



DESAFÍO N°2 - JUAN I. MUNAR

1. BOT DNN + Spacy (PyTorch)

1.1. Instalo e importo dependencias, obtengo un diccionario

```
# Librerías varias
import json
import string
import random
import numpy as np

# Ploteo
import matplotlib.pyplot as plt
import seaborn as sns

# Torch
import torch
import torch.nn.functional as F
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader

# Obtención resultados
import torchsummary

# Torch helpers
import os
import platform
if os.access('torch_helpers.py', os.F_OK) is False:
    if platform.system() == 'Windows':
        !curl !wget https://raw.githubusercontent.com/FIUBA-Posgrado-Inteligencia-Artificial
    else:
        !wget torch_helpers.py https://raw.githubusercontent.com/FIUBA-Posgrado-Inteligencia
```

```
# La última versión de spacy-stanza (>1.0) es compatible solo con spacy >=3.0
# Nota: spacy 3.0 incorpora al pepiline nlp transformers
!pip install -U spacy==3.1 --quiet
!pip install -U spacy-stanza==1.0.0 --quiet

# SpaCy armó un wrapper para los pipelines y modelos de Stanza (librería de NLP de Stanford)
import stanza
import spacy_stanza

# Descargo el diccionario en español y armo el pipeline de NLP con spacy
stanza.download("es")
nlp = spacy_stanza.load_pipeline("es")
```

```
Downloading 140k/?
https://raw.githubusercontent.com/stanfordnlp/stanza- [00:00<00:00,
resources/main/resources_1.2.2.json: 9.09MB/s]
INFO:stanza:Downloading default packages for language: es (Spanish)...
INFO:stanza:File exists: /root/stanza_resources/es/default.zip.
INFO:stanza:Finished downloading models and saved to /root/stanza_resources.
/usr/local/lib/python3.10/dist-packages/torch/__init__.py:614: UserWarning: torch.set_de
_C._set_default_tensor_type(t)
INFO:stanza:Loading these models for language: es (Spanish):
=====
| Processor | Package |
-----
| tokenize  | ancora  |
| mwt       | ancora  |
| pos       | ancora  |
| lemma     | ancora  |
| depparse  | ancora  |
| ner       | conll02 |
=====

INFO:stanza:Use device: cpu
INFO:stanza:Loading: tokenize
INFO:stanza:Loading: mwt
INFO:stanza:Loading: pos
INFO:stanza:Loading: lemma
```

1.2. Preprocesamiento del texto

```
# Librerías
import re
import unicodedata

# Función de preprocesamiento
def preprocess_clean_text(text):
    # Pasar todo el texto a minúsculas
    text = text.lower()
    # Sacar tildes de las palabras
    text = unicodedata.normalize('NFKD', text).encode('ascii', 'ignore').decode('utf-8', 'ignore')
    # Quitar caracteres especiales
    pattern = r'^a-zA-z0-9.,!?:/;\\"'\s]'
    text = re.sub(pattern, '', text)
    pattern = r'^a-zA-z.,!?:/;\\"'\s]'
    # Quitar números
    text = re.sub(pattern, '', text)
    # Quitar caracteres de puntuación
    text = ''.join([c for c in text if c not in string.punctuation])
    return text
```

