UNIVERSIDAD NACIONAL DEL COLOMBIA

Redes convolucionales y secuenciación de ADN

Presentado por : Anderson Llanos, Felipe Jimenez, David Poveda

Resumen La secuenciación de ADN es un proceso importante en la biología molecular que permite identificar la secuencia de nucleótidos en una molécula de ADN. Los motivos son patrones de secuencias de ADN que se repiten en diferentes regiones del genoma. El estudio de estos motivos es importante para entender la regulación génica y la evolución. El objetivo principal de este proyecto es demostrar la utilidad de las redes neuronales convolucionales en tareas de regresión y clasificación de secuencias de ADN.

Palabras clave - *ADN, ARN, Motivo, Red neuronal.*

I. INTRODUCCIÓN

El genoma humano es una colección que contiene toda la información del ADN (molécula que compone el genoma) en cada célula de nuestro cuerpo. El ADN contiene la información necesaria para determinar como funcionamos y nos desarrollamos. El ADN se compone de 4 elementos fundamentales, **Adenina** (A), **Citosina** (C), **Guanina** (G) y **Timina** (T).

A finales del siglo pasado se empezaron a desarrollar técnicas de secuenciación de ADN, como la técnica de *Sanger*, que permitían escribir el ADN de una forma secuenciada (*cadena de nucleótidos*) para su posterior estudio en la reciente área de la bioinformática que intentaba mezclar el vasto conocimiento en el área de la genética y la biología junto con el poder de computo de las máquinas, logrando como resultado avances significativos en distintas áreas como la son *medicina*, *agricultura*, *seguridad* y *investigación científica*.

II. MOTIVOS DE ADN

Un *motivo* de ADN es un patrón de secuencia de nucleótidos que se repite en diferentes regiones del genoma. El estudio de estos motifs es importante para entender la regulación génica y la evolución. Los motivos son útiles para predecir la estructura y función de proteínas desconocidas, ya que los motifs conservados en diferentes especies suelen estar asociados con funciones biológicas específicas. Los motivos son pequeñas subcadenas (5-10 nucleotidos) de cadenas de ADN bastante extensas.

Fig. 1. Secuencia de nucleótidos.

Fig. 2. Posible motivo en la secuencia previa de nucleótidos.

III. REDES NEURONALES

Una *red neuronal* es un modelo matemático que se inspira en el cerebro humano y se utiliza para resolver problemas complejos de aprendizaje automático. Está compuesta por capas de nodos, que incluyen una capa de entrada, una o más capas ocultas y una capa de salida. Cada nodo está conectado a otro y tiene un peso y un umbral asociados. Si la salida de un nodo individual está por encima del valor de umbral especificado, ese nodo se activa y envía datos a la siguiente capa de la red. De lo contrario, no se pasa ningún dato a la siguiente capa de la red.

Cada capa puede realizar operaciones, ajustar pesos entre las conexiones, aprender patrones para realizar tareas de predicción y clasificación tales como procesamiento de imágenes, procesamiento lenguaje natural.

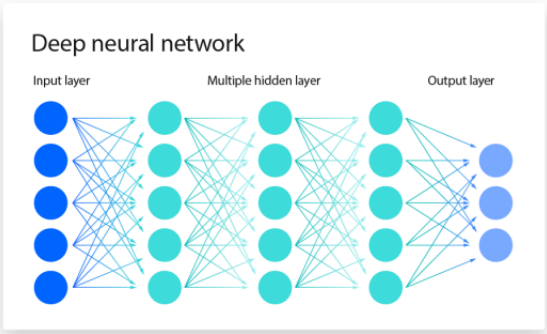

Fig. 3. Ilustración de una red neuronal.

IV. REDES NEURONALES CONVOLUCIONALES

Una *red neuronal convolucional* (CNN) es un tipo de red neuronal artificial que se utiliza comúnmente en tareas de visión por computadora y procesamiento de señales. La principal característica de las CNN es su capacidad para detectar patrones en datos de entrada, como imágenes, mediante el uso de filtros convolucionales. Estos filtros se aplican a la imagen de entrada para extraer características relevantes, como bordes, texturas y formas. Las capas de agrupación se utilizan para reducir la dimensionalidad de la imagen y mejorar la eficiencia computacional. Las CNN

se utilizan comúnmente para tareas de clasificación de imágenes, detección de objetos y segmentación de imágenes.

