TP2: Redes Recurrentes y Representaciones Incrustadas

1. (100 puntos) Red LSTM

Imports

In [31]:
```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
import pandas as pd
from sklearn.model_selection import train_test_split
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from sklearn.metrics import accuracy_score
import numpy as np
```

De ser necesario ejecutar lo siguiente para descargar paquetes de NLTK

In [ ]:
```
nltk.download('stopwords')
nltk.download('punkt')
nltk.download('wordnet')
nltk.download('omw-1.4')
```

Constantes

In [32]:
```
MAX_WORDS = 200 # Establece el TOP de palabras que se utilizan en el diccionario de palabras, el tamaño de la representación incrustada.
BATCH_SIZE = 64 # Tamaño del batch que se utilizará para el dataset de entrenamiento.
COLLECTION_PATH = ".\\smsspamcollection" # Ruta base para encontrar el dataset de pruebas de spam.
EPOCHS = 10
```

Funciones requeridas

Código Provisto

Se corresponde con el código encontrado en el archivo "Natural_disaster_NLP_LSTM.ipynb", del cual se realizaron las modificaciones y consideraciones que fueron mencionadas durante las clases del curso.

Dataset mapper

In [33]:
```
class DatasetMaper(Dataset):
    '''
    Handles batches of dataset
    '''

    def __init__(self, x, y):
        """
        Inits the dataset mapper
        """
        self.x = x
        self.y = y

    def __len__(self):
        """
        Returns the length of the dataset
        """
        return len(self.x)

    def __getitem__(self, idx):
        """
        Fetches a specific item by id
        """
        return self.x[idx], self.y[idx]
```

Modelo de entrenamiento

In [34]:
```
class LSTM_TweetClassifier(nn.ModuleList):

    def __init__(self, batch_size=BATCH_SIZE, hidden_dim=20, lstm_layers=2, max_words=MAX_WORDS):
        """
        param batch_size: batch size for training data
        param hidden_dim: number of hidden units used in the LSTM and the Embedding layer
        param lstm_layers: number of lstm_layers
        param max_words: maximum sentence length
        """
        super(LSTM_TweetClassifier, self).__init__()
        # batch size during training
        self.batch_size = batch_size
        # number of hidden units in the LSTM layer
        self.hidden_dim = hidden_dim
        # Number of LSTM layers
        self.LSTM_layers = lstm_layers
        self.input_size = max_words  # embedding dimension

        self.dropout = nn.Dropout(0.5)  # Para descartar
        #  N, D                 #  hidden_dim -> To set embedding size
        self.embedding = nn.Embedding(self.input_size, self.hidden_dim, padding_idx=0)  # learn representation
        self.lstm = nn.LSTM(input_size=self.hidden_dim, hidden_size=self.hidden_dim, num_layers=self.LSTM_layers,
                            batch_first=True)
        self.fc1 = nn.Linear(in_features=self.hidden_dim, out_features=257)
        self.fc2 = nn.Linear(257, 1)
```

```
    def forward(self, x):
        """
        Forward pass
        param x: model input
        """
        # it starts with noisy estimations of h and c
        h = torch.zeros((self.LSTM_layers, x.size(0), self.hidden_dim))  # "Context"
        c = torch.zeros((self.LSTM_layers, x.size(0), self.hidden_dim))  # "State"
        # Fills the input Tensor with values according to the method described in Understanding the difficulty of training deep feedforward neural networks - Glorot, X. & Bengio, Y. (2010), usin
        # The resulting tensor will have values sampled from \mathcal{N}(0, \text{std}^2)N(0,std)
        torch.nn.init.xavier_normal_(h)
        torch.nn.init.xavier_normal_(c)
        out = self.embedding(x)
        out, (hidden, cell) = self.lstm(out, (h, c))
        out = self.dropout(out)
        #  Fully connected network para la clasificacion
        out = torch.relu_(self.fc1(out[:, -1, :]))
        out = self.dropout(out)
        # sigmoid activation function
        out = torch.sigmoid(self.fc2(out))
        return out
```

