Bootcamp Inteligencia Artificial
Nivel Innovador

Sesión 12:

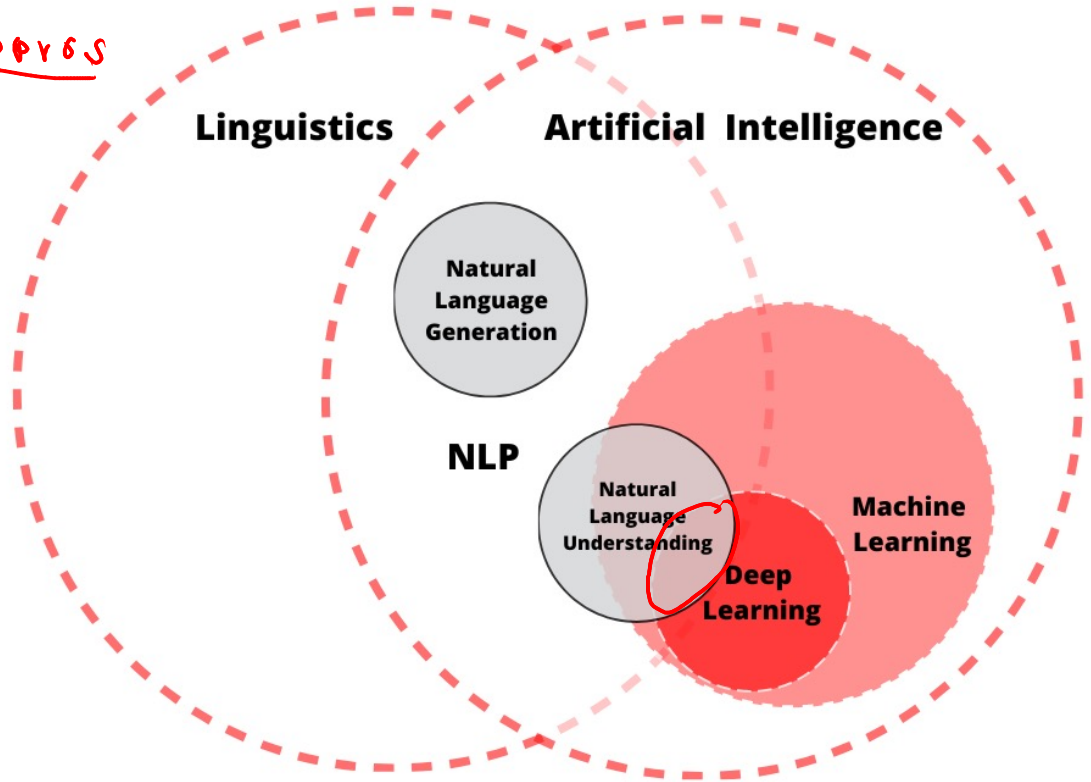
Aplicaciones de Deep Learning

Agenda

1. Procesamiento de Lenguaje Natural

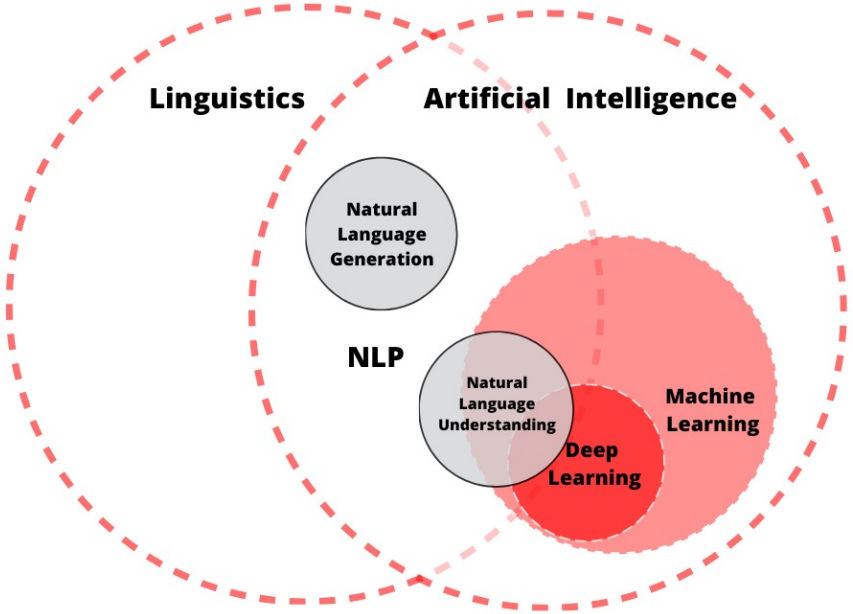
1.1 Introducción

El Procesamiento del Lenguaje Natural (NLP) es una rama del aprendizaje profundo que se ocupa de la interacción entre las computadoras y el lenguaje humano. En esta clase, exploraremos cómo aplicar técnicas de deep learning para abordar tareas específicas en NLP. Desde la tokenización hasta la generación de texto, veremos cómo las redes neuronales pueden capturar patrones complejos en datos de lenguaje natural.



1.2 Tokenización y Representación de Palabras

El primer paso crucial en el procesamiento del lenguaje natural (NLP) es la tokenización, que implica dividir un texto en