En el contexto de la secuenciación de ADN, las redes neuronales convolucionales se han utilizado para tareas de clasificación y regresión de secuencias de ADN. La función de las capas convolucionales es convertir la secuencia de ADN en valores numéricos que la red neuronal pueda interpretar y luego extraer patrones relevantes.

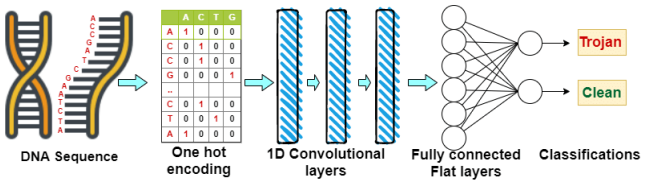

Fig. 4. CNN en la clasificación de motivos de ADN.

V. FORWARD PROPAGATION

El *forward propagation* es el proceso de propagar la entrada de la red a través de las capas de la red para producir una salida. Durante el proceso de forward propagation, cada capa de la red realiza una operación matemática en la entrada y produce una salida que se utiliza como entrada para la siguiente capa de la red. El proceso continúa hasta que se produce una salida final.

VI. BACKWARD PROPAGATION

El *back propagation* es el proceso de propagar el error de la red hacia atrás a través de la red para ajustar los pesos de la red. Durante el proceso de back propagation, se utiliza el cálculo de pérdida para calcular la derivada del error con respecto a cada peso en la red. Luego, se utiliza la derivada del error para ajustar los pesos de la red y mejorar el rendimiento de la red.

VII. LOSS CALCULATION

El *cálculo de pérdida* o *loss calculation* es el proceso de calcular la diferencia entre la salida real de la red y la salida deseada. El objetivo del cálculo de pérdida es medir qué tan bien está funcionando la red y proporcionar una señal de retroalimentación para ajustar los pesos de la red.

VIII. REGRESIÓN EN SECUENCIAS DE ADN

Para esta sección estamos interesados en estudiar que tan útil resulta ser una CNN a la hora de predecir valores cuantitativos asociados a secuencias de ADN. A modo de ejemplo, trabajaremos con las secuencias de nucleótidos de longitud 8. Pues que tenemos 4 posibilidades para cada nucleótido, tendremos $4^8 = 65536$ secuencias de ADN que usaremos para entrenar nuestra red neuronal.

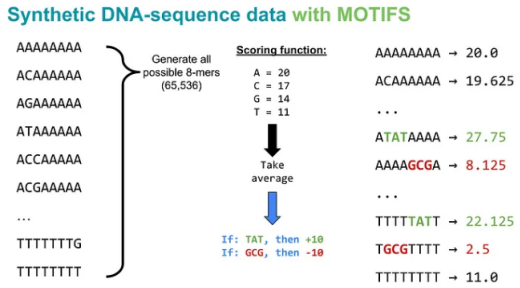

Fig. 5. Modelo de puntuación de las secuencias de ADN

A modo de ejemplificación, creamos un modelo de puntuación para cada aparición de un nucleótido en específico en nuestra cadena. Para cada nucleótido A sumaremos 20 puntos, para C 17 puntos, para G 14 puntos y para T 11 puntos. Adicionalmente, si en nuestra secuencia de nucleótidos aparece la subsecuencia TAT adicionaremos 10 puntos y si aparece la subsecuencia GCG restaremos 10 puntos. Finalmente, luego de asignar una puntuación por cada secuencia de tamaño 8, tomaremos el promedio como valor final asociado a la secuencia. En la imagen previa se muestra algunos ejemplos de esta mecánica.

Teniendo nuestros datos y modelo de puntuación definidos, tenemos que encontrar una manera de codificar nuestros nucleótidos ya que las redes neuronales funcionan en base a valores numéricos que se puedan operar, no letras. Para ello, se crea la siguiente codificación que asigna un vector binario a cada nucleótido como se muestra en la siguiente imagen.