Función de entrenamiento y evaluación

In [46]:
```
def calculate_accuray(y_pred, y_gt):
    return accuracy_score(y_pred, y_gt)


def evaluate_model(model, loader_test):
    predictions = []
    accuracies = []
    model.eval()
    with torch.no_grad():
        for x_batch, y_batch in loader_test:
            x_batch = torch.t(torch.stack(x_batch))
            x = x_batch.type(torch.LongTensor)
            y = y_batch.type(torch.FloatTensor)
            y_pred = model(x)
            y_pred = torch.round(y_pred).flatten()
            predictions += list(y_pred.detach().numpy())
            acc_batch = accuracy_score(y_pred, y)
            accuracies += [acc_batch]
    return np.array(accuracies)


def train_model(model, epochs=EPOCHS, learning_rate=0.01):
    optimizer = optim.RMSprop(model.parameters(), lr=learning_rate)
    for epoch in range(epochs):
        model.train()
        loss_dataset = 0
        for x_batch, y_batch in loader_training:
            x_batch = torch.t(torch.stack(x_batch))
            x = x_batch.type(torch.LongTensor)
            y = y_batch.type(torch.FloatTensor)
            y_pred = model(x).flatten()
            loss = F.binary_cross_entropy(y_pred, y)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()
            loss_dataset += loss
        accuracies = evaluate_model(model, loader_test)
        print("Epoch #", epoch, " Loss training: ", loss_dataset.item(), " Accuracy test (mean): ", accuracies.mean(), " Standard deviation: ", accuracies.std())
```

Código base implementado

Preprocesamientos

Métodos de preprocesado implementados según indicaciones del punto 1.a del TP2. Esto incluye la implementación de los métodos *preprocesar_documento_1*, *preprocesar_documento_2* y *preprocess_example*. Este último tiene por objetivo presentar la ejecución de los dos primeros métodos citados en la sección de *Resolución de ejercicios* de más adelante.

In [35]:
```
def preprocesar_documento_1(document, to_string=False):
    stop_words = set(stopwords.words('english'))
    word_tokens = word_tokenize(document)
    word_tokens = [token.lower() for token in word_tokens]
    word_tokens = [token for token in word_tokens if token.isalnum()]  # To remove punctuations
    filtration = [word for word in word_tokens if word not in stop_words]
    return filtration if not to_string else ' '.join(filtration)

def preprocesar_documento_2(document, to_string=False):
    word_tokens = preprocesar_documento_1(document)
    lemmatizer = WordNetLemmatizer()
    filtration = [lemmatizer.lemmatize(word, pos="v") for word in word_tokens]
    return filtration if not to_string else ' '.join(filtration)

def preprocess_example():
    with open(COLLECTION_PATH + "\\SMSSpamCollection", 'r') as collection:
        for line in collection:
            d0 = line.replace("ham", "").replace("spam", "").replace("\n", "").replace("\t", "")
            d1 = "I thought, I thought of thinking of thanking you for the gift"
            d2 = "She was thinking of going to go and get you a GIFT!"
            print("Testing preprocessing methods with different lines:\n\n{}\n{}\n{}".format(print_test(1, d0), print_test(2, d1), print_test(3, d2)))
            break

def print_test(test, line):
    return "* Test line #{}: {}\nPreprocess #1: {}\nPreprocess #2: {}\n".format(test, line, preprocesar_documento_1(line, to_string=True), preprocesar_documento_2(line, to_string=True))
```

Tokenizer

Los métodos *tokens_to_indexes*, *sequence_to_number_combination* y *adapt_to_input_layer* son implementaciones generadas como "equivalentes" para las funciones implementadas en el código provisto llamadas *prepare_tokens* y *sequence_to_token*. Su objetivo principal es, en tres pasos bien definidos por método:

1. Establecer el *diccionario* del TOP *MAX_WORDS* (cuya longitud se encuentra en la sección de *Constantes*) para la posterior definición de representación numérica para las líneas preprocesadas.

2. Generar la representación equivalente de cada línea según los valores numéricos obtenidos del diccionario creado en el punto anterior (se ignoran aquellas palabras que no estén en el diccionario).
3. Adaptar la representación obtenida del punto anterior al tamaño *MAX_WORDS* para que coincida con la entrada para el entrenamiento de este punto.