unidades más pequeñas llamadas tokens. En el ámbito del lenguaje humano, los tokens suelen ser palabras o subpalabras que forman la base del análisis de texto. La biblioteca nltk (Natural Language Toolkit) en Python proporciona herramientas efectivas para llevar a cabo este proceso.



1.3 Tokenización

Vamos a realizar un ejemplo simple de tokenización utilizando nltk:

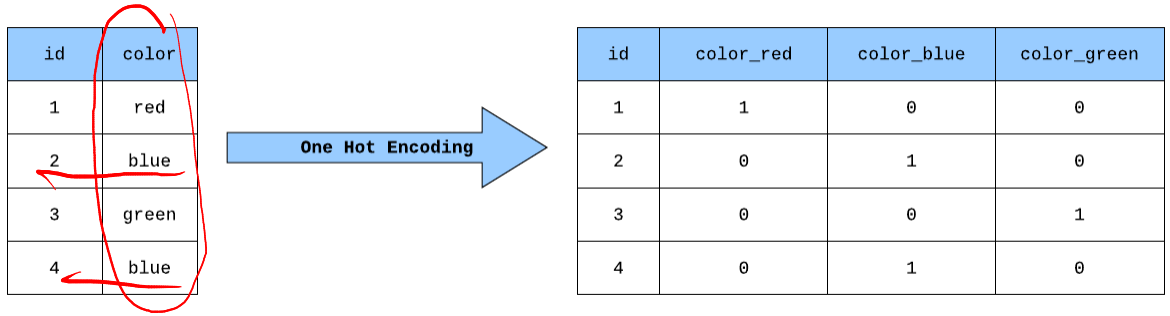
```
import nltk
from nltk.tokenize import word_tokenize

# Texto de ejemplo
text = "El procesamiento del lenguaje natural es fascinante."

# Tokenización
tokens = word_tokenize(text)
print(tokens)
```

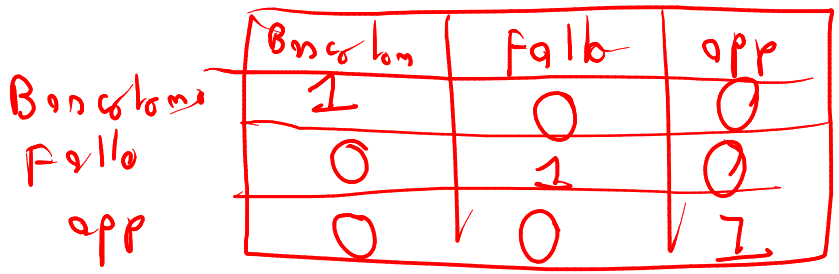
En este caso, el texto se divide en una lista de palabras. La tokenización es fundamental porque proporciona la unidad básica de procesamiento para tareas posteriores en NLP.