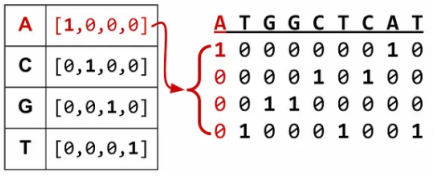

Fig. 6. Codificación de los nucleótidos

Sabiendo como se van a codificar los datos y como vamos a asignarles valores específicos a cada uno, procedemos a definir nuestros modelos de redes neuronales. Para este fin, usamos la librería *Pytorch* la cuál nos permite hacer uso de herramientas de *aprendizaje profundo* y entre estas, redes neuronales. A modo de ilustración, estamos interesados en mostrar el poder de predicción de las redes neuronales convolucionales sobre, por ejemplo, las redes neuronales lineales, las cuales realizar operaciones lineales (suma y multiplicación) en cada una de sus capas. A continuación se muestra una pequeña implementación de las redes neuronales lineales y convolucionales, respectivamente.

```python
from torch.nn.modules.activation import ReLU
import torch
import torch.nn as nn

class DNA_Linear(nn.Module):

  def __init__(self, seq_len):
    super().__init__()
    self.seq_len = seq_len
    self.lin = nn.Linear(4 * seq_len, 1)

  def forward(self, xb):
    xb = xb.view(xb.shape[0], self.seq_len * 4)
    out = self.lin(xb)
    return out

class DNA_CNN(nn.Module):
  def __init__(self, seq_len, num_filters = 32, kernel_size = 3):
    super().__init__()
    self.seq_len = seq_len
    self.conv_net = nn.Sequential(
        nn.Conv1d(4, num_filters, kernel_size=kernel_size),
        nn.ReLU(inplace=True),
        nn.Flatten(),
        nn.Linear(num_filters * (seq_len - kernel_size + 1), 1)
    )

  def forward(self, xb):
    xb = xb.permute(0, 2, 1)
    out = self.conv_net(xb)
    return out
```

Fig. 7. Red neuronal lineal y convolucional

Es importante hacer una distinción en la forma en que ambas redes neuronales funcionan. El modelo Lineal intenta predecir la puntuación simplemente ponderando los nucleótidos que aparecen en cada posición y, por otro lado, el modelo convolucional utiliza

32 filtros de longitud 3 para escanear las secuencias de 8 unidades en busca de patrones informativos de 3 unidades.

Teniendo nuestros modelos definidos, vamos a distribuir nuestros 65536 datos en tres componentes. Un componente de testeo, que corresponde al 20 % del total y el restante 80 % distribuido en dos secciones, un 80 % de entrenamiento y el restante 20 % en un componente de validación. Esta distribución es apropiada ya que no basta con que la red sea entrenada, la validación nos ayuda a garantizar que el entrenamiento se haga correctamente.

Teniendo definidos nuestros modelos de redes neuronales, la siguiente etapa es vital, pues en esta definimos los modos en los que van a ser entrenadas. Es decir, la cantidad de épocas (cantidad de iteraciones), la cantidad de batches (bloques) de datos para entrenar por cada época, la forma en la que se hará la validación del modelo en cada una de estas y como se llevará a cabo el *cálculo de pérdida*. En la siguiente imagen se muestra el método usado para el cálculo de pérdida en cada época, también se muestra el optimizador usado para cada época, el cual corresponde al optimizador de *gradiente zero*, ampliamente utilizado en el contexto de redes neuronales.

```python
def loss_batch(model, loss_func, xb, yb, opt=None,verbose=False):
    '''
    Apply loss function to a batch of inputs. If no optimizer
    is provided, skip the back prop step.
    '''
    if verbose:
        print('loss batch ****')
        print("xb shape:",xb.shape)
        print("yb shape:",yb.shape)
        print("yb shape:",yb.squeeze(1).shape)
        #print("yb",yb)

    # get the batch output from the model given your input batch
    # ** This is the model's prediction for the y labels! **
    xb_out = model(xb.float())

    if verbose:
        print("model out pre loss", xb_out.shape)
        #print('xb_out', xb_out)
        print("xb_out:",xb_out.shape)
        print("yb:",yb.shape)
        print("yb.long:",yb.long().shape)

    #loss = loss_func(xb_out.float(), yb.long().float().squeeze(1))
    loss = loss_func(xb_out, yb.float()) # for MSE/regression
    # __FOOTNOTE 2__

    if opt is not None: # if opt
        loss.backward()
        opt.step()
        opt.zero_grad()

    return loss.item(), len(xb)
```

Fig. 8. Cálculo de pérdida y re-entrenamiento.

Después de haber definido cada uno de los pasos previos, solo resta ejecutar los modelos y ver su comportamiento en cada una de las épocas. Para ello, podemos utilizar la siguiente porción de código para decirle a *Google Colab* que en caso que detecte una GPU disponible, que la use, pues esto nos ayudara a realizar los cálculos en la fase de entrenamiento de manera más eficiente sin dejarle toda la carga al procesador.

