Laboratorio 4

Sean bienvenidos de nuevo al laboratorio 4 de Deep Learning y Sistemas Inteligentes. Así como en los laboratorios pasados, espero que esta ejercitación les sirva para consolidar sus conocimientos en el tema de Encoder-Decoder y AutoEnconders.

Para este laboratorio estaremos usando una herramienta para Jupyter Notebooks que facilitará la calificación, no solo asegurándo que ustedes tengan una nota pronto sino también mostrandoles su nota final al terminar el laboratorio.

Espero que esta vez si se muestren los *marks*. De nuevo me discupo si algo no sale bien, seguiremos mejorando conforme vayamos iterando. Siempre pido su comprensión y colaboración si algo no funciona como debería.

Al igual que en el laboratorio pasado, estaremos usando la librería de Dr John Williamson et al de la University of Glasgow, además de ciertas piezas de código de Dr Bjorn Jensen de su curso de Introduction to Data Science and System de la University of Glasgow para la visualización de sus calificaciones.

NOTA: Ahora tambien hay una tercera dependecia que se necesita instalar. Ver la celda de abajo por favor

In [ ]:

```
# Una vez instalada la librería por favor, recuerden volverla a comentar.
#!pip install -U --force-reinstall --no-cache https://github.com/johnhw/jhwutils/zipball/master
#!pip install scikit-image
#!pip install -U --force-reinstall --no-cache https://github.com/AlbertS789/lautils/zipball/master
```

```
Collecting https://github.com/johnhw/jhwutils/zipball/master
  Downloading https://github.com/johnhw/jhwutils/zipball/master
     - 119.1 kB 326.0 kB/s 0:00:00
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Preparing metadata (pyproject.toml) ... done
Building wheels for collected packages: jhwutils
  Building wheel for jhwutils (pyproject.toml) ... done
  Created wheel for jhwutils: filename=jhwutils-1.3-py3-none-any.whl size=41911 sha256=8ce6a9b3a83923bc5ddde68412a48a1b1bbe34c9a94c946abc6896726aeb8bf3
  Stored in directory: /tmp/pip-ephem-wheel-cache-5sxninkb/wheels/c9/d0/2e/946a586bab0de84a4ee2b053e8d50eb28d56d8556f3ebefa84
Successfully built jhwutils
Installing collected packages: jhwutils
Successfully installed jhwutils-1.3

[notice] A new release of pip is available: 25.1.1 -> 25.2
[notice] To update, run: pip install --upgrade pip
Collecting scikit-image
  Using cached scikit_image-0.25.2-cp313-cp313-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (14 kB)
Requirement already satisfied: numpy>=1.24 in ./.venv/lib/python3.13/site-packages (from scikit-image) (2.3.2)
Requirement already satisfied: scipy>=1.11.4 in ./.venv/lib/python3.13/site-packages (from scikit-image) (1.16.1)
Requirement already satisfied: networkx>=3.0 in ./.venv/lib/python3.13/site-packages (from scikit-image) (3.5)
Requirement already satisfied: pillow>=10.1 in ./.venv/lib/python3.13/site-packages (from scikit-image) (11.3.0)
Collecting imageio!=2.35.0,>=2.33 (from scikit-image)
  Using cached imageio-2.37.0-py3-none-any.whl.metadata (5.2 kB)
Collecting tifffile>=2022.8.12 (from scikit-image)
  Using cached tifffile-2025.6.11-py3-none-any.whl.metadata (32 kB)
Requirement already satisfied: packaging>=21 in ./.venv/lib/python3.13/site-packages (from scikit-image) (25.0)
Collecting lazy-loader>=0.4 (from scikit-image)
  Using cached lazy_loader-0.4-py3-none-any.whl.metadata (7.6 kB)
```

```
Using cached scikit_image-0.25.2-cp313-cp313-manylinux_2_17_x86_64.manylinux2014_x86_64.w
hl (15.0 MB)
Using cached imageio-2.37.0-py3-none-any.whl (315 kB)
Using cached lazy_loader-0.4-py3-none-any.whl (12 kB)
Using cached tifffile-2025.6.11-py3-none-any.whl (230 kB)
Installing collected packages: tifffile, lazy-loader, imageio, scikit-image
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 4/4 [scikit-image][0m [scikit-image]
Successfully installed imageio-2.37.0 lazy-loader-0.4 scikit-image-0.25.2 tifffile-2025.6
.11

[notice] A new release of pip is available: 25.1.1 -> 25.2
[notice] To update, run: pip install --upgrade pip
Collecting https://github.com/AlbertS789/lautils/zipball/master
  Downloading https://github.com/AlbertS789/lautils/zipball/master
     - 4.2 kB ? 0:00:00[0m
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Preparing metadata (pyproject.toml) ... done
Building wheels for collected packages: lautils
  Building wheel for lautils (pyproject.toml) ... done
  Created wheel for lautils: filename=lautils-1.0-py3-none-any.whl size=2882 sha256=bd5fc
58b03d5d25dd85f3ab27f0a516b12a25d4070868a0833b51303bdfdb1aa
  Stored in directory: /tmp/pip-ephem-wheel-cache-fopq159g/wheels/36/c3/e8/8587120370e8cd
4fdbdd8f43ff303dcac80f115c2696d2447f
Successfully built lautils
Installing collected packages: lautils
Successfully installed lautils-1.0

[notice] A new release of pip is available: 25.1.1 -> 25.2
[notice] To update, run: pip install --upgrade pip
```

In [53]:

```
import numpy as np
import copy
import matplotlib.pyplot as plt
import scipy
from PIL import Image
import os
from collections import defaultdict

#from IPython import display
#from base64 import b64decode


# Other imports
from unittest.mock import patch
from uuid import getnode as get_mac

from jhwutils.checkarr import array_hash, check_hash, check_scalar, check_string, array_
hash, _check_scalar
import jhwutils.image_audio as ia
import jhwutils.tick as tick
from lautils.gradeutils import new_representation, hex_to_float, compare_numbers, compare
_lists_by_percentage, calculate_coincidences_percentage

###
tick.reset_marks()

%matplotlib inline
```

In [ ]:

```
# Seeds
seed_ = 2023
np.random.seed(seed_)
```

In [ ]:

```
# Celda escondida para utlidades necesarias, por favor NO edite esta celda
```

Parte 2 - Encoder - Decoder

Créditos: La segunda parte de este laboratorio está tomado y basado en uno de los repositorios de Ben Trevett

En esta ocasión vamos a centrarnos en una arquitectura Sequence to Sequence (Seq2Seq), entonces estaremos desarrollando un modelo que nos ayude a traducir de alemán a inglés. Tomaremos como base el paper [Sequence to Sequence Learning with Neural Networks]. Recuerden que a pesar que esto es para frases/oraciones, los conceptos pueden ser aplicados para otras arquitecturas similares.