1.4 Representación de Palabras

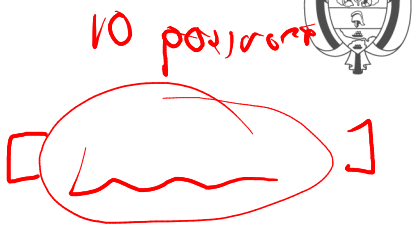
La representación de palabras es esencial para que las computadoras comprendan el significado de las palabras en un contexto dado. Métodos tradicionales como **el one-hot encoding** asignan un vector binario único a cada palabra, pero este enfoque tiene limitaciones significativas, ya que no captura relaciones semánticas.

Una técnica más avanzada es el uso de **embeddings**, que asignan palabras a vectores de números reales en un espacio semántico continuo. Un modelo popular para generar embeddings es **Word2Vec.**





1.5 Representación de Palabras



Word Embeddings con Word2Vec

Word2Vec es una técnica popular que asigna vectores de alta dimensionalidad a palabras de manera que palabras similares tengan vectores cercanos. A través de modelos como Skip-Gram y Continuous Bag of Words (CBOW), Word2Vec captura la semántica de las palabras en función de su contexto.

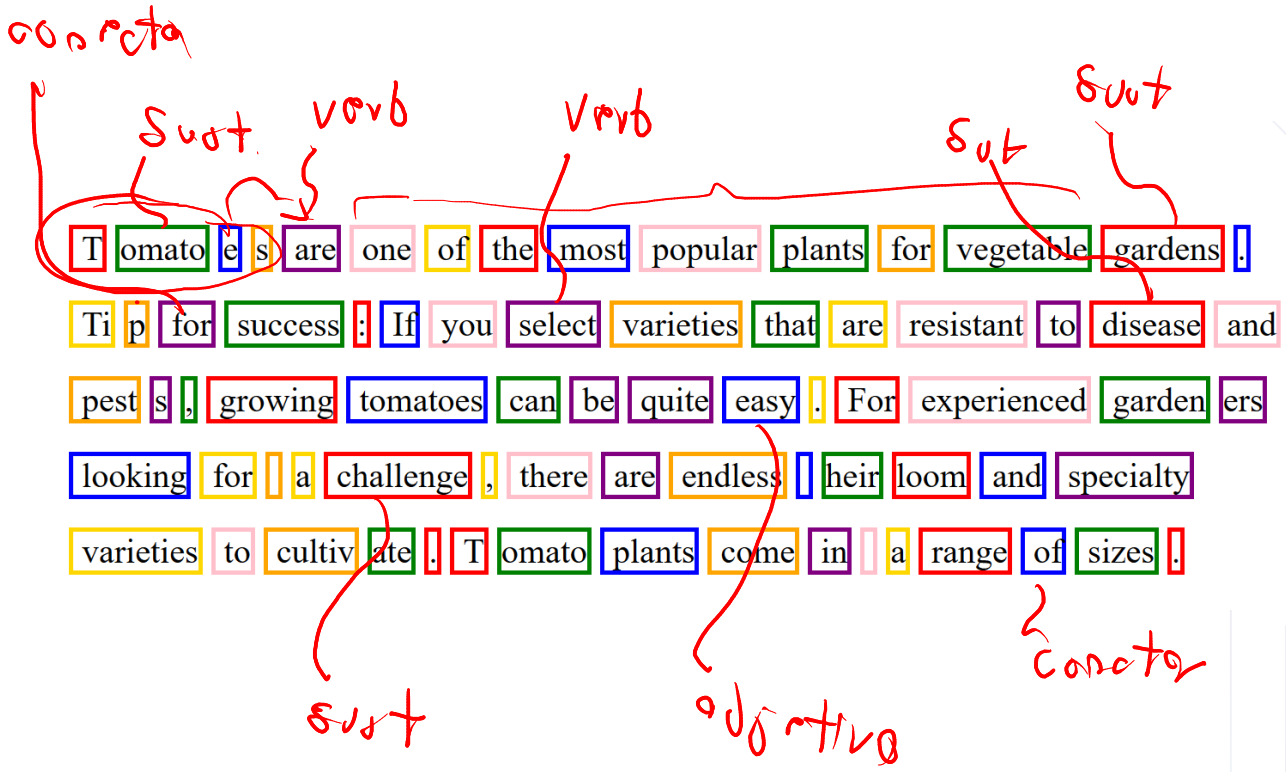
En este código, estamos entrenando un modelo Word2Vec con la lista de tokens que obtuvimos previamente. Luego, podemos acceder al embedding de una palabra específica, en este caso, 'procesamiento', para ver la representación vectorial aprendida.

```python
from gensim.models import Word2Vec

# Entrenar un modelo Word2Vec
model = Word2Vec([tokens], vector_size=10, window=3, min_count=1, workers=4)
word_vectors = model.wv
print("Embedding de 'procesamiento':", word_vectors['procesamiento'])
```