Hacemos la ejecución de nuestros modelos usando el código expuesto en la figura 9 y de esta manera obtendremos la siguiente gráfica de la figura 10.

De la gráfica anterior podemos observar que la red neuronal convolucional fue capaz de aprender, pues su tasa de pérdida está

```python
seq_len = len(train_df['seq'].values[0])

# create Linear model object
model_cnn = DNA_CNN(seq_len)
model_cnn.to(DEVICE) # put on GPU

# run the model with default settings!
cnn_train_losses, cnn_val_losses = run_model(
    train_dl,
    val_dl,
    model_cnn,
    DEVICE
)

cnn_data_label = (cnn_train_losses,cnn_val_losses,"CNN")
quick_loss_plot([lin_data_label,cnn_data_label])
```

Fig. 9. Código para usar GPU en los cálculos de entrenamiento.

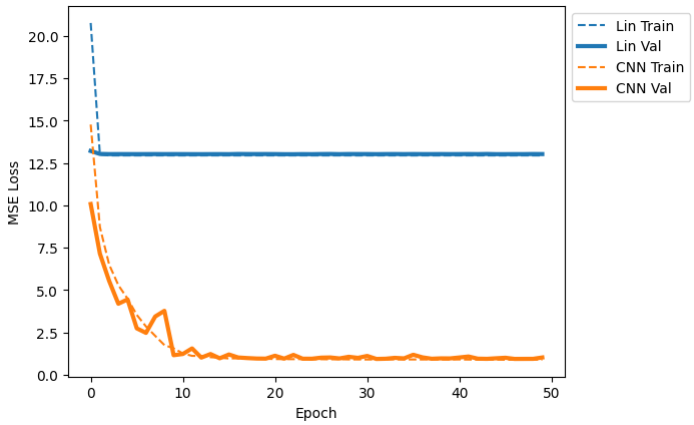

Fig. 10. Aprendizaje de la red neuronal lineal vs convolucional

tendiendo a cero. En otras palabras, la red neuronal convolucional fue capaz de aprender a detectar patrones importantes del conjunto de datos con la que fue entrenada mientras que la red neuronal lineal no.

A modo de ilustración, veamos las puntuaciones que nuestra red neuronal lineal predice en comparación con nuestra red neuronal convolucional. El último valor entre paréntesis es un indicador de error entre el valor real con el valor predicho.

```
AAGCGAAA: pred:17.086 actual:8.125 (8.961)
CGCGCCCC: pred:12.365 actual:6.250 (6.115)
GGGCGGGG: pred:8.273 actual:4.375 (3.898)
TTGCGTTT: pred:13.254 actual:2.500 (10.754)
```

Fig. 11. Red neuronal lineal : Valor real vs predicción

```
AAGCGAAA: pred:8.776 actual:8.125 (0.651)
CGCGCCCC: pred:6.937 actual:6.250 (0.687)
GGGCGGGG: pred:5.160 actual:4.375 (0.785)
TTGCGTTT: pred:3.101 actual:2.500 (0.601)
```

Fig. 12. Red convolucional : Valor real vs predicción

De aquí es fácil deducir que las redes neuronales convolucionales resultan ser óptimas para procesos de predicción de valores cuantitativos asociadas a secuencias de ADN. Para el ejemplo, se creó una estrategia de puntuación para las secuencias y, una

vez entrenadas nuestras redes con todas nuestras configuraciones, podemos ver el grado de acierto que tiene la red convolucional en comparación con una red neuronal simple cuya arquitectura interna no está pensada para solucionar problemas complejos como el estudio del ADN.

En [1] se puede encontrar más información del uso de las CNN en tareas de regresión en el contexto de secuenciación de ADN,.

IX. CLASIFICACIÓN DE MOTIVOS DE ADN

Sabiendo que las CNN se pueden utilizar en tareas de regresión con buenos resultados de predicción, presentamos esta sección en las que utilizamos las redes neuronales con fines de clasificar secuencias de ADN. Como se mencionó en un inicio, la ventaja principal de las redes neuronales convolucionales es que estas pueden aprender a detectar patrones relevantes ya sean en imágenes o, en este contexto, en secuencias de ADN grandes. Los patrones que estas redes podrían detectar serán, entonces, lo que ya definimos como *motivos*, componentes esenciales en el estudio del genoma de los seres humanos.