IMPORTANTE: Recuerden usar virtual enviroments debido a que estaremos usando versiones viejas de la librerías. ¿Por qué? Las librerías eran un poco más explícitas que sus versiones más recientes. A continuación se dejan los comandos para la instalación de las más importantes

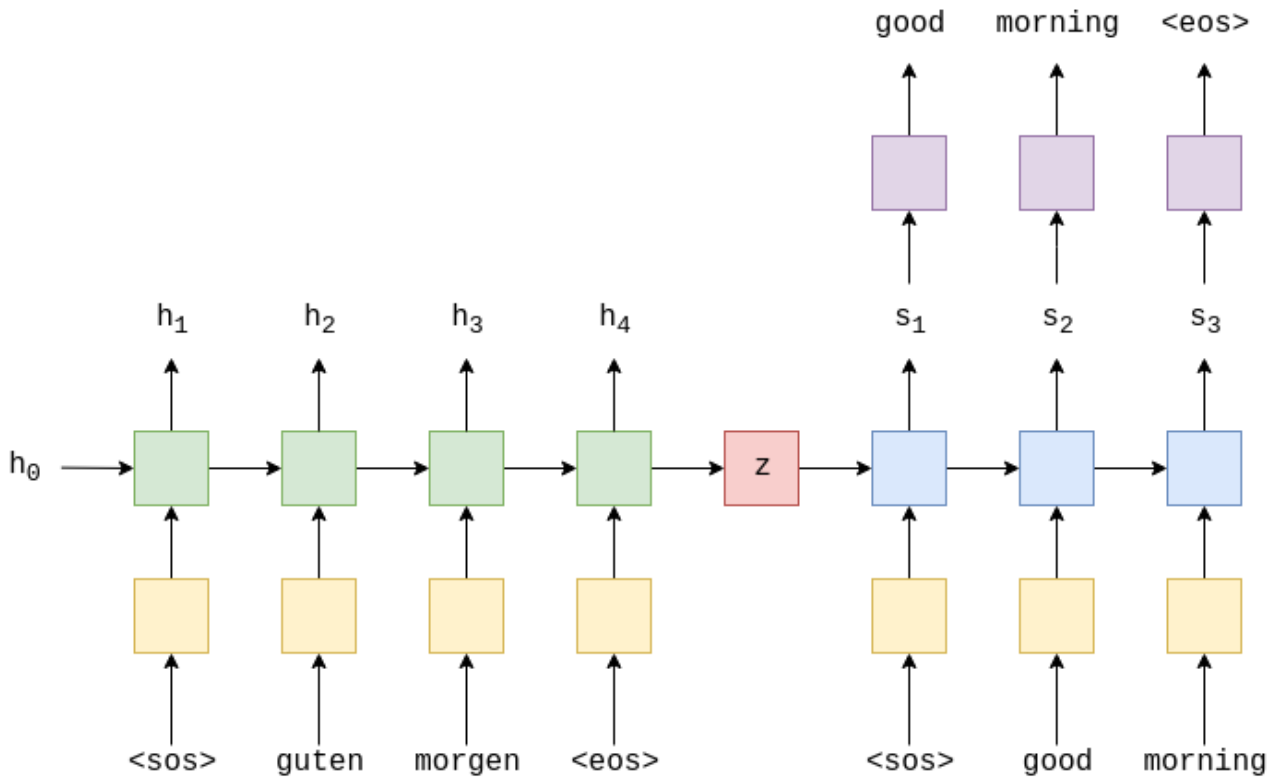
```
pip install -U torch==1.9.0+cu111 -f  https://download.pytorch.org/whl/cu111/torch_stable.html
pip install -U torchtext==0.10.0
```

El primer comando instalará la librería de PyTorch con CUDA 11.1 El segundo, instala TorchText en una versión donde la formulación del vocabulario para training, test y validation era más claro (esta es la principal por la que estamos usando esta versiones).

Introducción

Los modelos más comunes seq2seq son los modelos *encoder-decoder*, los cuales usan una RNN para encodear el input y llevarlo a un solo vector. En este laboratorio nos estaremos refiriendo a dicho vector como *vector contexto*. Pensemos sobre el vector contexto como un ser abstracto que representa una frase completa. Este vector es luego decodeado por una segunda RNN, que aprende a generar la frase target (output) deseada al generar palabra por palabra.

Consideren la siguiente ilustración para representar el proceso que estaremos realizando



Crédito de imagen al autor, imagen tomada de "Sequence to Sequence Learning with Neural Networks" de Ben Trevett

Noten como la frase de input "guten morgen", se pasa a través de una capa de embedding (cuadros amarillos) y luego entra en los encodeadores (cuadros verdes). En esta ocasión agregamos un token de "start of sequence" (`<sos>`) al inicio de la frase, además de un token de "end of sequence" (`<eos>`) al final de la oración. Vean como en cada paso, la entrada del encoder RNN es tanto la representación embedding e de la palabra actual $e(x_t)$, así como el estado oculto del paso anterior h_{t-1}, y el encoder genera un nuevo hidden state h_t. Entonces, podemos pensar en el hidden state como una representación vectorial de la oración hasta ese momento. La

RNN se puede representar como una función de tanto $e(x_t)$ y h_{t-1}

$$h_t = \text{EncoderRNN}(e(x_t), h_{t-1})$$

Por favor noten que estamos usando el termino RNN de forma general en este contexto, puede ser cualquier arquitectura como LSTM o GRU.

Entonces estaremos trabajando con una secuencia como $X = \{x_1, x_2, \ldots, x_T\}$, donde $x_1 =< sos >$, $x_2 = guten$, y así consecutivamente. El hidden state inicial h_0 es usualmente iniciado con ceros o con algún parametro pre-aprendido.

Una vez la palabra final X_T ha pasado en la RNN a través de la embedding layer, usamos el hidden state final h_I como vector de contexto. Es decir, $h_T = z$. El cual será la representación vectorial de toda la oración.

Ahora que tenemos nuestro vector de contexto z, podemos empezar a decodear para obtener la oración target, "good morning". De nuevo, agregamos los tokens de inicio y fin de la secuencia de nuestra oración target. En cada paso, el input al decoder RNN (cuadros azules de la imagen) es la versión embedding d de la palabra actual $d(y_t)$ así como también el hidden state del paso previo s_{t-1}m donde el hidden state del decoder incial s_0 es el vector de contexto $s_0 = z = h_T$, es decir, el hidden state decoder es el último hidden state encoder. Por ende, simlar al encoder, podemos representarlo como:

$$s_t = \text{DecoderRNN}(d(y_t), s_{t-1})$$

A pesar que el input embeeding layer e y el target embedding layer d están representados como cuadros amarillos en la imagen, como dijimos en clase, estas son dos embedding layers diferentes con sus propios parametros.

