Lab 3:

- Carlos Jarrin
- Fausto Yugcha

NLP and Neural Networks

In this exercise, we'll apply our knowledge of neural networks to process natural language. As we did in the bigram exercise, the goal of this lab is to predict the next word, given the previous one.

Data set

Load the text from "One Hundred Years of Solitude" that we used in our bigrams exercise. It's located in the data folder.

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Important note:

Start with a smaller part of the text. Maybe the first 10 parragraphs, as the number of tokens rapidly increases as we add more text.

Later you can use a bigger corpus.

Don't forget to prepare the data by generating the corresponding tokens.

```
import torch
import torch.nn as nn
import torch.optim as optim

from nltk import bigrams
from nltk.tokenize import TreebankWordTokenizer

from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score

import numpy as np
```

```
# Cargar el texto y tokenizar
tokenizer = TreebankWordTokenizer()
text = open('/content/drive/MyDrive/NLP/datos/cap1.txt', 'r').read().lower()
tokens = tokenizer.tokenize(text)
print(f"tokens = {len(tokens)=}")
```

```
tokens = len(tokens)=6293
```

Let's prepare the data set.

Our neural network needs to have an input X and an output y. Remember that these sets are numerical, so you'd need something to map the tokens into numbers, and viceversa.

```
# Generar bigramas (pares de palabras)
bigram_list = list(bigrams(tokens))


X = [bigram[0] for bigram in bigram_list] # Primera palabra del bigrama
y = [bigram[1] for bigram in bigram_list] # Segunda palabra del bigrama


# Convertir las palabras a una representación numérica
vectorizer = CountVectorizer()
X_vectorized = vectorizer.fit_transform(X)


# Dividir los datos en entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X_vectorized, y, test_size = 0.2, ran


# Codificar las etiquetas
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y)

# Convertir los datos a tensores
X_tensor = torch.tensor(X_vectorized.toarray(), dtype = torch.float32)
y_tensor = torch.tensor(y_encoded, dtype=torch.long)

# Dividir en conjunto de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X_tensor, y_tensor, test_size = 0.2,


# Note that our vectors are integers, which can be thought as a categorical variables.
# torch provides the one_hot method, that would generate tensors suitable for our nn
# make sure that the dtype of your tensor is float.


type(X_tensor)
type(y_tensor)
```

```
torch.Tensor
```