Para esta sección se trabajará con una base de datos de 3800 secuencias de ADN clasificadas en 3 clases distintas, *EI*, *IE* y *N*. La base de datos es de acceso público y se puede encontrar en [2]. Antes de continuar presentamos unas definiciones importantes para la realización de la tarea de clasificación.

X. ADN Y ARN

El *ADN* es una molécula biológica que contiene la información genética que determina el desarrollo y funcionamiento de los organismos vivos. El *ARN* es una molécula que participa en la transmisión de la información genética y en la síntesis de proteínas.

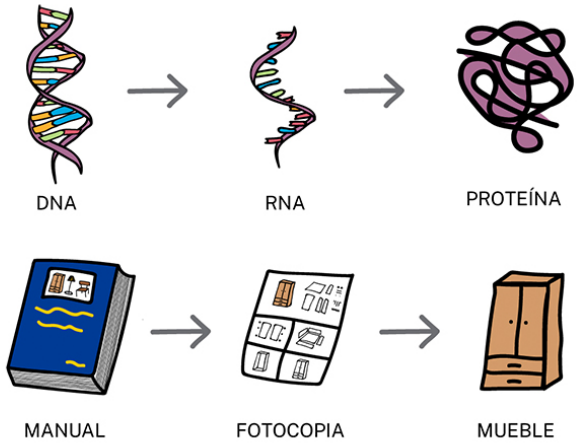

Fig. 13. Analogía del ADN y ARN

XI. INTRON Y EXON

Un *exón* es una secuencia de ADN que codifica proteínas durante la traducción. Un *intrón* es una región del ADN que forma parte de la transcripción primaria de ARN, pero a diferencia de los exones, son eliminados del transcrito maduro, previamente a su traducción. En otras palabras, los exones son moldes que contienes información para construir proteínas, los intrones son el paquete de instrucciones necesarías para que la fabricación de la proteína se haga correctamente.

XII. CLASE EI - IE - N

La *clase EI* es un tipo de sitio de empalme indica que la secuencia de ADN corresponde a la unión entre un exón (la región codificante de un gen) y un intrón (la región no codificante), ie, es un sitio de empalme que marca el inicio de un intrón y el final de un exón en la secuencia de ADN. A este empalme se le conoce como empalme *donante*.

La *clase IE* es un sitio de empalme indica la unión entre un intrón y un exón. Se refiere al sitio de empalme que marca el final de un intrón y el inicio de un exón en la secuencia de ADN. A este empalme se le conoce como empalme aceptor.

La *clase N* es la clase que no es ni EI ni IE. Estas designaciones son fundamentales para comprender cómo se lleva a cabo el empalme en el proceso de transcripción génica. Los intrones se eliminan, los exones se unen y forman moléculas de ARN maduras.

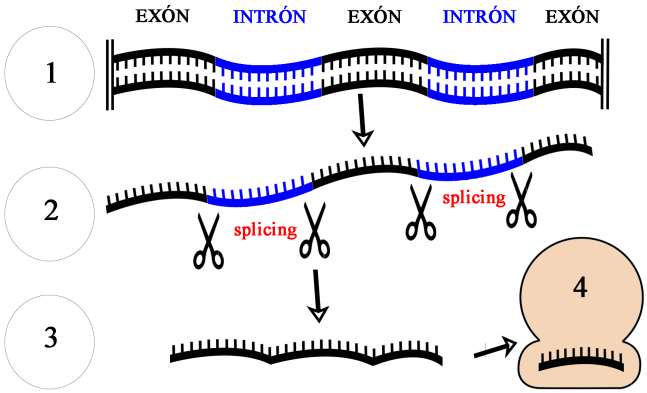

Fig. 14. Proceso de creación de ARN

Con estas definiciones en mente, resulta natural entender por que las redes neuronales convolucionales están reprensentando un rol importante en la tarea de clasificar secuencias de ADN, pues de nuevo, las CNN's pueden aprender a reconocer motivos importantes para la regulación génica. Estas redes pueden identificar patrones específicos de nucleótidos que corresponden a sitios de unión de factores de transcripción, regiones promotoras u otras características funcionales en el genoma.

Nuestra tarea de clasificación se centra de sitios de empalme (EI-IE-N). Los sitios de empalme son críticos para la formación del ARN mensajero (ARNm) funcional, ya que determinan qué regiones del ADN deben incluirse o excluirse en el ARN final.