1.3. Diccionario de entrada, será el bot de un dealer de drogas paranoico

```
# Diccionario de entrada en formato json con posibles preguntas y respuestas
dataset = {"intents": [
    {"tag": "bienvenida",
      "patterns": ["Hola", "¿Cómo estás?", "¿Qué tal?", "¿Cómo va?", "¿Cómo andas?",
      "responses": ["Hola, buen día, se ha comunicado con un centro de denuncias de
    },
    {"tag": "estupefacientes",
      "patterns": ["Están vendiendo drogas", "venden cocaína",
        "venden fafafa", "venden marihuana",
        "venden extásis", "venden pastillas", "venden LSD",
        "hay dealers", "narcotráfico"],
      "responses": ["Dígame la dirección"]
    },
    {"tag": "direccion",
      "patterns": ["la dirección es calle", "la ubicación es calle", "el lugar queda",
      "responses": ["Envío un efectivos al lugar. ¿Algo más en que lo pueda ayudar?"
    },
    {"tag": "homicidio",
      "patterns": ["hubo un asesinato", "hubo un homicidio",
        "mataron a una persona", "mataron a alguien",
        "le dispararon a alguien", "le dispararon a mi"],
      "responses": ["Por favor póngase a resguardo, Dígame la dirección"]
    },
    {"tag": "robo",
      "patterns": ["hubo un robo", "hubo un hurto", "hay un ladrón",
        "hay un chorro", "me robaron", "le robaron"],
      "responses": ["Si pudo identificar al ladrón, por favor dirijase a la comisari
    },
    {"tag": "heridos",
      "patterns": [ "hay un herido", "hay un hombre lastimado",
        "hay una persona grave", "hay un quemado",
        "fue apuñalada", "estoy herido", "estoy lastimado"],
      "responses": ["Hay una ambulancia en camino. ¿Algo más en que lo pueda ayudar?
    },
    {"tag": "fuego",
      "patterns": [ "hay fuego", "hay un incendio", "se está quemando"],
      "responses": ["Muy interesante, lo anotaré en mi máquina de escribir invisible
    },
    {"tag": "despedida",
      "patterns": ["Adios", "Chau", "Hasta luego", "Bye", "ok", "si", "no"],
      "responses": ["Gracias, adios"]
    }
  ]
}
```

1.4. Preprocesamiento y armado del dataset

```

# Datos que necesitaremos, las palabras o vocabulario
words = []
classes = []
doc_X = []
doc_y = []

# Por cada intención (intents) debemos tomar los patrones que la caracterizan
# a esa intención y transformarla a tokens para almacenar en doc_X

# El tag de cada intención se almacena como doc_Y (la clase a predecir)

for intent in dataset["intents"]:
    for pattern in intent["patterns"]:
        # transformar el patron a tokens
        tokens = nlp(preprocess_clean_text(pattern))
        # lematizar los tokens
        for token in tokens:
            words.append(token.lemma_)

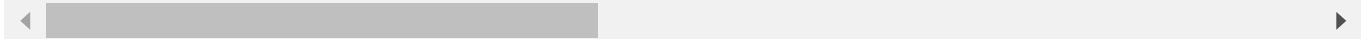
        doc_X.append(pattern)
        doc_y.append(intent["tag"])

# Agregar el tag a las clases
if intent["tag"] not in classes:
    classes.append(intent["tag"])

# Eliminar duplicados con "set" y ordenar el vocabulario y las clases por orden alfabético
words = sorted(set(words))
classes = sorted(set(classes))

<ipython-input-5-de38f27299c1>:15: UserWarning: Due to multiword token expansion or an a
    tokens = nlp(preprocess_clean_text(pattern))
<ipython-input-5-de38f27299c1>:15: UserWarning: Can't set named entities because of mult
Words: ['se', 'e', 'esta', 'quemando']
Entities: []
    tokens = nlp(preprocess_clean_text(pattern))

```

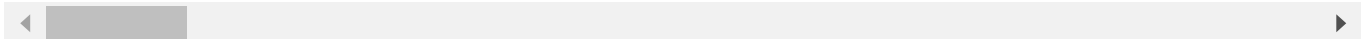


```

print("words:", words)
print("classes:", classes)
print("doc_X:", doc_X)
print("doc_y:", doc_y)

words: ['a', 'adios', 'alguien', 'andar', 'apunalar', 'asesinato', 'buen', 'bye', 'calle
classes: ['bienvenida', 'despedida', 'direccion', 'estupefacientes', 'fuego', 'heridos',
doc_X: ['Hola', '¿Cómo estás?', '¿Qué tal?', '¿Cómo va?', '¿Cómo andas?', 'Buen día', 'E
doc_y: ['bienvenida', 'bienvenida', 'bienvenida', 'bienvenida', 'bienvenida', 'bienvenic

