loss_fn = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate, weight_decay=regularization_value)

    num_epochs = 1000
    log_epochs = num_epochs / 10
    best_val_accuracy = 0.0

    for epoch in range(num_epochs):
        model.train()
        for x_batch, y_batch in train_dl:
            optimizer.zero_grad()
            pred = model(x_batch)
            loss = loss_fn(pred, y_batch.long())
            loss.backward()
            optimizer.step()

        model.eval()
        val_accuracy = 0.0
        with torch.no_grad():
            for x_val_batch, y_val_batch in val_dl:
                val_pred = model(x_val_batch)
                val_accuracy += (torch.argmax(val_pred, dim=1) == y_val_batch).float().mean().item()

        val_accuracy /= len(val_dl)
        if val_accuracy > best_val_accuracy:
            best_val_accuracy = val_accuracy
            best_model_state_dict = model.state_dict()

    return best_val_accuracy
```

Los resultados son los siguientes:

- Regularization value: 0.0, Validation Accuracy: 0.9472222328186035
- Regularization value: 0.001, Validation Accuracy: 0.9486111005147299
- Regularization value: 0.01, Validation Accuracy: 0.9458333253860474
- Regularization value: 0.05, Validation Accuracy: 0.9375
- Regularization value: 0.1, Validation Accuracy: 0.9041666587193807
- Regularization value: 0.2, Validation Accuracy: 0.825000007947286
- Regularization value: 0.3, Validation Accuracy: 0.6680555542310079
- Best regularization value: 0.001, Best validation accuracy: 0.9486111005147299