Para nuestra tarea de clasificacion primero debemos extraer los datos de la base de datos y darle una etiqueta a cada clase, 0 para EI, 1 para IE y 2 para N.

	seq	type	type_encoded
0	CCAGCTGCATCACAGGAGGCCAGCGAGCAGG...	EI	0
1	AGACCCGCCGGGAGGCGGAGGACCTGCAGGG...	EI	0
2	GAGGTGAAGGACGTCCTTCCCCAGGAGCCGG...	EI	0
3	GGGCTGCGTTGCTGGTCACATTCCTGGCAGGT...	EI	0
4	GCTCAGCCCCCAGGTCACCCAGGAACTGACGTG...	EI	0

Fig. 15. Etiquetado de clases

Teniendo nuestras clases etiquetadas, procedemos a codificar nuestras secuencias de ADN (de longitud 60) de la misma forma que se hizo en VIII. Seguido a esto, creamos distribuimos los datos para realizar las tareas de entrenamiento, validación y testeo.

```
import random

def quick_split(df, split_frac=0.7):
    shuffled = df.sample(frac=1).reset_index(drop=True)
    split_idx = int(len(df) * split_frac)
    train_df = shuffled[:split_idx]
    rest_df = shuffled[split_idx:]
    return train_df, rest_df

# Genera los conjuntos de entrenamiento, validación y prueba
train_df, remaining_df = quick_split(real_data, split_frac=0.7)
val_df, test_df = quick_split(remaining_df, split_frac=0.5)

# Verifica los tamaños de los conjuntos
print(f"Entrenamiento: {len(train_df)} muestras")
print(f"Validación: {len(val_df)} muestras")
print(f"Prueba: {len(test_df)} muestras")
```

```
Entrenamiento: 2233 muestras
Validación: 478 muestras
Prueba: 479 muestras
```

Fig. 16. Distribución de los datos

Seguido a esto, creamos un método que nos permitirá decirle a la red neuronal algunos de los motivos presentes en cada clase de nuestro interés.

```
def generate_simulated_dna_sequence(length, type):
    """Generate a simulated DNA sequence with more complexity."""
    motifs = {
    'EI': ['GT', 'GTAAGT'],  # Exon-Intron splice sites
    'IE': ['AG', 'CAGG'],  # Intron-Exon splice sites
    'N': ['TATA', 'CAAT', 'AAA', 'TTT', 'CCC', 'GGG']
  }
```

Fig. 17. Algunos motivos destacados de cada clase

Teniendo los datos definidos y distribuídos, se procede a definir la red neuronal convolucional como se hizo en la sección VIII.

```
class DNA_CNN(nn.Module):
    def __init__(self, seq_len, num_classes):
        super(DNA_CNN, self).__init__()
        self.conv1 = nn.Conv1d(4, 32, kernel_size=3, stride=1, padding=1)
        self.pool = nn.MaxPool1d(2, 2)
        self.conv2 = nn.Conv1d(32, 64, kernel_size=3, stride=1, padding=1)
        self.dropout = nn.Dropout(0.5)
        self.fc1 = nn.Linear(64 * (seq_len // 4), 128)
        self.fc2 = nn.Linear(128, num_classes)

    def forward(self, x):
        x = x.permute(0, 2, 1)
        x = self.pool(nn.functional.relu(self.conv1(x)))
        x = self.pool(nn.functional.relu(self.conv2(x)))
        x = x.view(x.size(0), -1)
        x = self.dropout(x)
        x = nn.functional.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Fig. 18. Definición de la red neuronal a usar

Teniendo nuestra red neuronal definida, procedemos a definir las funciones bucle de entrenamiento, la función de validación, el optimizador y la función de pérdida.

```
import torch.optim as optim
# Definir el número de épocas
num_epochs = 20  # Puedes ajustar este número según tus necesidades

# Cambiar a un optimizador más eficiente como Adam
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Definir la función de perdida
loss_func = nn.CrossEntropyLoss()

# Considera implementar una estrategia de ajuste de ratio de aprendizaje
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min')

def train_one_epoch(model, train_loader, loss_func, optimizer, device):
    model.train()  # Establecer el modelo en modo de entrenamiento
    total_loss = 0

    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)

        # Paso hacia adelante
        outputs = model(inputs)
        loss = loss_func(outputs, labels)

        # Paso hacia atrás y optimización
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    avg_loss = total_loss / len(train_loader)
    return avg_loss

def validate(model, val_loader, loss_func, device):
    model.eval()  # Establecer el modelo en modo de evaluación
    total_loss = 0
    with torch.no_grad():
        for inputs, labels in val_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            loss = loss_func(outputs, labels)
            total_loss += loss.item()

    avg_loss = total_loss / len(val_loader)
    return avg_loss
```

Fig. 19. Fase de entrenamiento y validación

Para nuestra tarea de clasificación hemos usado el optimizador de *Adam* junto con la función *CrossEntropyLoss* como nuestra función de pérdida. Ejecutamos nuestro bucle de entrenamiento.