In [36]:
```
def tokens_to_indexes(sentences: list) -> dict:
    words_count = {}
    for sentence in sentences:
        for word in sentence:
            if word in words_count:
                words_count[word] += 1
            else:
                words_count[word] = 1
    words_to_list = list(dict(sorted(words_count.items(), key=lambda item: item[1])))
    words_to_list.reverse()
    top_max_words = words_to_list[0: MAX_WORDS - 1]
    index_words = {}
    counter = 1
    for word in top_max_words:
        index_words[word] = counter
        counter += 1
    return index_words

def sequence_to_number_combination(word: list, index_words: dict):
    sequence = []
    for token in word:
        if token in index_words:
            sequence.append(index_words[token])
    return sequence

def adapt_to_input_layer(dataset, input_layer_size=MAX_WORDS):
    new_dataset = []
    for data in dataset:
        zeros_to_add = input_layer_size - len(data)
        new_data_list = [0 for zero in range(zeros_to_add)]
        new_data_list.extend(data)
        new_dataset.append(new_data_list)
    return new_dataset
```

Data Loader

Este método concentra todo lo discutido anteriormente y termina por generar los datasets de entrenamiento y pruebas por utilizar en la siguiente sección de *Resolución de ejercicios*, más específicamente en los puntos 1.b y 1.c.

In [47]:
```
def load_process_data(f_prepros, test_size=0.4, train_size=0.6):
    # Load dataset
    dataset_frame = pd.read_csv(COLLECTION_PATH + '\\SMSSpamCollection', delimiter='\t', header=None)
    # Preprocess document
    sentences_list = [f_prepros(sentence) for sentence in dataset_frame[1]]
    # Add index to every word
    words_dictionary = tokens_to_indexes(sentences_list)
    # Transform tokens (words) to indexes (numbers)
    sentences_list = [sequence_to_number_combination(sentence, words_dictionary) for sentence in sentences_list]
    # One-hot tags ham = 0, spam = 1
    tags_list = [0 if tag == "ham" else 1 for tag in dataset_frame[0]]
    # Build train and test datasets
    X_train_raw, X_test_raw, y_train, y_test = train_test_split(sentences_list, tags_list, test_size=test_size, train_size=train_size)
    # Adapt data to input layer
    x_train = adapt_to_input_layer(X_train_raw)
    x_test = adapt_to_input_layer(X_test_raw)
    training_set = DatasetMaper(x_train, y_train)
    test_set = DatasetMaper(x_test, y_test)
    loader_training = DataLoader(training_set, batch_size=BATCH_SIZE)
    loader_test = DataLoader(test_set)
    return loader_training, loader_test
```

Resolución de ejercicios

En esta sección, haciendo uso del código provisto e implementado anteriormente, se dará resolución a los ejercicios propuestos en el punto #1 del TP2.

1. Implemente la siguiente arquitectura de una red LSTM:

a. 1) Muestre un ejemplo con una entrada escogida del pre-procesamiento con ambos enfoques, y explique brevemente los posibles efectos de utilizar el segundo enfoque al primero.

In [39]:
```
preprocess_example()
```

```
Testing preprocessing methods with different lines:

* Test line #1: Go until jurong point, crazy.. Available only in bugis n great world la e buffet... Cine there got amore wat...
Preprocess #1: go jurong point crazy available bugis n great world la e buffet cine got amore wat
Preprocess #2: go jurong point crazy available bugis n great world la e buffet cine get amore wat

* Test line #2: I thought, I thought of thinking of thanking you for the gift
Preprocess #1: thought thought thinking thanking gift
Preprocess #2: think think think thank gift

* Test line #3: She was thinking of going to go and get you a GIFT!
Preprocess #1: thinking going go get gift
Preprocess #2: think go go get gift
```