1.6 Tokenización y Representación de Palabras

La tokenización y representación de palabras son pasos fundamentales para que las máquinas comprendan el lenguaje natural. Las embeddings como las generadas por Word2Vec permiten que las palabras se representen de manera significativa, capturando las relaciones semánticas y contextuales.

Estas representaciones vectoriales son la base para muchos modelos de procesamiento del lenguaje natural basados en aprendizaje profundo.

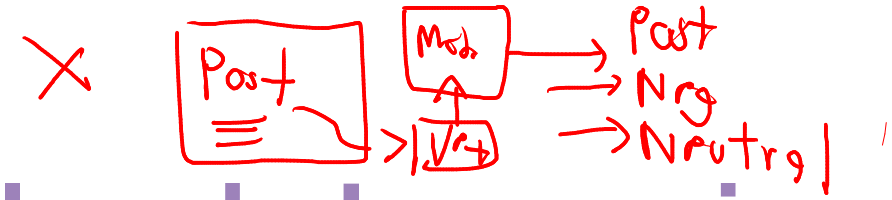


1.7 Modelos de Lenguaje y Embeddings

El componente esencial del Procesamiento del Lenguaje Natural (NLP) es la capacidad de las computadoras para entender y generar lenguaje humano. La construcción de modelos de lenguaje efectivos es crucial para esta tarea, y en el contexto del aprendizaje profundo, los embeddings desempeñan un papel fundamental.

1.8 Modelos de Lenguaje

Los modelos de lenguaje son sistemas estadísticos que asignan probabilidades a secuencias de palabras. En términos simples, estos modelos aprenden patrones y estructuras del lenguaje a partir de datos textuales y se utilizan para predecir la probabilidad de la siguiente palabra en una secuencia dada. Los modelos basados en redes neuronales, como LSTM y GRU, han demostrado ser efectivos para modelar dependencias a largo plazo en el texto.