```
# Ejemplo de cómo ejecutar el bucle de entrenamiento y validación
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

train_losses = []
val_losses = []

for epoch in range(num_epochs):
    train_loss = train_one_epoch(model, train_loader, loss_func, optimizer, device)
    val_loss = validate(model, val_loader, loss_func, device)
    scheduler.step(val_loss)
    train_losses.append(train_loss)
    val_losses.append(val_loss)
    print(f'Epoch {epoch+1}/{num_epochs},
    Train Loss: {train_loss:.4f},
    Val Loss: {val_loss:.4f}')
```

Fig. 20. Ejecución de la fase de entrenamiento

Al ejecutar nuestro modelo podremos ver sus variaciones y su

línea de aprendizaje en cada época. Recordemos que un indicador de un buen aprendizaje en las redes neuronales es que la *pérdida* disminuya con el paso de las épocas.

```
Epoch 1/20, Train Loss: 0.9004, Val Loss: 0.5294
Epoch 2/20, Train Loss: 0.4120, Val Loss: 0.2169
Epoch 3/20, Train Loss: 0.2390, Val Loss: 0.1410
Epoch 4/20, Train Loss: 0.1901, Val Loss: 0.1403
Epoch 5/20, Train Loss: 0.1582, Val Loss: 0.1124
Epoch 6/20, Train Loss: 0.1347, Val Loss: 0.1148
Epoch 7/20, Train Loss: 0.1198, Val Loss: 0.1013
Epoch 8/20, Train Loss: 0.0993, Val Loss: 0.1075
Epoch 9/20, Train Loss: 0.1114, Val Loss: 0.1034
Epoch 10/20, Train Loss: 0.1043, Val Loss: 0.1030
Epoch 11/20, Train Loss: 0.0835, Val Loss: 0.1042
Epoch 12/20, Train Loss: 0.0782, Val Loss: 0.1101
Epoch 13/20, Train Loss: 0.0856, Val Loss: 0.1056
Epoch 14/20, Train Loss: 0.0842, Val Loss: 0.0887
Epoch 15/20, Train Loss: 0.0726, Val Loss: 0.1214
Epoch 16/20, Train Loss: 0.0576, Val Loss: 0.1094
Epoch 17/20, Train Loss: 0.0513, Val Loss: 0.0986
Epoch 18/20, Train Loss: 0.0590, Val Loss: 0.1215
Epoch 19/20, Train Loss: 0.0593, Val Loss: 0.0916
Epoch 20/20, Train Loss: 0.0537, Val Loss: 0.1083
```

Fig. 21. Entrenamiento a lo largo de las épocas

Con el fin de asegurarnos que el modelo se evaluo correctamente, podemos crear un método que nos brinde métricas del proceso de evaluación para corroborar que todo se hizo correctamente.

```python
def evaluate_model(model, test_loader, device):
    model.eval()  # Establecer el modelo en modo de evaluación
    all_preds = []
    all_labels = []

    with torch.no_grad():
        for inputs, labels in test_loader:
            inputs = inputs.to(device)
            labels = labels.to(device)
            outputs = model(inputs)

            _, predicted = torch.max(outputs, 1)
            all_preds.extend(predicted.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

    return all_preds, all_labels

# Evaluar el modelo en el conjunto de prueba
all_preds, all_labels = evaluate_model(model, test_loader, device)

# Calcular y mostrar métricas de rendimiento
print(classification_report(all_labels, all_preds))
print(confusion_matrix(all_labels, all_preds))
```