R/ En el primer preprocesamiento se emplea únicamente la eliminación de mayúsculas, signos de puntuación y *stopwords*. Estas últimas son palabras cuya significancia es mínima en el procesamiento de lenguaje natural por lo que es posible retirarlas sin generar mayor impacto en el entendimiento de las expresiones. Como se puede observar en las anteriores entradas, su funcionamiento consiste en estandarizar la expresión al retirar mayúsculas y puntuación, además de eliminar preposiciones y palabras que sean entendidas como *stopwords* por *nltk*. Puede observarse en el *test line #1* como la palabra *until* desaparece al ser preprocesada. Entre los ejemplos que se muestran, el *test line #1* elimina palabras como *until*, *only* e *in*, mientras que en el *test line #2* también se retiran *of*, *you*, *for* y *the*. Esto evidentemente reduce el espectro de palabras a tomar en cuenta para el procesamiento.

En el segundo preprocesamiento, aparte de aplicar primero lo expuesto anteriormente, se emplea también la *lematización* de la expresión. Esto permite obtener la *raíz* de una palabra a partir de cualquiera de sus variables posibles, tal como se puede observar en los ejemplos anteriores donde *thinking* pasa a su forma base *think*. Además de eso, otras funciones con las que cuenta es reducir las palabras a su forma singular, cambiar el tiempo, entre otros. Por ejemplo, en el *test line #1* se cambia el verbo en pasado *got* por el verbo *get*, en el *test line #2* se cambian las palabras *thought*, *thinking* y *thanking* por *think*, *think* y *thank* y, finalmente, en el *test line #3* se cambian las palabras *thinking* y *going* por *think* y *go*.

Los efectos individuales que se identifican para el primer preprocesamiento consiste en estandarizar todas las frases del dataset por utilizar. Esto igualmente mantiene palabras que son iguales pero que pueden estar conjugadas, haciendo que al ser procesadas se interpreten como palabras distintas, afectando el entrenamiento y posterior prueba utilizando frases muy similares pero que estén conjugadas en otros tiempos, solo por dar un ejemplo.

Esto precisamente es lo que se busca con la especialización del método #1, es decir, el segundo preprocesamiento. Este no solo aprovecha las virtudes de estandarización del primer preprocesamiento, sino que también incluye la lematización, lo cual reduce aún más la gama de palabras a utilizar para el análisis, produciendo que tanto al entrenar como probar se encuentre que las palabras *get* y *got*, aunque conjugadas en distinto tiempo, corresponden a la misma palabra y tienen un significado y aplicación relativamente similares. El efecto que se identifica es que optimizará la ejecución del entrenamiento del modelo y favorecerá una mayor efectividad al momento de predecir ya que no tomará las conjugaciones y otras variantes de las palabras como palabras completamente distintas, haciendo que ahora se les pueda dar más sentido al interpretar que son la misma palabra.

b) 10 corridas para D=20, D=100 utilizando el segundo enfoque de preprocesamiento, resultados en tablas con medias y desviaciones estandar:

D=20

In [42]:
```
# Preprocessing and dataset selection based on the train_test_split method using 40% for test size and 60% for train size.
# train_test_split returns random partitions for training and testing.
loader_training, loader_test = load_process_data(preprocesar_documento_2, test_size=0.4, train_size=0.6)
model = LSTM_TweetClassifier(hidden_dim=20)
train_model(model, EPOCHS)
accuracies = evaluate_model(model, loader_test)
print("D = 20")
print("Final average accuracy (mean): ", accuracies.mean(), "Final standard deviation: ", accuracies.std())
```

```
Epoch # 0  Loss training:  16.875024795532227  Accuracy test (mean):  0.9537909376401974  Standard deviation:  0.20993757385382558
Epoch # 1  Loss training:  6.4264678955078125  Accuracy test (mean):  0.9591745177209511  Standard deviation:  0.19788573034893636
Epoch # 2  Loss training:  4.92019510269165  Accuracy test (mean):  0.9591745177209511  Standard deviation:  0.19788573034893636
Epoch # 3  Loss training:  3.6861252784729004  Accuracy test (mean):  0.9569313593539704  Standard deviation:  0.20301165690406248
Epoch # 4  Loss training:  3.0383758544921875  Accuracy test (mean):  0.9614176760879318  Standard deviation:  0.19259732135627547
Epoch # 5  Loss training:  2.3465917110443115  Accuracy test (mean):  0.9573799910273665  Standard deviation:  0.20199887080824516
Epoch # 6  Loss training:  2.394618272781372  Accuracy test (mean):  0.9641094661283086  Standard deviation:  0.1860172127790768
Epoch # 7  Loss training:  1.7769286632537842  Accuracy test (mean):  0.9659039928218932  Standard deviation:  0.18147580960727866
Epoch # 8  Loss training:  1.9297988414764404  Accuracy test (mean):  0.9627635711081203  Standard deviation:  0.18934063815055593
Epoch # 9  Loss training:  1.6380583047866821  Accuracy test (mean):  0.9632122027815164  Standard deviation:  0.1882404185989166
D = 20
Final average accuracy (mean) :  0.9632122027815164 Final standard deviation:  0.1882404185989166
```