1.9 Modelos de Lenguaje – LSTM (Long Short-Term Memory)

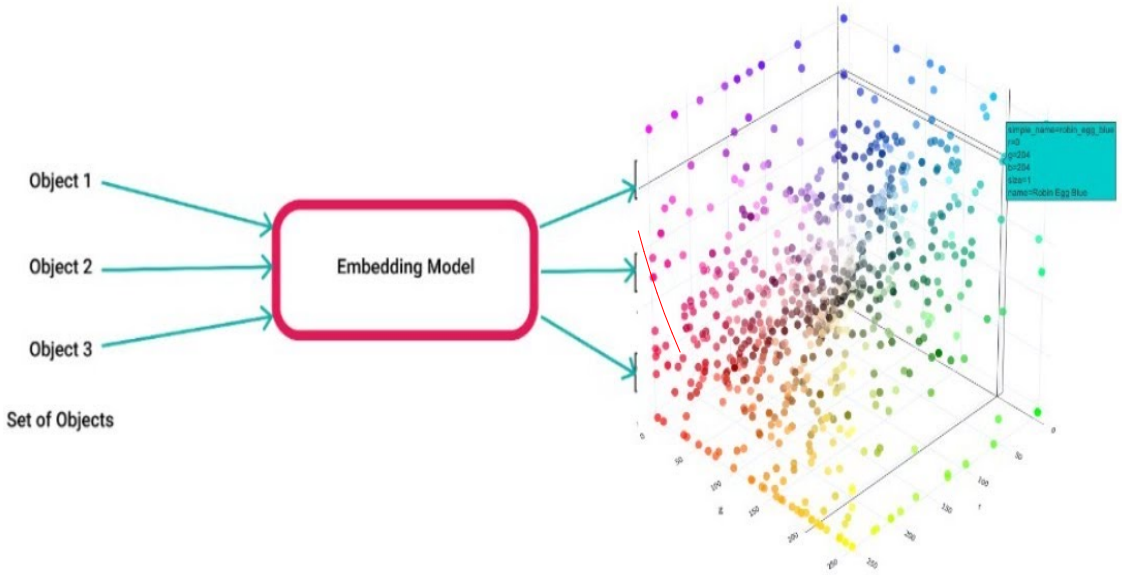
Las LSTM son una variante de las redes neuronales recurrentes (RNN) diseñada para superar el problema de desvanecimiento de gradientes en secuencias largas. Su arquitectura permite aprender dependencias a largo plazo y ha sido ampliamente utilizada en tareas de modelado de lenguaje.

```python
from keras.models import Sequential
from keras.layers import Embedding, LSTM, Dense

# Definir el modelo de lenguaje con LSTM
model_language = Sequential()
model_language.add(Embedding(input_dim=vocab_size,
output_dim=100, input_length=max_len))
model_language.add(LSTM(100))
model_language.add(Dense(vocab_size,
activation='softmax'))
```

1.10 Embeddings Preentrenados

La representación de palabras es un aspecto crucial para que las computadoras comprendan el significado de las palabras en el contexto de una tarea específica. Aunque los embeddings pueden ser aprendidos junto con el modelo, la utilización de embeddings preentrenados proporciona ventajas significativas. Dos enfoques comunes son:

Word2Vec y GloVe



1.11 Embeddings Preentrenados

Word2Vec y GloVe son técnicas que asignan vectores de alta dimensionalidad a palabras de manera que palabras similares tengan vectores cercanos. Estos modelos capturan relaciones semánticas y similitudes entre palabras.

```python
from gensim.models import Word2Vec

# Entrenar un modelo Word2Vec
model = Word2Vec([tokens], vector_size=10, window=3, min_count=1, workers=4)
word_vectors = model.wv
print("Embedding de 'procesamiento':", word_vectors['procesamiento'])
```

1.12 Embeddings Preentrenados

Embeddings de GloVe

GloVe, por otro lado, es una técnica que utiliza estadísticas de co-ocurrencia globales para generar embeddings. Puede descargar embeddings preentrenados y utilizarlos en su modelo.

```
# Descargar embeddings preentrenados de GloVe
!wget http://nlp.stanford.edu/data/glove.6B.zip
!unzip glove.6B.zip

# Cargar embeddings en memoria
embeddings_index = {}
with open('glove.6B.100d.txt', 'r', encoding='utf-8') as file:
    for line in file:
        values = line.split()
        word = values[0]
        coefs = np.asarray(values[1:], dtype='float32')
        embeddings_index[word] = coefs

# Crear matriz de embeddings para el vocabulario del conjunto de datos
embedding_matrix = np.zeros((vocab_size, 100))
for word, i in tokenizer.word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector
```