```



[illegible]

```
class Data(Dataset):
    def __init__(self, x, y):
        # Convertir los arrays de numpy a tensores.
        # pytorch espera en general entradas 32bits
        self.x = torch.from_numpy(x.astype(np.float32))
        # las loss function esperan la salida float
        self.y = torch.from_numpy(y.astype(np.float32))

        self.len = self.y.shape[0]

    def __getitem__(self, index):
        return self.x[index], self.y[index]

    def __len__(self):
        return self.len

data_set = Data(train_X, train_y)

input_dim = data_set.x.shape[1]
print("Input dim", input_dim)

output_dim = data_set.y.shape[1]
print("Output dim", output_dim)

    Input dim 62
    Output dim 8

from torch.utils.data import DataLoader

train_loader = DataLoader(data_set, batch_size=32, shuffle=False)
```



```

class Model1(nn.Module):
    def __init__(self, input_dim, output_dim):
        super().__init__()
        self.fc1 = nn.Linear(in_features=input_dim, out_features=128) # fully connected layer
        self.fc2 = nn.Linear(in_features=128, out_features=64) # fully connected layer
        self.fc3 = nn.Linear(in_features=64, out_features=output_dim) # fully connected layer

        self.relu = nn.ReLU()
        self.softmax = nn.Softmax(dim=1) # normalize in dim 1
        self.dropout = nn.Dropout(0.5)

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

# Crear el modelo basado en la arquitectura definida
model1 = Model1(input_dim=input_dim, output_dim=output_dim)
# Crear el optimizador la una función de error
model1_optimizer = torch.optim.Adam(model1.parameters(), lr=0.001)
model1_criterion = torch.nn.CrossEntropyLoss() # Para clasificación multi categórica

torchsummary.summary(model1, input_size=(1, input_dim))

```

```

-----
Layer (type)          Output Shape          Param #
=====
      Linear-1         [-1, 1, 128]          8,064
      ReLU-2           [-1, 1, 128]           0
    Dropout-3          [-1, 1, 128]           0
      Linear-4         [-1, 1, 64]           8,256
      ReLU-5           [-1, 1, 64]           0
    Dropout-6          [-1, 1, 64]           0
      Linear-7         [-1, 1, 8]            520
      Softmax-8        [-1, 1, 8]            0
=====
Total params: 16,840
Trainable params: 16,840
Non-trainable params: 0
-----
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.06
Estimated Total Size (MB): 0.07
-----