Network design

To start, we are going to have a very simple network. Define a single layer network

```
# Parámetros de la red
input_size = X_train.shape[1]
hidden_size = 128  # Ajustado para más capas ocultas
output_size = len(label_encoder.classes_)
dropout_rate = 0.3  # Para evitar el sobreajuste


# Definir una red neuronal más profunda
class ImprovedNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(ImprovedNN, self).__init__()
        # Primera capa densa
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu1 = nn.ReLU()
        # Segunda capa densa
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.relu2 = nn.ReLU()
        # Dropout para evitar sobreajuste
        self.dropout = nn.Dropout(dropout_rate)
        # Capa de salida
        self.fc3 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu1(x)
        x = self.fc2(x)
        x = self.relu2(x)
        x = self.dropout(x)
        x = self.fc3(x)
        return x


# Crear el modelo
model = ImprovedNN(input_size, hidden_size, output_size)

# Definir el criterio de pérdida y el optimizador
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)


# Entrenar el modelo
n_epochs = 180

for epoch in range(n_epochs):
    model.train()

    # Hacer predicciones y calcular la pérdida
    outputs = model(X_train)
    loss = criterion(outputs, y_train)
```

```
    # Actualizar los pesos
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    # Calcular precisión en los datos de entrenamiento
    _, predicted = torch.max(outputs, 1)
    train_accuracy = accuracy_score(y_train, predicted)

    # Evaluar en el conjunto de prueba
    model.eval()
    with torch.no_grad():
        outputs_test = model(X_test)
        _, predicted_test = torch.max(outputs_test, 1)
        test_accuracy = accuracy_score(y_test, predicted_test)

    if (epoch+1) % 20 == 0:
        print(f"Epoch [{epoch+1}/{n_epochs}], Loss: {loss.item():.4f}, Train Accuracy: {t
```

```
Epoch [20/180], Loss: 7.5063, Train Accuracy: 1.93%, Test Accuracy: 1.83%
Epoch [40/180], Loss: 6.6460, Train Accuracy: 2.56%, Test Accuracy: 1.83%
Epoch [60/180], Loss: 5.7510, Train Accuracy: 7.17%, Test Accuracy: 6.59%
Epoch [80/180], Loss: 5.5864, Train Accuracy: 7.87%, Test Accuracy: 6.91%
Epoch [100/180], Loss: 5.5013, Train Accuracy: 7.59%, Test Accuracy: 6.99%
Epoch [120/180], Loss: 5.4314, Train Accuracy: 9.16%, Test Accuracy: 7.07%
Epoch [140/180], Loss: 5.3218, Train Accuracy: 11.74%, Test Accuracy: 8.18%
Epoch [160/180], Loss: 5.1450, Train Accuracy: 15.58%, Test Accuracy: 9.37%
Epoch [180/180], Loss: 4.9121, Train Accuracy: 20.23%, Test Accuracy: 9.93%
```

```
# Función para predecir la siguiente palabra dada una palabra
def predict_next_word(input_word):
    input_vectorized = vectorizer.transform([input_word]).toarray()
    input_tensor = torch.tensor(input_vectorized, dtype=torch.float32)

    model.eval()
    with torch.no_grad():
        output = model(input_tensor)
        probabilities = torch.softmax(output, dim=1)
        predicted_prob, predicted_idx = torch.max(probabilities, 1)

    predicted_word = label_encoder.inverse_transform(predicted_idx.numpy())[0]
    return predicted_word, predicted_prob.item()


# Probar predicción
max_n_pred = 10
for _ in range(10):
  word = 'aldea'
  full_pred = word
  for i in range(max_n_pred):
    word2 = predict_next_word(word)[0]
    full_pred = full_pred + ' ' + word2
```

```
    word = word2
  print(full_pred)
```

```
aldea , de la la la la la la la la
aldea , de la la la la la la la la
aldea , de la la la la la la la la
aldea , de la la la la la la la la
aldea , de la la la la la la la la
aldea , de la la la la la la la la
aldea , de la la la la la la la la
aldea , de la la la la la la la la
aldea , de la la la la la la la la
aldea , de la la la la la la la la
```

Analysis

1. Test your network with a few words

```
def pred_n_words(word = 'buendia', max_n_pred = 10):
  full_pred = word
  l1 = 0
  for i in range(max_n_pred):
    word2 = predict_next_word(word)[0]
    pr = predict_next_word(word)[1]
    full_pred = full_pred + ' ' + word2
    word = word2
    l1 += np.log(pr)

  n_ll = l1/max_n_pred
  print(full_pred, '| neg log:', n_ll)

palabras = ['buendia', 'niño', 'posibilidad', 'casa', 'muchos']
for w in palabras:
  pred_n_words(word = w, max_n_pred =1)
print(' ')
for w in palabras:
  pred_n_words(word = w)
```

```
buendia de | neg log: -3.309816360084446
niño de | neg log: -1.8394332701900957
posibilidad de | neg log: -0.1501823283726358
casa , | neg log: -1.903839449475963
muchos de | neg log: -2.4020923375421463

buendia de la la la la la la la la la | neg log: -5.30245997287211
niño de la la la la la la la la la | neg log: -5.1554216638826755
posibilidad de la la la la la la la la la | neg log: -4.986496569700929
casa , de la la la la la la la la | neg log: -4.9020894451768395
muchos de la la la la la la la la la | neg log: -5.2116875706178805
```

2. What does each value in the tensor represents?

Al ser un tensor de convolucion requiere de valores en forma matricial para funcionar de manera adecuada, por lo que el tensor proporcionado debe ajustarse.