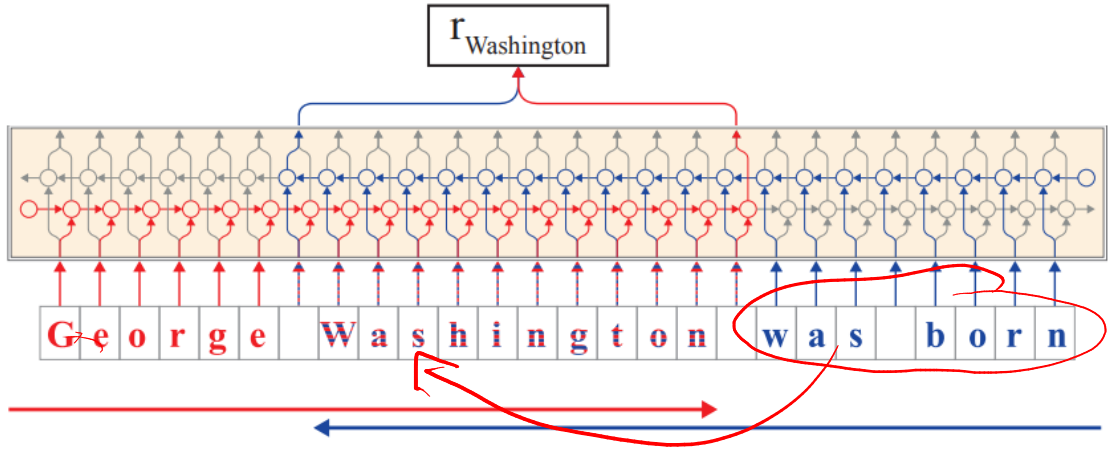
1.13 Uso de Embeddings en Modelos

Una vez que se han obtenido embeddings preentrenados, se pueden utilizar para inicializar las capas de embeddings en los modelos de lenguaje. Esto es especialmente útil cuando el conjunto de datos de interés es pequeño y no es suficiente para aprender embeddings efectivos desde cero.

```
# Uso de embeddings preentrenados en Keras
from keras.layers import Embedding

# Inicializar capa de embedding con embeddings preentrenados
model.add(Embedding(input_dim=vocab_size, output_dim=100, weights=[embedding_matrix], trainable=False))
```

1.14 Embeddings Contextuales

Aunque los embeddings mencionados anteriormente capturan significados de palabras en función de sus apariciones en un conjunto de datos estático, los embeddings contextuales, como BERT (Bidirectional Encoder Representations from Transformers) y GPT (Generative Pretrained Transformer), llevan esto un paso más allá.



1.15 Embeddings Contextuales

BERT (Bidirectional Encoder Representations from Transformers)

BERT, basado en la arquitectura de Transformer, captura el contexto bidireccional de las palabras en una oración. Sus embeddings son contextualmente más ricos y se han vuelto fundamentales en tareas avanzadas de NLP como la clasificación de texto y la comprensión del lenguaje.

```python
from transformers import BertTokenizer, BertModel

tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
model_bert = BertModel.from_pretrained("bert-base-uncased")

# Tokenización y obtención de embeddings con BERT
input_text = "Deep learning revolutionizes natural language processing."
input_ids = tokenizer.encode(input_text, return_tensors="pt")
output = model_bert(input_ids)
embeddings = output.last_hidden_state
```

1.16 Embeddings Contextuales

GPT (Generative Pretrained Transformer)