D=100

In [43]:
```
# Preprocessing and dataset selection based on the train_test_split method using 40% for test size and 80% for train size.
# train_test_split returns random partitions for training and testing.
loader_training, loader_test = load_process_data(preprocesar_documento_2, test_size=0.4, train_size=0.6)
model = LSTM_TweetClassifier(hidden_dim=100)
train_model(model, EPOCHS)
accuracies = evaluate_model(model, loader_test)
print("D = 100")
print("Final average accuracy (mean): ", accuracies.mean(), "Final standard deviation: ", accuracies.std())
```

```
Epoch # 0  Loss training:  68.91210174560547  Accuracy test (mean):  0.9385374607447285  Standard deviation:  0.24017680055235485
Epoch # 1  Loss training:  7.869880676269531  Accuracy test (mean):  0.9519964109466128  Standard deviation:  0.21377381619688762
Epoch # 2  Loss training:  4.8323163986206055  Accuracy test (mean):  0.9676985195154778  Standard deviation:  0.1767995894877311
Epoch # 3  Loss training:  3.4339821338653564  Accuracy test (mean):  0.9672498878420817  Standard deviation:  0.1779818595013608
Epoch # 4  Loss training:  2.911142349243164  Accuracy test (mean):  0.9726334679228353  Standard deviation:  0.16314902696379824
Epoch # 5  Loss training:  2.7866263389587402  Accuracy test (mean):  0.9668012561686855  Standard deviation:  0.1791552043322695
Epoch # 6  Loss training:  2.3446109294891357  Accuracy test (mean):  0.9690444145356663  Standard deviation:  0.17319739372431128
Epoch # 7  Loss training:  2.3271267414093018  Accuracy test (mean):  0.9690444145356663  Standard deviation:  0.17319739372431128
Epoch # 8  Loss training:  2.4226813316345215  Accuracy test (mean):  0.9685957828622701  Standard deviation:  0.17440754652163487
Epoch # 9  Loss training:  2.00620698928833  Accuracy test (mean):  0.9699416778824586  Standard deviation:  0.17074782396569396
D = 100
Final average accuracy (mean):  0.9699416778824586 Final standard deviation:  0.17074782396569396
```

Tabla de resumen:

~		D = 20			D = 100	
Epoc		Accuracy			Accuracy	
	Loss	Mean	Std	Loss	Mean	Std
1	16,8750248	0,953790938	0,209937574	68,91210175	0,938537461	0,240176801
2	6,426467896	0,959174518	0,19788573	7,869880676	0,951996411	0,213773816
3	4,920195103	0,959174518	0,19788573	4,832316399	0,96769852	0,176799589
4	3,686125278	0,956931359	0,203011657	3,433982134	0,967249888	0,17798186
5	3,038375854	0,961417676	0,192597321	2,911142349	0,972633468	0,163149027
6	2,346591711	0,957379991	0,201998871	2,786626339	0,966801256	0,179155204
7	2,394618273	0,964109466	0,186017213	2,344610929	0,969044415	0,173197394
8	1,776928663	0,965903993	0,18147581	2,327126741	0,969044415	0,173197394
9	1,929798841	0,962763571	0,189340638	2,422681332	0,968595783	0,174407547
10	1,638058305	0,963212203	0,188240419	2,006206989	0,969941678	0,170747824
Mean		0,963212203			0,969941678	
Std		0,188240419			0,170747824	