```

```
from torch_helpers import categorical_acc

def train(model, train_loader, optimizer, criterion, epochs=100):
    # Defino listas para realizar graficas de los resultados
    train_loss = []
    train_accuracy = []

    ## Defino mi loop de entrenamiento

    for epoch in range(epochs):

        epoch_train_loss = 0.0
        epoch_train_accuracy = 0.0

        for train_data, train_target in train_loader:

            # Seteo los gradientes en cero ya que, por defecto, PyTorch
            # los va acumulando
            optimizer.zero_grad()

            output = model(train_data)

            # Computo el error de la salida comparando contra las etiquetas
            loss = criterion(output, train_target)

            # Almaceno el error del batch para luego tener el error promedio de la epoca
            epoch_train_loss += loss.item()

            # Computo el nuevo set de gradientes a lo largo de toda la red
            loss.backward()

            # Realizo el paso de optimizacion actualizando los parametros de toda la red
            optimizer.step()

            # Calculo el accuracy del batch
            accuracy = categorical_acc(output, train_target)
            # Almaceno el accuracy del batch para luego tener el accuracy promedio de la epoca
            epoch_train_accuracy += accuracy.item()

        # Calculo la media de error y accuracy para la epoca de entrenamiento.
        # La longitud de train_loader es igual a la cantidad de batches dentro de una epoca.
        epoch_train_loss = epoch_train_loss / len(train_loader)
        train_loss.append(epoch_train_loss)
        epoch_train_accuracy = epoch_train_accuracy / len(train_loader)
        train_accuracy.append(epoch_train_accuracy)

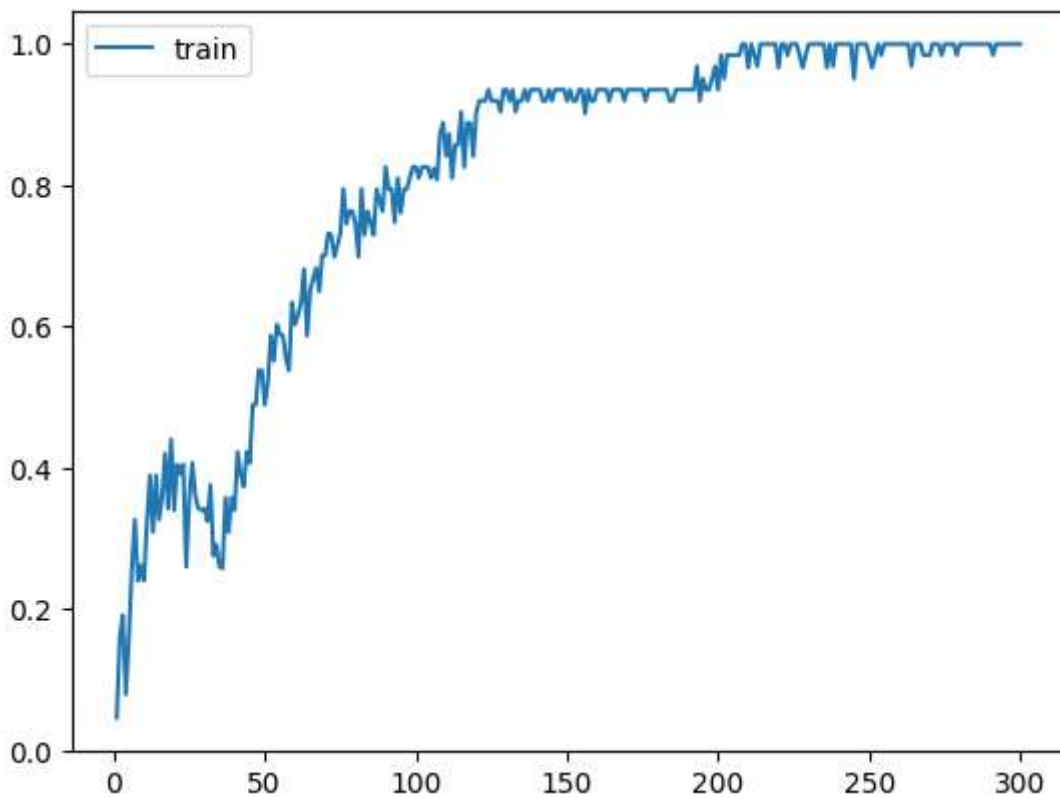
        print(f"Epoch: {epoch+1}/{epochs} - Train loss {epoch_train_loss:.3f} - Train accuracy {epoch_train_accuracy:.3f}")

    history = {
        "loss": train_loss,
        "accuracy": train_accuracy,
```

```
}  
  
return history  
  
history1 = train(model1,  
                  train_loader,  
                  model1_optimizer,  
                  model1_criterion,  
                  epochs=300  
                  )
```

```
Epoch: 286/300 - Train loss 1.281 - Train accuracy 1.000  
Epoch: 287/300 - Train loss 1.285 - Train accuracy 1.000  
Epoch: 288/300 - Train loss 1.288 - Train accuracy 1.000  
Epoch: 289/300 - Train loss 1.286 - Train accuracy 1.000  
Epoch: 290/300 - Train loss 1.289 - Train accuracy 1.000  
Epoch: 291/300 - Train loss 1.314 - Train accuracy 0.984  
Epoch: 292/300 - Train loss 1.288 - Train accuracy 1.000  
Epoch: 293/300 - Train loss 1.280 - Train accuracy 1.000  
Epoch: 294/300 - Train loss 1.284 - Train accuracy 1.000  
Epoch: 295/300 - Train loss 1.296 - Train accuracy 1.000  
Epoch: 296/300 - Train loss 1.287 - Train accuracy 1.000  
Epoch: 297/300 - Train loss 1.283 - Train accuracy 1.000  
Epoch: 298/300 - Train loss 1.284 - Train accuracy 1.000  
Epoch: 299/300 - Train loss 1.285 - Train accuracy 1.000  
Epoch: 300/300 - Train loss 1.294 - Train accuracy 1.000
```

```
epoch_count = range(1, len(history1['accuracy']) + 1)  
sns.lineplot(x=epoch_count, y=history1['accuracy'], label='train')  
plt.show()
```



1.6. Test validación

```
def text_to_tokens(text):
    lemma_tokens = []
    tokens = nlp(preprocess_clean_text(text))
    for token in tokens:
        lemma_tokens.append(token.lemma_)
    #print(lemma_tokens)
    return lemma_tokens

def bag_of_words(text, vocab):
    tokens = text_to_tokens(text)
    bow = [0] * len(vocab)
    for w in tokens:
        for idx, word in enumerate(vocab):
            if word == w:
                bow[idx] = 1
    #print(bow)
    return np.array(bow)

def pred_class(text, vocab, labels):
    bow = bag_of_words(text, vocab)
    words_recognized = sum(bow)

    return_list = []
    if words_recognized > 0:
        x = torch.from_numpy(np.array([bow]).astype(np.float32))
```