GPT, por otro lado, es un modelo generativo que utiliza la arquitectura Transformer. Puede generar texto coherente y contextualmente relevante, y se ha aplicado en tareas como la generación de texto creativo y la escritura autónoma.

```python
from transformers import GPT2LMHeadModel, GPT2Tokenizer

tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
model_gpt = GPT2LMHeadModel.from_pretrained("gpt2")

# Generar texto condicionalmente con GPT
input_text = "En un día soleado, un gato"
input_ids = tokenizer.encode(input_text, return_tensors='pt')
output = model_gpt.generate(input_ids, max_length=100, num_return_sequences=1)
generated_text = tokenizer.decode(output[0], skip_special_tokens=True)
print(generated_text)
```

1.17 Evaluación y Afinación de Modelos de Lenguaje

La evaluación y afinación de modelos de lenguaje son etapas críticas en el desarrollo de sistemas de procesamiento del lenguaje natural (NLP) basados en aprendizaje profundo. La efectividad de un modelo se mide por su capacidad para comprender, generar y manipular el lenguaje humano de manera coherente y útil para tareas específicas.

1.18 Evaluación y Afinación de Modelos de Lenguaje

Perplexidad

La perplexidad es una medida de la incertidumbre o la "sorpresa" asociada con la predicción de una secuencia de palabras. Para un modelo de lenguaje, se calcula como la inversa de la probabilidad de la secuencia normalizada por la longitud de la secuencia. Un modelo ideal tiene una perplexidad mínima.

```python
from math import exp, log

def calculate_perplexity(probability, length):
    return exp(-log(probability) / length)
```

1.19 Evaluación y Afinación de Modelos de Lenguaje

Precisión en Tareas Específicas

En tareas específicas como la clasificación de texto, se utilizan métricas como la precisión, la recuperación y el puntaje F1 para evaluar el rendimiento del modelo en un conjunto de datos de prueba.

```python
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

# Evaluar clasificación de texto
y_true = [1, 0, 1, 1, 0, 1]
y_pred = [1, 0, 1, 0, 0, 1]

accuracy = accuracy_score(y_true, y_pred)
precision = precision_score(y_true, y_pred)
recall = recall_score(y_true, y_pred)
f1 = f1_score(y_true, y_pred)
```

1.20 Evaluación y Afinación de Modelos de Lenguaje

a test
an example

BLEU (Bilingual Evaluation Understudy)

BLEU es una métrica comúnmente utilizada para evaluar la calidad de las traducciones en tareas de procesamiento de lenguaje natural, especialmente en modelos de traducción automática.

```
from nltk.translate.bleu_score import sentence_bleu

reference = [['this', 'is', 'a', 'test'], ['another', 'example']]
candidate = ['this', 'is', 'an', 'example']

bleu_score = sentence_bleu(reference, candidate)
```

1.21 Afinación de Modelos de Lenguaje

Ajuste de Hiperparámetros

Los hiperparámetros, como la tasa de aprendizaje, el tamaño de la red, la longitud de la secuencia y la cantidad de capas, tienen un impacto significativo en el rendimiento del modelo. Se pueden ajustar mediante técnicas como búsqueda en cuadrícula o búsqueda aleatoria.

```python
from sklearn.model_selection import GridSearchCV
from keras.models import Sequential
from keras.layers import LSTM, Dense

# Definir el modelo
model = Sequential()
model.add(LSTM(50, input_shape=(X.shape[1], X.shape[2])))
model.add(Dense(1))

# Definir parámetros a ajustar
param_grid = {'batch_size': [32, 64, 128], 'epochs': [10, 20, 30]}

# Configurar búsqueda en cuadrícula
grid_search = GridSearchCV(estimator=model, param_grid=param_grid, scoring='accuracy', cv=3)
grid_result = grid_search.fit(X, y)
```