Como se puede observar, cuando $D = 20$, el loss y precisión inicial son menores comparado con $D = 100$; al aumentar la dimensionalidad de la capa de embedding, se aumenta el vector de características, esto a largo plazo puede facilitar la detección de patrones más extensos pero afecta el proceso de entrenamiento en velocidad y cantidad de épocas necesarias para llegar a ese punto. Al final de las ejecuciones, sin embargo, en ambos lados se puede ver una precisión muy similar en media 0.9632 vs 0.9699 y desviación estándar 0.1882 vs 0.1707, aunque para $D = 100$, el loss no disminuyó tanto, se mantuvo superior al inicio y al final.

c) 10 corridas para D=20, D=100 utilizando el primer enfoque de preprocesamiento, resultados en tablas con medias y desviaciones estandar:

D=20

In [44]:
```
# Preprocessing and dataset selection based on the train_test_split method using 40% for test size and 60% for train size.
# train_test_split returns random partitions for training and testing.
loader_training, loader_test = load_process_data(preprocesar_documento_1, test_size=0.4, train_size=0.6)
model = LSTM_TweetClassifier(hidden_dim=20)
train_model(model, EPOCHS)
accuracies = evaluate_model(model, loader_test)
print("D = 20")
print("Final average accuracy (mean): ", accuracies.mean(), "Final standard deviation: ", accuracies.std())
```

```
Epoch # 0  Loss training:  15.560389518737793  Accuracy test (mean):  0.9560340960071781  Standard deviation:  0.20501927538384254
Epoch # 1  Loss training:  7.25687837600708  Accuracy test (mean):  0.9632122027815164  Standard deviation:  0.18824041859891663
Epoch # 2  Loss training:  5.014931678771973  Accuracy test (mean):  0.9623149394347241  Standard deviation:  0.19043344447724353
Epoch # 3  Loss training:  4.117300033569336  Accuracy test (mean):  0.9663526244952894  Standard deviation:  0.18031979820961322
Epoch # 4  Loss training:  3.5236101150512695  Accuracy test (mean):  0.9654553611484971  Standard deviation:  0.18262340150737022
Epoch # 5  Loss training:  2.5243136882781982  Accuracy test (mean):  0.9663526244952894  Standard deviation:  0.18031979820961322
Epoch # 6  Loss training:  2.147249460220337  Accuracy test (mean):  0.9650067294751009  Standard deviation:  0.18376273164836845
Epoch # 7  Loss training:  2.262073040008545  Accuracy test (mean):  0.9645580978017048  Standard deviation:  0.18489395275903883
Epoch # 8  Loss training:  2.221160650253296  Accuracy test (mean):  0.9663526244952894  Standard deviation:  0.18031979820961322
Epoch # 9  Loss training:  1.4608172178268433  Accuracy test (mean):  0.9659039928218932  Standard deviation:  0.1814758096072787
D = 20
Final average accuracy (mean):  0.9659039928218932 Final standard deviation:  0.1814758096072787
```