3. Why does it make sense to choose that number of neurons in our layer?

Cada capa de entrada debe tener la misma cantidad de salida por que asi fue definido el biagram.

4. What's the negative likelihood for each example?

Es una medida que nos ayuda a cuantificar segun el modelo propuesto que tan probable es que la palabra sea la verdadera.

5. Try generating a few sentences?

Se debe generar con un bucle para generar un amplio vocabulario y no repetir las mismas palabras en bucle cerrado.

6. What's the negative likelihood for each sentence?

Vendria a ser la sumatoria de cada una de las palabras.

Design your own neural network (more layers and different number of neurons)

The goal is to get sentences that make more sense

NUEVO INTENTO DEL MODELO

```
import numpy as np
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Dense, LSTM, Bidirectional

# Tokenización a nivel de palabras
tokenizer = Tokenizer()
tokenizer.fit_on_texts([text])
total_words = len(tokenizer.word_index) + 1

# Crear secuencias de bigramas
sequences = []
input_sequences = text.split()  # Dividir el texto en palabras
```

```
for i in range(len(input_sequences) - 1):
    bigram = input_sequences[i:i+2]  # Crear bigramas de exactamente 2 palabras
    seq = tokenizer.texts_to_sequences([bigram])[0]  # Convertir bigrama a secuencia de í

    # Asegurarse de que la secuencia sea de longitud 2
    if len(seq) == 2:
        sequences.append(seq)

# Convertir las secuencias a arrays de NumPy
sequences = np.array(sequences)
X, y = sequences[:, 0], sequences[:, 1]
X = np.expand_dims(X, axis=-1)  # Añadir una dimensión para que X tenga 3 dimensiones

# Convertir etiquetas a one-hot encoding
y = to_categorical(y, num_classes=total_words)

# Definir el modelo de predicción de palabras
model = Sequential()
model.add(Embedding(input_dim=total_words, output_dim=10, input_length=1))  # Embedding d
model.add(Bidirectional(LSTM(50)))  # La salida del embedding es 3D, compatible con LSTM
model.add(Dense(total_words, activation='softmax'))

# Compilar el modelo
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

# Entrenar el modelo
model.fit(X, y, epochs=50, verbose=2)
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/embedding.py:90: Use
  warnings.warn(
Epoch 1/50
145/145 - 5s - 38ms/step - accuracy: 0.0894 - loss: 7.3735
Epoch 2/50
145/145 - 2s - 11ms/step - accuracy: 0.0920 - loss: 6.0652
Epoch 3/50
145/145 - 1s - 7ms/step - accuracy: 0.0920 - loss: 5.6774
Epoch 4/50
145/145 - 1s - 7ms/step - accuracy: 0.0920 - loss: 5.5525
Epoch 5/50
145/145 - 1s - 6ms/step - accuracy: 0.0920 - loss: 5.4678
Epoch 6/50
145/145 - 1s - 8ms/step - accuracy: 0.0926 - loss: 5.4052
Epoch 7/50
145/145 - 1s - 6ms/step - accuracy: 0.0924 - loss: 5.3458
Epoch 8/50
145/145 - 1s - 7ms/step - accuracy: 0.0924 - loss: 5.2752
Epoch 9/50
145/145 - 2s - 10ms/step - accuracy: 0.0931 - loss: 5.2073
Epoch 10/50
145/145 - 3s - 18ms/step - accuracy: 0.0987 - loss: 5.1355
Epoch 11/50
145/145 - 1s - 6ms/step - accuracy: 0.1163 - loss: 5.0509
Epoch 12/50
145/145 - 1s - 8ms/step - accuracy: 0.1312 - loss: 4.9583
Epoch 13/50
145/145 - 1s - 9ms/step - accuracy: 0.1358 - loss: 4.8586
```

```
Epoch 14/50
145/145 - 1s - 6ms/step - accuracy: 0.1384 - loss: 4.7540
Epoch 15/50
145/145 - 1s - 9ms/step - accuracy: 0.1575 - loss: 4.6526
Epoch 16/50