```
              precision    recall  f1-score   support

           0       0.98      0.94      0.96       122
           1       0.89      0.99      0.94       117
           2       1.00      0.97      0.98       240

    accuracy                           0.97       479
   macro avg       0.96      0.97      0.96       479
weighted avg       0.97      0.97      0.97       479
```

Fig. 22. Evaluación del modelo

Lo que nos dice la gráfica anterior es que la red se ha entrenado de forma tal que puede clasificar secuencias de ADN con un grado de precisión bastante alto. El menor grado de precisión se dio para la clase 1 con un porcentaje de éxito del 89 %.

La siguiente gráfica muestra que la pérdida del modelo disminuía con el paso de las épocas, lo cual es un buen indicador, además que la gráfica de la validación tiende a un punto de equilibrio, indicando que durante el entrenamiento, la validación se encargó que el modelo se re-ajustara de forma correcta sin llegar a algún sobreajuste, es decir, que el modelo entrenó bien.

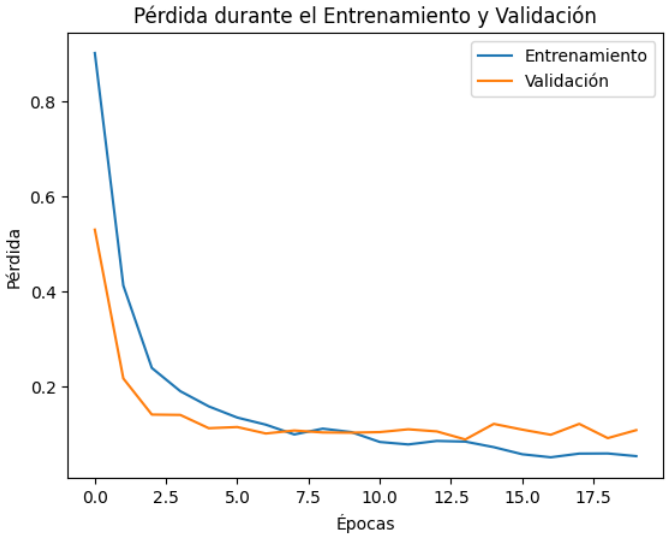

Fig. 23. Entrenamiento y validación en distintas épocas

Teniendo ahora nuestra red entrenada, podemos darle como input alguna cadena de ADN de longitud 60 para que esta nos la pueda clasificar en algunas de las tres clases. A modo de ejemplo elegimos la cadena GCCCTGGCGCCCAGCACCATGAAGATCAAGGTGAGTCGAGGGGTTGGTGGCCCTCTGCCT, la cual se encuentra disponible en [2] como una secuencia de clase EI. El programa nos da entonces como respuesta un 0, que corresponde a la etiqueta para la clase EI, de modo que en efecto la red neuronal puede predecir y lo hace con un buen grado de precisión.

```python
# Carga tu modelo (asegúrate de que esté en modo de evaluación)
model.eval()

# Cadena de ADN de ejemplo para clasificar
dna_sequence = "GCCCTGGCGCCCAGCACCATGAAGATCAAGGTGAGTCGAGGGGTTGGTGGCCCTCTGCCT"

# Codificar la secuencia
encoded_sequence = one_hot_encode(dna_sequence).unsqueeze(0)

# Realizar la predicción
with torch.no_grad():
    output = model(encoded_sequence)
    predicted_class = torch.argmax(output, dim=1)

# Imprimir la clase predicha
print(f"Clase predicha: {predicted_class.item()}")
```

```
Clase predicha: 0
```

Fig. 24. Tarea de clasificación de secuencias de ADN

XIII. CONCLUSIONES

1. Las redes neuronales convolucionales en el contexto de bioinformática sirven tanto para tareas de regresión como de clasificación dada a su arquitectura interna que permita detectar patrones importantes de los datos en la fase de entrenamiento.

2. Los porcentajes de fiabilidad pueden variar en base al tamaño de los datos de entrenamiento. Entre más grande sea la base de datos, más puede aprender la red neuronal y más confiable

resulta a la hora de hacer predicciones.

3. Para alguna de bioinformática en un contexto más real es necesario apoyarse y colaborar con distintas áreas de la biología que puedan proveer e interpretar los datos que una red neuronal utiliza para alimentarse y producir resultados precisos.

REFERENCES

[1] https://towardsdatascience.com/modeling-dna-sequences-with-pytorch-de28b0a05036

[2] https://archive.ics.uci.edu/ml/machine-learning-databases/molecular-biology/splice-junction-gene-sequences/splice.data

[3] DNA Sequence Classification by Convolutional Neural Network. Nguyen Giang, Vu Anh Tran, Duc Luu Ngo, Dau Phan.