D=100

In [45]:
```
# Preprocessing and dataset selection based on the train_test_split method using 40% for test size and 80% for train size.
# train_test_split returns random partitions for training and testing.
loader_training, loader_test = load_process_data(preprocesar_documento_1, test_size=0.4, train_size=0.6)
model = LSTM_TweetClassifier(hidden_dim=100)
train_model(model, EPOCHS)
accuracies = evaluate_model(model, loader_test)
print("D = 100")
print("Final average accuracy (mean): ", accuracies.mean(), "Final standard deviation: ", accuracies.std())
```

```
Epoch # 0  Loss training:  33.43926239013672  Accuracy test (mean):  0.946164199192463  Standard deviation:  0.2256933923688246
Epoch # 1  Loss training:  8.200517654418945  Accuracy test (mean):  0.9605204127411395  Standard deviation:  0.19473302095107162
Epoch # 2  Loss training:  5.922861576080322  Accuracy test (mean):  0.9641094661283086  Standard deviation:  0.1860172127790768
Epoch # 3  Loss training:  4.44744873046875  Accuracy test (mean):  0.9632122027815164  Standard deviation:  0.18824041859891663
Epoch # 4  Loss training:  3.727796792984009  Accuracy test (mean):  0.9650067294751009  Standard deviation:  0.18376273164836845
Epoch # 5  Loss training:  3.3696773052215576  Accuracy test (mean):  0.9560340960071781  Standard deviation:  0.20501927538384251
Epoch # 6  Loss training:  3.1202781200408936  Accuracy test (mean):  0.9623149394347241  Standard deviation:  0.19043344447724353
Epoch # 7  Loss training:  2.2847073078155518  Accuracy test (mean):  0.9605204127411395  Standard deviation:  0.19473302095107162
Epoch # 8  Loss training:  2.618330478668213  Accuracy test (mean):  0.9596231493943472  Standard deviation:  0.1968414604213796
Epoch # 9  Loss training:  3.423551082611084  Accuracy test (mean):  0.9632122027815164  Standard deviation:  0.1882404185989166
D = 100
Final average accuracy (mean):  0.9632122027815164 Final standard deviation:  0.1882404185989166
```

Tabla de resumen:

~		D = 20			D = 100	
Epoc		Accuracy			Accuracy	
	Loss	Mean	Std	Loss	Mean	Std
1	15,56038952	0,956034096	0,205019275	33,43926239	0,946164199	0,225693392
2	7,256878376	0,963212203	0,188240419	8	0,960520413	0,194733021
3	5,014931679	0,962314939	0,190433444	5,922861576	0,964109466	0,186017213
4	4,117300034	0,966352624	0,180319798	4,44744873	0,963212203	0,188240419
5	3,523610115	0,965455361	0,182623402	3,727796793	0,965006729	0,183762732
6	2,524313688	0,966352624	0,180319798	3,369677305	0,956034096	0,205019275
7	2,14724946	0,650067295	0,183762732	3,12027812	0,962314939	0,190433444
8	2,26207304	0,964558098	0,184893953	2,284707308	0,960520413	0,194733021
9	2,22116065	0,966352624	0,180319798	2,618330479	0,959623149	0,19684146
10	1,460817218	0,965903993	0,18147581	3,423551083	0,963212203	0,188240419
Mean		0,965903993			0,963212203	
Std		0,188240419			0,188240419	

Ahora bien, para las 10 corridas con $D = 20$ y $D = 100$ se observa un comportamiento similar al presentado en el punto *B*, pero menos acentuado, por lo que vamos a ver a continuación. Básicamente, y tomando como principal factor de observación la pérdida de cada uno, se evidencia que tener un valor para D mayor, puede ser más difícil de manejar que un valor bajo. Comparando lo obtenido, el valor 15.5604 obtenido para $D = 20$ contrasta con el valor 33.4393 obtenido para $D = 100$. Considerando el hecho de que estos valores altos para D son mas valiosos a largo plazo (es decir, con más *epochs*), se deja ver que en ejecuciones más cortas como las realizadas en este TP, la ejecución con más capas es la que se verá más afectada o desfavorecida. Sin embargo, algo que no se puede observar en esta ejecución, es el hecho de que a largo plazo la ejecución con $D = 100$, aunque vaya a ser más complicada y lenta a nivel de tiempo y recursos, terminará por obtener mejores valores para la precisión de predicción.

Esto último que se comenta si es posible observarlo medianamente a nivel de *accuracy*, ya que, aún teniendo un balance para la pérdida de 1.4608 contra 3.4236 (que marca mejores valores a favor del D más bajo), se encuentra que para la media y desviación estándar del $D = 100$ se obtienen valores de precisión muy similares e incluso mayores en ocasiones a los provistos por $D = 20$. Esto podría dar indicios de que a mayor cantidad de *epochs* el modelo entrenado para con $D = 100$ va a tener mayor efectividad y precisión en sus predicciones, en comparación con un modelo entrenado con $D = 20$ que, aunque sea más eficiente al principio (hablando de los valores en *epochs* tempranos de la tabla anterior), se va a ver desplazado por el modelo con más capas.