145/145 - 1s - 9ms/step - accuracy: 0.1668 - loss: 4.5545
Epoch 17/50
145/145 - 1s - 6ms/step - accuracy: 0.1839 - loss: 4.4596
Epoch 18/50
145/145 - 1s - 6ms/step - accuracy: 0.1989 - loss: 4.3675
Epoch 19/50
145/145 - 1s - 9ms/step - accuracy: 0.2226 - loss: 4.2731
Epoch 20/50
145/145 - 1s - 10ms/step - accuracy: 0.2566 - loss: 4.1747
Epoch 21/50
145/145 - 3s - 19ms/step - accuracy: 0.2657 - loss: 4.0802
Epoch 22/50
145/145 - 2s - 13ms/step - accuracy: 0.2746 - loss: 3.9862
Epoch 23/50
145/145 - 1s - 9ms/step - accuracy: 0.2768 - loss: 3.8939
Epoch 24/50
145/145 - 1s - 9ms/step - accuracy: 0.2822 - loss: 3.8097
Epoch 25/50
145/145 - 1s - 6ms/step - accuracy: 0.2907 - loss: 3.7269
Epoch 26/50
145/145 - 1s - 6ms/step - accuracy: 0.2974 - loss: 3.6510
Epoch 27/50
145/145 - 1s - 9ms/step - accuracy: 0.3043 - loss: 3.5787
```

```
# Función para predecir la siguiente palabra basada en un bigrama dado
def predict_next_word(input_text, model, tokenizer, total_words):
    input_seq = tokenizer.texts_to_sequences([input_text.split()])[-1]  # Convertir el bi

    # Si el bigrama tiene más de una palabra, solo tomamos la última
    input_seq = np.array([input_seq[-1]])  # Convertimos a numpy array y obtenemos el últ
    input_seq = np.expand_dims(input_seq, axis=-1)  # Asegurarse de que la entrada sea 3D

    # Predecir la próxima palabra
    prediction = model.predict(input_seq)
    predicted_word_index = np.argmax(prediction, axis=-1)[0]  # Obtener el índice de la p

    # Convertir el índice predicho a la palabra correspondiente
    predicted_word = tokenizer.index_word[predicted_word_index]
    return predicted_word

# Texto de prueba
input_bigram = "Aureliano"  # Por ejemplo, un bigrama en tu dataset de prueba
predicted_word = predict_next_word(input_bigram, model, tokenizer, total_words)
print(f"Predicción para el bigrama '{input_bigram}': {predicted_word}")
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 1s/step
Predicción para el bigrama 'Aureliano': buendía
```

```
def generate_sentence(seed_text, model, tokenizer, total_words, max_words=20):
    for _ in range(max_words):
```

```
        # Predecir la siguiente palabra
        predicted_word = predict_next_word(seed_text, model, tokenizer, total_words)

        # Añadir la palabra predicha a la semilla de texto
        seed_text += " " + predicted_word

        # Detener la generación si se encuentra un punto (.)
        if predicted_word == '.':
            break

    return seed_text

# Iniciar con un bigrama de prueba
seed_text = "Aureliano"
predicted_sentence = generate_sentence(seed_text, model, tokenizer, total_words, max_word

print(f"Oración generada: {predicted_sentence}")
# Iniciar con un bigrama de prueba
seed_text = "palabras"
predicted_sentence = generate_sentence(seed_text, model, tokenizer, total_words, max_word

print(f"Oración generada: {predicted_sentence}")
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 46ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 37ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 32ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step
Oración generada: Aureliano buendía no había de su padre lo llevó a la
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step
Oración generada: palabras de su padre lo llevó a la aldea e implacable
```