En el decoder, necestamos ir del hidden state a la palabra actual, por ello en cada paso usamos s_t para predecir (a traves de pasarlo en una layer lineal, mostrada como cuadros morados) lo que se cree que es la siguiente palabra en la secuencia $\hat{y}_t$

$$\hat{y}_t = f(s_t)$$

Las palabras en el decoder son siempre generadas una después de la otra, con una por paso. Siempre usamos `<sos>` para el primer input del decodr y_1 y algumas veces usamos la palabra predicha por nuestro decoder, $\hat{y}_{t-1}$. Que, como mencionamos en clase, se le llama *teacher forcing*.

Cuando estamos entrenando o probando nuestro modelo, siempre sabemso cuantas palabras hay en nuestra secuencia target, entonces nos detenemos de generar palabras una vez alcanzamos esa cantidad. Durante las fases de inferencia (uso del modelo en la "vida real") seguimos generando palabras hasta que el modelo genere un token `<eos>` o después de una cierta cantidad de palabras dada. (Esto tambien lo mencionamoos en clase, es solo para refrescar los conceptos)

Una vez tengamos nuestra secuencia target predicha $\hat{Y} = \{\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_T\}$, la comparamos contra nuestra secuencia target real. $Y = \{y_1, y_2, \ldots, y_T\}$, para calcular la perdida. Usamos esta pérdida para actualizar los parámetros del modelo, como bien hemos hecho en otras ocasiones.

Preparación de Data

Es momento de ponernos a manos a la obra. Estaremos programando nuestro modelo usando PyTorch y usando torchtext para ayudarnos a hacer todo el pre-procesamiento necesario. Ahora usaremos spaCy para ayudarnos en la tokenización de los datos

In [28]:

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

```
from torchtext.datasets import Multi30k

train_url = "https://raw.githubusercontent.com/neychev/small_DL_repo/master/datasets/Multi30k/training.tar.gz"
val_url = "https://raw.githubusercontent.com/neychev/small_DL_repo/master/datasets/Multi30k/validation.tar.gz"
test_url = "https://raw.githubusercontent.com/neychev/small_DL_repo/master/datasets/Multi30k/mmt16_task1_test.tar.gz"

# Update the URLs in the Multi30k module
Multi30k.urls = (train_url, val_url, test_url)

from torchtext.data import Field, BucketIterator

import spacy
import numpy as np

import random
import math
import time
```

Colocamos las semillas para tener resultados consistentes.

In [29]:

```
SEED = 1234

random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.cuda.manual_seed(SEED)
torch.backends.cudnn.deterministic = True
```

Ahora, necesitamos crear un tokenizador. Un tokenizador es una función que ayudará a convertir un string de alguna frase u oración en una lista de tokens individuales. Consideren que en una frase como "good morning!" se tienen tres tokens, siendo cada uno "good", "morning" y "!", noten que a pesar que el signo de admiracion no se considera una palabra, sí se considera como un token.

Para la creación de nuestro tokenizador nos apoyaremos en spaCy, en este caso necesitamos los paquetes de aleman e inglés (se nombran abajo).

Para instalar spaCy necesitarán ejecutar en la cmd

```
pip install spacy
python -m spacy download en_core_web_sm
python -m spacy download de_core_news_sm
```

IMPORTANTE: Recuerden usar virtual environments de Python, debido a que este laboratorio usa algunas librerías deprecadas, que como se explicó previamente, se hizo de este modo para ser más explícito el aprendizaje.

Regresando al tema del tokenizer, primero cargaremos las dos versiones para los diferentes idiomas con los que estamos trabajando.

Despues, crearemos unas funciones de tokenización. Estas pueden ser pasadas a TorchText y tomarán una oración y regresara la oración como una lista de tokens.

Cabe la pena mencionar que en el paper que estamos tomando de base, ellos encontrarón util el revertir el orden del input dado que se cree que introducía varias dependencias a corto plazo en los datos que facilitan mucho el problema de optimización.

Más adelante, usaremos `Field` (que actualmente está deprecado :( ) para manejar como la data debería ser procesada. Después, seteamos el parametro `tokenize` como función para cada caso. El aleman será el `SRC` y el inglés será el `TRG`. Además también se agrega el token para inicio y fin de la secuencia, además que convertirá todo en lowercase.

In [30]:

```
spacy_de = spacy.load('de_core_news_sm')
spacy_en = spacy.load('en_core_web_sm')
```

In [31]:

```
def tokenize_de(text):
    """
    Tokenizes German text from a string into a list of strings (tokens) and reverses it
    """
    return [tok.text for tok in spacy_de.tokenizer(text)][::-1]

def tokenize_en(text):
    """
    Tokenizes English text from a string into a list of strings (tokens)
    """
    return [tok.text for tok in spacy_en.tokenizer(text)]
```

In [32]:

```
SRC = Field(tokenize = tokenize_de,
            init_token = '<sos>',
            eos_token = '<eos>',
            lower = True)

TRG = Field(tokenize = tokenize_en,
            init_token = '<sos>',
            eos_token = '<eos>',
            lower = True)
```

Ahora, debemos descargar el dataset. Para este caso estaremos usando el dataset llamado Multi30k. Este tiene aproximadamente 30K frases en inglés, aleman y francés, cada uno tiene alrededor de 12 palabras por frase.

Además noten que `exts` especifica cual lenguage se debe usar como source y target, y `fields` da cuales campos usar para el source y target.

In [33]:

```
# Si esta celda falla, vayan a 'Home → datasets → Multi30k → multi30k' y cambien todos lo
s archivos de nombre 'test' a 'test2016'.
# No toquen la extensión de ninguno.
train_data, valid_data, test_data = Multi30k.splits(exts = ('.de', '.en'),
                                                    fields = (SRC, TRG),
                                                    root = './datasets/Multi30k')
```

In [34]:

```
print(f"Numero de observaciones de training: {len(train_data.examples)}")
print(f"Numero de observaciones en validation: {len(valid_data.examples)}")
print(f"Numero de observaciones en test: {len(test_data.examples)}")
```

```
Numero de observaciones de training: 29000
Numero de observaciones en validation: 1014
Numero de observaciones en test: 1000
```

In [35]:

```
SRC.build_vocab(train_data, min_freq = 2)
TRG.build_vocab(train_data, min_freq = 2)
```

In [36]:

```
print(vars(train_data.examples[0]))
```

```
{'src': ['.', 'büsche', 'vieler', 'nähe', 'der', 'in', 'freien', 'im', 'sind', 'männer',
'weiße', 'junge', 'zwei'], 'trg': ['two', 'young', ',', 'white', 'males', 'are', 'outside
', 'near', 'many', 'bushes', '.']}
```

Observen como el punto está al comienzo de la oración en alemán (src), por lo que parece que la oración se invirtió correctamente.

Ahora, construiremos el vocabulario para los idiomas de source y de target. El vocabulario se utiliza para asociar cada token único con un índice (un número entero). Los vocabularios de los idiomas de origen y de destino son distintos.

Usando el argumento `min_freq`, solo permitimos que aparezcan en nuestro vocabulario tokens que aparecen al menos 2 veces. Los tokens que aparecen solo una vez se convierten en un token desconocido `<unk>`.

Es importante tener en cuenta que nuestro vocabulario solo debe construirse a partir del conjunto de entrenamiento y no del conjunto de validación/test. Esto evita la "fuga de información" en nuestro modelo, dándonos puntajes de validación/prueba inflados artificialmente.

In [37]:

```
print(f"Unique tokens in source (de) vocabulary: {len(SRC.vocab)}")
print(f"Unique tokens in target (en) vocabulary: {len(TRG.vocab)}")
```

```
Unique tokens in source (de) vocabulary: 7853
Unique tokens in target (en) vocabulary: 5893
```

El paso final de preparar los datos es crear los iteradores. Estos se pueden iterar para devolver un lote de datos que tendrá un atributo `src` (los tensores de PyTorch que contienen un lote de oraciones de origen numeradas) y un atributo `trg` (los tensores de PyTorch que contienen un batch de oraciones de destino numeradas). "Numericalized" es solo una forma elegante de decir que se han convertido de una secuencia de tokens legibles a una secuencia de índices correspondientes, usando el vocabulario.

También necesitamos definir un dispositivo `torch.device`. Esto se usa para indicarle a torchText que coloque o no los tensores en la GPU. Usamos la función `torch.cuda.is_available()`, que devolverá True si se detecta una GPU en nuestra computadora. Pasamos este dispositivo al iterador.

Cuando obtenemos un lote de ejemplos usando un iterador, debemos asegurarnos de que todas las oraciones de origen tengan la misma longitud, al igual que las oraciones de destino. ¡Afortunadamente, los iteradores de torchText manejan esto por nosotros!

Usamos un `BucketIterator` en lugar del `Iterador` estándar, ya que crea lotes de tal manera que minimiza la cantidad de padding en las oraciones de origen y de destino.

In [38]:

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(device)
```

```
cpu
```

In [39]:

```
BATCH_SIZE = 128

train_iterator, valid_iterator, test_iterator = BucketIterator.splits(
    (train_data, valid_data, test_data),
    batch_size = BATCH_SIZE,
    device = device)
```

Construyendo el Modelo Seq2Seq

Vamos a definir nuestro modelo en tres partes, el encoder, el decoder y el modelo Seq2Seq. Este ultimo encapsulará el proceso y transferencia entre los primeros dos.

Encoder

Primero, el encoder, es un LSTM de 2 capas. El paper que estamos implementando usa un LSTM de 4 capas, pero en favor del tiempo de entrenamiento lo reducimos a 2 capas. El concepto de RNN multicapa es fácil de expandir de 2 a 4 capas.

Para un RNN multicapa, la oración de entrada, X, después de ser embeddida va a la primera capa (inferior) del RNN y los estados ocultos, $H = \{h_1, h_2, \ldots, h_T\}$, la salida de esta capa se utiliza como entrada a la RNN en la capa superior. Así, representando cada capa con un superíndice, los hidden states en la primera capa vienen dados por:

$$h_t^1 = \text{EncoderRNN}^1(e(x_t), h_{t-1}^1)$$

Las hidden states en la segunda layer son dadas por:

$$h_t^2 = \text{EncoderRNN}^2(h_t^1, h_{t-1}^2)$$

El uso de un RNN multicapa también significa que también necesitaremos un hidden state inicial como entrada por capa, h_0^l, y también generaremos un vector de contexto por capa, z^l.

Si desean repasar un poco sobre LSTM pueden consultar este [enlance] (https://colah.github.io/posts/2015-08-Understanding-LSTMs/) Para este laboratorio, es suficiente que recuerden que lo que necesitamos saber es los LSTM, en lugar de simplemente tomar un estado oculto y devolver un nuevo estado oculto por paso de tiempo, también toman y devuelven un *estado de celda*, c_t, por paso de tiempo.

$$h_t = \text{RNN}(e(x_t), h_{t-1})$$
$$(h_t, c_t) = \text{LSTM}(e(x_t), h_{t-1}, c_{t-1})$$

Podemos pensar en c_t como otro tipo de hidden state. Similar a h_0^l, c_0^l se inicializará en un tensor de ceros. Además, nuestro vector de contexto ahora será tanto el hidden state final como el estado de celda final, es decir, $z^l = (h_T^l, c_T^l)$.

Al extender nuestras ecuaciones multicapa a LSTM, obtenemos:

$$(h_t^1, c_t^1) = \text{EncoderLSTM}^1(e(x_t), (h_{t-1}^1, c_{t-1}^1))$$
$$(h_t^2, c_t^2) = \text{EncoderLSTM}^2(h_t^1, (h_{t-1}^2, c_{t-1}^2))$$

Observen cómo solo nuestro hidden state de la primera capa se pasa como entrada a la segunda capa, y no el estado de la celda.

Así que nuestro codificador se parece a esto:

IMAGEN

Creamos esto en el código creando un módulo `Encoder`, que requiere que heredemos de `torch.nn.Module` y usemos `super().__init__()` como un código repetitivo. El codificador toma los siguientes argumentos:

- `input_dim` es el tamaño/dimensionalidad de los vectores one-hot que se ingresarán al codificador. Esto es igual al tamaño del vocabulario de entrada (fuente).
- `emb_dim` es la dimensionalidad de la capa de embedding. Esta capa convierte los vectores one-hot en vectores densos con dimensiones `emb_dim`.
- `hid_dim` es la dimensionalidad de los estados ocultos y de celda.
- `n_layers` es el número de capas en el RNN.
- `dropout` es la cantidad de abandono a utilizar. Este es un parámetro de regularización para evitar el overfitting. Consulte [aqui] (https://www.coursera.org/lecture/deep-neural-network/understanding-dropout-YaGbR) para obtener más detalles sobre dropout.

No vamos a discutir la capa de embedding en detalle durante aqui pues ya lo hicimos previamente. Todo lo que necesitamos saber es que hay un paso antes de que las palabras (técnicamente, los índices de las palabras) pasen al RNN, donde las palabras se transforman en vectores. Para leer más sobre embedding de palabras, consulten estos artículos: 1, 2, 3, 4.

La capa de embedding se crea usando `nn.Embedding`, el LSTM con `nn.LSTM` y una capa de dropout con `nn.Dropout`. Consulten la [documentación de PyTorch] (https://pytorch.org/docs/stable/nn.html) para obtener más información al respecto.

Una cosa a tener en cuenta es que el argumento `dropout` para el LSTM es cuánto dropout aplicar entre las capas de un RNN multicapa, es decir, entre la salida de estados ocultos de la capa l y esos mismos estados ocultos que se utilizan para el entrada de la capa $l + 1$.

En el método `forward`, pasamos la oración fuente, X, que se convierte en vectores densos usando la capa

`embedding`, y luego se aplica el dropout. Estos embedding luego se pasan a la RNN. A medida que pasamos una secuencia completa a la RNN, ¡automáticamente hará el cálculo recurrente de los estados ocultos en toda la secuencia por nosotros! Tenga en cuenta que no pasamos un estado inicial oculto o de celda al RNN. Esto se debe a que, como se indica en la documentación, si no se pasa ningún estado de celda/oculto a la RNN, crea automáticamente un estado inicial de celda/oculto como un tensor de ceros.

El RNN devuelve: `outputs` (el hidden state de la capa superior para cada paso de tiempo), `hidden` (el hidden state final para cada capa, h_T, apiladas una encima de la otra) y `cell` (la estado de celda final para cada capa, c_T, apilados uno encima del otro).

Como solo necesitamos los hidden state y de celda finales (para hacer nuestro vector de contexto), `forward` solo devuelve `hidden` y `cell`.

Los tamaños de cada uno de los tensores se dejan como comentarios en el código. En esta implementación, `n_directions` siempre será 1, sin embargo, tengan en cuenta que los RNN bidireccionales (cubiertos en el tutorial 3) tendrán `n_directions` como 2.

In [40]:

```
class Encoder(nn.Module):
    def __init__(self, input_dim, emb_dim, hid_dim, n_layers, dropout):
        super().__init__()

        self.hid_dim = hid_dim
        self.n_layers = n_layers

        # Capa de embedding que convierte índices de tokens en vectores densos
        self.embedding = nn.Embedding(input_dim, emb_dim)

        # LSTM multicapa con dropout entre capas
        self.rnn = nn.LSTM(emb_dim, hid_dim, n_layers, dropout=dropout)

        self.dropout = nn.Dropout(dropout)

    def forward(self, src):

        #src = [src len, batch size]

        embedded = self.dropout(self.embedding(src))

        #embedded = [src len, batch size, emb dim]

        outputs, (hidden, cell) = self.rnn(embedded)

        #outputs = [src len, batch size, hid dim * n directions]
        #hidden = [n layers * n directions, batch size, hid dim]
        #cell = [n layers * n directions, batch size, hid dim]

        #outputs are always from the top hidden layer

        return hidden, cell
```

Decoder

Ahora pasaremos a construir el decoder, el cual también será una 2-layer (4 en el paper) LSTM.

La clase `Decoder` hace un solo paso de decodificación, es decir, genera un solo token por paso. La primera capa recibirá un hidden state y de celda del paso de tiempo anterior, (s_{t-1}^1, c_{t-1}^1), y lo alimenta a través del LSTM con el token incrustado actual, y_t, para producir un nuevo hidden state y de celda, (s_t^1, c_t^1). Las capas subsiguientes usarán el estado oculto de la capa inferior, s_t^{l-1}, y los estados ocultos y de celda anteriores de su capa, (s_{t-1}^l, c_{t-1}^l). Esto proporciona ecuaciones muy similares a las del codificador.

$$(s_t^1, c_t^1)$$
$$= \text{DecoderLSTM}^1$$
$$(d(y_t), (s_{t-1}^1,$$

$$
\begin{aligned}
&(s_t^1, c_t^1) \\
&= \text{DecoderLSTM}^1(s_t^1, (s_{t-1}^1, \\
&c_{t-1}^1)) \\
&(s_t^2, c_t^2) \\
&= \text{DecoderLSTM}^2 \\
&(s_t^1, (s_{t-1}^2, \\
&c_{t-1}^2))
\end{aligned}
$$

Recuerde que los estados iniciales ocultos y de celda de nuestro decoder son nuestros vectores de contexto, que son los estados finales ocultos y de celda de nuestro decoder de la misma capa, es decir, $(s_0^l, c_0^l) = z^l$.

$$= (h_T^l, c_T^l)$$

Luego pasamos el hidden state desde la capa superior del RNN, s_t^L, a través de una capa lineal, f, para hacer una predicción de cuál será el siguiente token en la secuencia de destino (salida). debería ser, $\hat{y}_{t+1}$.

$$
\begin{aligned}
&\sombrero{y}_{t+1} \\
&= f(s_t^L)
\end{aligned}
$$

Los argumentos y la inicialización son similares a la clase `Encoder`, excepto que ahora tenemos un `output_dim` que es el tamaño del vocabulario para la salida/objetivo. También está la adición de la capa 'Lineal', utilizada para hacer las predicciones desde el hidden state de la capa superior.

Dentro del método `forward`, aceptamos un batch de tokens de entrada, hidden state anteriores y estados de celda anteriores. Como solo estamos decodificando un token a la vez, los tokens de entrada siempre tendrán una longitud de secuencia de 1. "Aflojamos" los tokens de entrada para agregar una dimensión de longitud de oración de 1. Luego, de forma similar al encoder, pasamos a través de una capa de embedding y aplicamos dropout. Este batch de tokens embeddidos luego se pasa al RNN con los estados ocultos y de celda anteriores. Esto produce una "salida" (hidden state de la capa superior de la RNN), un nuevo "hidden state" (uno para cada capa, apilados uno encima del otro) y una nueva "celda". estado (también uno por capa, apilados uno encima del otro). Luego pasamos la `salida` (después de deshacernos de la dimensión de longitud de la oración) a través de la capa lineal para recibir nuestra `predicción`. Luego devolvemos la `predicción`, el nuevo hidden state y el nuevo estado `celular`.

Nota: como siempre tenemos una longitud de secuencia de 1, podríamos usar `nn.LSTMCell`, en lugar de `nn.LSTM`, ya que está diseñado para manejar un lote de entradas que no son necesariamente en una secuencia. `nn.LSTMCell` es solo una sola celda y `nn.LSTM` es un envoltorio alrededor de múltiples celdas potenciales. Usando `nn.LSTMCell` en este caso significaría que no tenemos que `descomprimir` para agregar una dimensión de longitud de secuencia falsa, pero necesitaríamos un `nn.LSTMCell` por capa en el decoder y para asegurar que cada `nn.LSTMCell` recibe el hidden state inicial correcto del codificador. Todo esto hace que el código sea menos conciso, de ahí la decisión de seguir con el `nn.LSTM` regular.

In [41]:

```
class Decoder(nn.Module):
    def __init__(self, output_dim, emb_dim, hid_dim, n_layers, dropout):
        super().__init__()

        # Guardar dimensiones y configuración del modelo
        self.output_dim = output_dim
        self.hid_dim = hid_dim
        self.n_layers = n_layers

        # Capa de embedding para el vocabulario target (inglés)
        self.embedding = nn.Embedding(output_dim, emb_dim)

        # LSTM multicapa que recibe embeddings y estados previos
        self.rnn = nn.LSTM(emb_dim, hid_dim, n_layers, dropout=dropout)

        self.fc_out = nn.Linear(hid_dim, output_dim)

        self.dropout = nn.Dropout(dropout)

    def forward(self, input, hidden, cell):
```

```
        #input = [batch size]
        #hidden = [n layers * n directions, batch size, hid dim]
        #cell = [n layers * n directions, batch size, hid dim]

        #n directions in the decoder will both always be 1, therefore:
        #hidden = [n layers, batch size, hid dim]
        #context = [n layers, batch size, hid dim]

        input = input.unsqueeze(0)

        #input = [1, batch size]

        embedded = self.dropout(self.embedding(input))

        #embedded = [1, batch size, emb dim]

        output, (hidden, cell) = self.rnn(embedded, (hidden, cell))

        #output = [seq len, batch size, hid dim * n directions]
        #hidden = [n layers * n directions, batch size, hid dim]
        #cell = [n layers * n directions, batch size, hid dim]

        #seq len and n directions will always be 1 in the decoder, therefore:
        #output = [1, batch size, hid dim]
        #hidden = [n layers, batch size, hid dim]
        #cell = [n layers, batch size, hid dim]

        prediction = self.fc_out(output.squeeze(0))

        #prediction = [batch size, output dim]

        return prediction, hidden, cell
```

Seq2Seq

Para la parte final de la implementación, implementaremos el modelo seq2seq. Esto manejará:

- **recibir la oración de entrada/fuente**
- **usar el encoder para producir los vectores de contexto**
- **usar el decoder para producir la salida predicha/oración objetivo**

Nuestro modelo completo se verá así:

El modelo `Seq2Seq` incluye un `Encoder`, un `Decoder` y un `dispositivo` (usado para colocar tensores en la GPU, si existe).

Para esta implementación, debemos asegurarnos de que el número de capas y las dimensiones ocultas (y de celda) sean iguales en el 'Encoder' y 'Decoder'. Este no es siempre el caso, no necesariamente necesitamos la misma cantidad de capas o los mismos tamaños de dimensiones ocultas en un modelo de sequence to sequence. Sin embargo, si hiciéramos algo como tener un número diferente de capas, tendríamos que tomar decisiones sobre cómo manejar esto. Por ejemplo, si nuestro encoder tiene 2 capas y nuestro decoder solo tiene 1, ¿cómo se maneja esto? ¿Promediamos los dos vectores de contexto generados por el decoder? ¿Pasamos ambos por una capa lineal? ¿Solo usamos el vector de contexto de la capa más alta? Etc.

Nuestro método "forward" toma la oración fuente, la oración objetivo y un ratio de teacher-forcing. El ratio de teacher-forcing se usa cuando entrenamos nuestro modelo. Al decodificar, en cada paso, predeciremos cuál será el próximo token en la secuencia de destino de los tokens anteriores decodificados, $\hat{y}_{t+1} = f(s_t^L)$. Con una probabilidad igual a la tasa de teacher forcing (`teacher_forcing_ratio`), utilizaremos el siguiente token real de la secuencia como entrada al decoder durante el siguiente paso. Sin embargo, con probabilidad `1 - Teacher_forcing_ratio`, usaremos el token que el modelo predijo como la próxima entrada al modelo, incluso si no coincide con el siguiente token real en la secuencia.

Lo primero que hacemos en el método `forward` es crear un tensor `outputs` que almacenará todas nuestras predicciones, $\hat{Y}$.

Luego alimentamos la oración de entrada/fuente, `src`, en el encoder y recibimos los estados ocultos y de celda finales.

La primera entrada al decoder es el token de inicio de secuencia (`<sos>`). Como nuestro tensor `trg` ya tiene el token `<sos>` agregado (desde cuando definimos el `init_token` en nuestro campo `TRG`) obtenemos nuestro y_1 cortándolo. Sabemos qué tan largas deben ser nuestras oraciones de destino (`max_len`), por lo que las repetimos muchas veces. El último token ingresado en el decoder es el **antes** del token `<eos>` - el `<eos>` el token nunca se ingresa en el decoder.

Durante cada iteración del ciclo, nosotros:

- pasar la entrada, los estados de celda anteriores ocultos y anteriores (y_t, s_{t-1}, c_{t-1}) al decoder
- recibir una predicción, el siguiente estado oculto y el siguiente estado de celda ($\hat{y}_{t+1}, s_t, c_t$) del decoder
- colocar nuestra predicción, $\hat{y}_{t+1}$/`output` en nuestro tensor de predicciones, $\hat{Y}$/`outputs`
- decidir si vamos a "fuerza de maestros" o no
  - si lo hacemos, la siguiente 'entrada' es el siguiente token de verdad fundamental en la secuencia, y_{t+1}/`trg[t]`
  - si no lo hacemos, la siguiente `entrada` es el siguiente token predicho en la secuencia, $\hat{y}_{t+1}$/`top1`, que obtenemos al hacer un `argmax` sobre el tensor de salida

Una vez que hemos hecho todas nuestras predicciones, devolvemos nuestro tensor lleno de predicciones, $\hat{Y}$/`outputs`.

Nota: nuestro ccilo decodificador comienza en 1, no en 0. Esto significa que el elemento 0 de nuestro tensor de `salidas` sigue siendo todo ceros. Así que nuestras `trg` y `outputs` se parecen a:

$$
\begin{alinear*}
\text{trg} = [<sos>, &y_1, y_2, y_3, <eos>]\\
\text{resultados} = [0, &\hat{y}_1, \hat{y}_2, \hat{y}_3, <eos>]
\end{align*}
$$

Posteriormente cuando calculamos la pérdida, cortamos el primer elemento de cada tensor para obtener:

$$
\begin{alinear*}
\text{trg} = [&y_1, y_2, y_3, <eos>]\\
\text{salidas} = [&\hat{y}_1, \hat{y}_2, \hat{y}_3, <eos>]
\end{align*}
$$

In [42]:

```
class Seq2Seq(nn.Module):
    def __init__(self, encoder, decoder, device):
        super().__init__()

        self.encoder = encoder
        self.decoder = decoder
        self.device = device

        assert encoder.hid_dim == decoder.hid_dim, \
            "Hidden dimensions of encoder and decoder must be equal!"
        assert encoder.n_layers == decoder.n_layers, \
            "Encoder and decoder must have equal number of layers!"

    def forward(self, src, trg, teacher_forcing_ratio = 0.5):

        #src = [src len, batch size]
        #trg = [trg len, batch size]
        #teacher_forcing_ratio is probability to use teacher forcing
        #e.g. if teacher_forcing_ratio is 0.75 we use ground-truth inputs 75% of the time

        batch_size = trg.shape[1]
        trg_len = trg.shape[0]
        trg_vocab_size = self.decoder.output_dim

        #tensor to store decoder outputs
        outputs = torch.zeros(trg_len, batch_size, trg_vocab_size).to(self.device)

        #last hidden state of the encoder is used as the initial hidden state of the deco
```

```
der
        hidden, cell = self.encoder(src)

        #first input to the decoder is the <sos> tokens
        input = trg[0,:]

        for t in range(1, trg_len):

            #insert input token embedding, previous hidden and previous cell states
            #receive output tensor (predictions) and new hidden and cell states

            # Llamada al decoder con el token actual y los estados previos
            output, hidden, cell = self.decoder(input, hidden, cell)

            #place predictions in a tensor holding predictions for each token
            outputs[t] = output

            #decide if we are going to use teacher forcing or not
            teacher_force = random.random() < teacher_forcing_ratio

            #get the highest predicted token from our predictions
            top1 = output.argmax(1)

            #if teacher forcing, use actual next token as next input
            #if not, use predicted token
            input = trg[t] if teacher_force else top1

        return outputs
```

Training Seq2Seq Model

Ahora que tenemos nuestro modelo implementado, podemos comenzar a entrenarlo.

Primero, inicializaremos nuestro modelo. Como se mencionó anteriormente, las dimensiones de entrada y salida están definidas por el tamaño del vocabulario. Las dimensiones de embedding y el dropout del encoder y el decoder pueden ser diferentes, pero el número de capas y el tamaño de los estados ocultos/de celda deben ser los mismos.

Luego definimos el encoder, el decoder y luego nuestro modelo Seq2Seq, que colocamos en el "device".

El siguiente paso es inicializar los pesos de nuestro modelo. En el paper afirman que inicializan todos los pesos a partir de una distribución uniforme entre -0,08 y +0,08, es decir, $\mathcal{U}(-0,08, 0,08)$.

Inicializamos los pesos en PyTorch creando una función que "aplicamos" a nuestro modelo. Al usar `apply`, se llamará a la función `init_weights` en cada módulo y submódulo dentro de nuestro modelo. Para cada módulo, recorremos todos los parámetros y los muestreamos desde una distribución uniforme con `nn.init.uniform_`.

También definimos una función que calculará el número de parámetros entrenables en el modelo.

Definimos nuestro optimizador, que usamos para actualizar nuestros parámetros en el ciclo de entrenamiento. Consulte esta publicación para obtener información sobre diferentes optimizadores. Aquí usaremos a Adam

A continuación, definimos nuestra función de pérdida. La función `CrossEntropyLoss` calcula tanto el log softmax como la log-likelihood negativo de nuestras predicciones.

Nuestra función de pérdida calcula la pérdida promedio por token, sin embargo, al pasar el índice del token `<pad>` como el argumento `ignore_index`, ignoramos la pérdida siempre que el token de destino sea un token de relleno (padding).

In [43]:

```
INPUT_DIM = len(SRC.vocab)
OUTPUT_DIM = len(TRG.vocab)
ENC_EMB_DIM = 256
DEC_EMB_DIM = 256
HID_DIM = 512
```

```
N_LAYERS = 2
ENC_DROPOUT = 0.5
DEC_DROPOUT = 0.5

enc = Encoder(INPUT_DIM, ENC_EMB_DIM, HID_DIM, N_LAYERS, ENC_DROPOUT)
dec = Decoder(OUTPUT_DIM, DEC_EMB_DIM, HID_DIM, N_LAYERS, DEC_DROPOUT)

model = Seq2Seq(enc, dec, device).to(device)
```

In [44]:

```
def init_weights(m):
    for name, param in m.named_parameters():
        nn.init.uniform_(param.data, -0.08, 0.08)

model.apply(init_weights)
```

Out[44]:

```
Seq2Seq(
  (encoder): Encoder(
    (embedding): Embedding(7853, 256)
    (rnn): LSTM(256, 512, num_layers=2, dropout=0.5)
    (dropout): Dropout(p=0.5, inplace=False)
  )
  (decoder): Decoder(
    (embedding): Embedding(5893, 256)
    (rnn): LSTM(256, 512, num_layers=2, dropout=0.5)
    (fc_out): Linear(in_features=512, out_features=5893, bias=True)
    (dropout): Dropout(p=0.5, inplace=False)
  )
)
```

In [45]:

```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

print(f'El modelo tiene {count_parameters(model):,} parametros entrenables')
```

```
El modelo tiene 13,898,501 parametros entrenables
```

In [46]:

```
optimizer = optim.Adam(model.parameters())
```

In [47]:

```
TRG_PAD_IDX = TRG.vocab.stoi[TRG.pad_token]

criterion = nn.CrossEntropyLoss(ignore_index = TRG_PAD_IDX)
```

A continuación, definiremos nuestro ciclo de entrenamiento.

Primero, configuraremos el modelo en "modo de entrenamiento" con `model.train()`. Esto activará el dropout (y batch normalization, que no estamos usando) y luego iterará a través de nuestro iterador de datos.

Como se indicó anteriormente, nuestro ciclo decodificador comienza en 1, no en 0. Esto significa que el elemento 0 de nuestro tensor de "salidas" sigue siendo todo ceros. Así que nuestras `trg` y `outputs` se parecen a:

$$\begin{alinear*}
\text{trg} = [<sos>, &y_1, y_2, y_3, <eos>]\\
\text{resultados} = [0, &\hat{y}_1, \hat{y}_2, \hat{y}_3, <eos>]
\end{align*}$$

Aquí, cuando calculamos la pérdida, cortamos el primer elemento de cada tensor para obtener:

$$\begin{alinear*}
\text{trg} = [&y_1, y_2, y_3, <eos>]\\
\text{salidas} = [&\hat{y}_1, \hat{y}_2, \hat{y}_3, <eos>]
\end{align*}$$

En cada iteración:

- **obtener las oraciones de origen y de destino del lote, X y Y**
- **poner a cero los gradientes calculados a partir del último lote**
- **introduzca el origen y el destino en el modelo para obtener el resultado, $\hat{Y}$**
- **como la función de pérdida solo funciona en entradas 2d con objetivos 1d, necesitamos aplanar cada uno de ellos con `.view`**
  - **cortamos la primera columna de los tensores de salida y destino como se mencionó anteriormente**
- **calcula los gradientes con `loss.backward()`**
- **recorte los gradientes para evitar que exploten (un problema común en RNN)**
- **actualizar los parámetros de nuestro modelo haciendo un paso optimizador**
- **sumar el valor de la pérdida a un total acumulado**

Finalmente, devolvemos la pérdida que se promedia en todos los batches.

In [48]:

```
def train(model, iterator, optimizer, criterion, clip):

    model.train()

    epoch_loss = 0

    for i, batch in enumerate(iterator):

        src = batch.src
        trg = batch.trg

        # Limpiar gradientes del paso anterior
        optimizer.zero_grad()

        output = model(src, trg)

        #trg = [trg len, batch size]
        #output = [trg len, batch size, output dim]

        output_dim = output.shape[-1]

        output = output[1:].view(-1, output_dim)
        trg = trg[1:].view(-1)

        #trg = [(trg len - 1) * batch size]
        #output = [(trg len - 1) * batch size, output dim]

        # Calcular la pérdida usando el criterio definido
        loss = criterion(output, trg)

        loss.backward()

        torch.nn.utils.clip_grad_norm_(model.parameters(), clip)

        optimizer.step()

        epoch_loss += loss.item()

    return epoch_loss / len(iterator)
```

Nuestro ciclo de evaluación es similar a nuestro ciclo de entrenamiento, sin embargo, como no estamos actualizando ningún parámetro, no necesitamos pasar un optimizador o un valor de clip.

Debemos recordar poner el modelo en modo de evaluación con `model.eval()`. Esto desactivará el dropout (y la batch normalization, si se usa).

Usamos el bloque `with torch.no_grad()` para garantizar que no se calculen gradientes dentro del bloque. Esto reduce el consumo de memoria y acelera el proceso.

El ciclo de iteración es similar (sin las actualizaciones de parámetros); sin embargo, debemos asegurarnos de

desactivar el forzado del maestro para la evaluación. } Esto hará que el modelo solo use sus propias predicciones para hacer más predicciones dentro de una oración, lo que refleja cómo se usaría en la implementación.

In [49]:

```
def evaluate(model, iterator, criterion):

    model.eval()

    epoch_loss = 0

    with torch.no_grad():

        for i, batch in enumerate(iterator):

            src = batch.src
            trg = batch.trg

            output = model(src, trg, 0) #turn off teacher forcing

            #trg = [trg len, batch size]
            #output = [trg len, batch size, output dim]

            output_dim = output.shape[-1]

            output = output[1:].view(-1, output_dim)
            trg = trg[1:].view(-1)

            #trg = [(trg len - 1) * batch size]
            #output = [(trg len - 1) * batch size, output dim]

            loss = criterion(output, trg)

            epoch_loss += loss.item()

    return epoch_loss / len(iterator)
```

A continuación, crearemos una función que usaremos para decirnos cuánto tarda una época.

In [50]:

```
def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs
```

Ahora sí, ¡empecemos a entrenar a nuestro modelo!

En cada época, comprobaremos si nuestro modelo ha logrado la mejor pérdida de validación hasta el momento. Si es así, actualizaremos nuestra mejor pérdida de validación y guardaremos los parámetros de nuestro modelo (llamado `state_dict` en PyTorch). Luego, cuando lleguemos a probar nuestro modelo, usaremos los parámetros guardados para lograr la mejor pérdida de validación.

Estaremos mostrando tanto la pérdida como la perplejidad en cada época. Es más fácil ver un cambio en la perplejidad que un cambio en la pérdida ya que los números son mucho mayores.

Ademas, cargaremos los parámetros (`state_dict`) que dieron a nuestro modelo la mejor pérdida de validación y ejecutaremos el modelo en el conjunto de prueba.

In [51]:

```
# Definir hiperparámetros de entrenamiento
N_EPOCHS = 5
CLIP = 1

best_valid_loss = float('inf')
```

```
for epoch in range(N_EPOCHS):

    start_time = time.time()

    train_loss = train(model, train_iterator, optimizer, criterion, CLIP)
    valid_loss = evaluate(model, valid_iterator, criterion)

    end_time = time.time()

    epoch_mins, epoch_secs = epoch_time(start_time, end_time)

    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'tut1-model.pt')

    print(f'Epoch: {epoch+1:02} | Time: {epoch_mins}m {epoch_secs}s')
    print(f'\tTrain Loss: {train_loss:.3f} | Train PPL: {math.exp(train_loss):7.3f}')
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. PPL: {math.exp(valid_loss):7.3f}')
```

```
Epoch: 01 | Time: 10m 31s
 Train Loss: 5.058 | Train PPL: 157.307
  Val. Loss: 4.958 |  Val. PPL: 142.370
Epoch: 02 | Time: 10m 37s
 Train Loss: 4.496 | Train PPL:  89.671
  Val. Loss: 4.795 |  Val. PPL: 120.908
Epoch: 03 | Time: 10m 40s
 Train Loss: 4.219 | Train PPL:  67.943
  Val. Loss: 4.628 |  Val. PPL: 102.312
Epoch: 04 | Time: 10m 46s
 Train Loss: 4.008 | Train PPL:  55.022
  Val. Loss: 4.516 |  Val. PPL:  91.424
Epoch: 05 | Time: 9m 15s
 Train Loss: 3.852 | Train PPL:  47.065
  Val. Loss: 4.447 |  Val. PPL:  85.351
```

In [54]:

```
# Se valuara que el loss de training sea menor a 4 y el de validacion a 4.51

with tick.marks(25):
    assert compare_numbers(new_representation(train_loss), "3c3d", '0x1.0000000000000p+2
')

with tick.marks(25):
    assert compare_numbers(new_representation(valid_loss), "3c3d", '0x1.2000000000000p+2
')
```

✓ [25 marks]

✓ [25 marks]

In [55]:

```
model.load_state_dict(torch.load('tut1-model.pt'))

test_loss = evaluate(model, test_iterator, criterion)

print(f'| Test Loss: {test_loss:.3f} | Test PPL: {math.exp(test_loss):7.3f} |')
```

```
| Test Loss: 4.408 | Test PPL:  82.144 |
```

In [56]:

```
print()
print("La fraccion de abajo muestra su rendimiento basado en las partes visibles de este
```

```
laboratorio")
tick.summarise_marks() #
```

```
La fraccion de abajo muestra su rendimiento basado en las partes visibles de este laborat
orio
```

50 / 50 marks (100.0%)