1.22 Afinación de Modelos de Lenguaje

Aumento de Datos

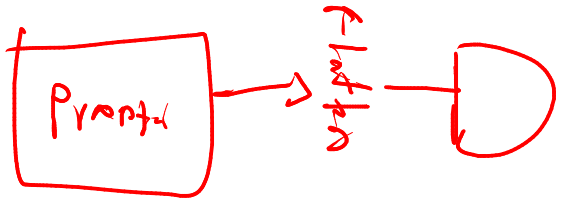
El aumento de datos implica generar variantes del conjunto de datos original mediante transformaciones aleatorias. En el contexto de modelos de lenguaje, esto puede implicar agregar ruido, cambiar el orden de las palabras o introducir variaciones sintácticas.

```python
import nlpaug.augmenter.word as naw

# Augmentación de texto con nlpaug
aug = naw.ContextualWordEmbsAug()
augmented_text = aug.augment("El procesamiento del lenguaje natural es fascinante.")
```

1.23 Afinación de Modelos de Lenguaje

Fine-Tuning y Transfer Learning

Fine-tuning implica ajustar un modelo preentrenado en un conjunto de datos específico para una tarea particular. Transferir el conocimiento de un modelo entrenado en una tarea relacionada a otra puede acelerar el aprendizaje.

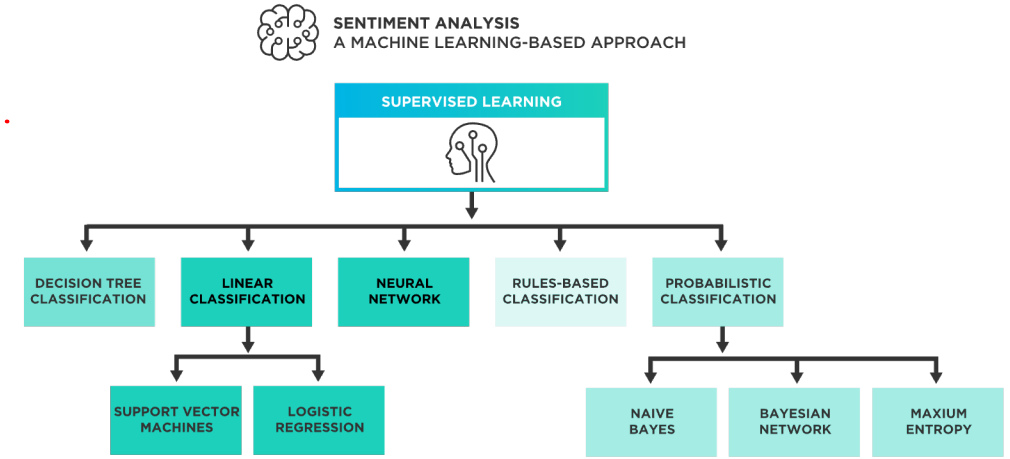



```python
from keras.applications import VGG16
from keras.models import Sequential
from keras.layers import Dense, Flatten

# Cargar modelo preentrenado
base_model = VGG16(weights='imagenet',
include_top=False, input_shape=(224, 224, 3))

# Agregar capas personalizadas
model = Sequential()
model.add(base_model)
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

# Compilar y entrenar el modelo
model.compile(optimizer='adam',
loss='binary_crossentropy', metrics=['accuracy'])
model.fit(X_train, y_train, epochs=10, validation_data=
(X_val, y_val))
```

1.24 Clasificación de Sentimientos

La clasificación de sentimientos es una tarea clave en NLP que implica determinar la polaridad emocional de un texto, clasificándolo como positivo, negativo o neutro.

Es una aplicación valiosa en campos como las redes sociales, el análisis de comentarios de productos y la retroalimentación del cliente. Los modelos de aprendizaje profundo han demostrado ser altamente efectivos en esta tarea, capturando matices semánticos y contextuales.



1.25 Modelo LSTM para Clasificación de Sentimientos

Un enfoque común es utilizar redes neuronales recurrentes (RNN), específicamente Long Short-Term Memory (LSTM), para la clasificación de sentimientos. Las capas LSTM son capaces de capturar dependencias temporales en secuencias de texto, lo que las hace ideales para este propósito.

```
from keras.models import Sequential
from keras.layers import Embedding, LSTM, Dense

# Crear modelo de clasificación de sentimientos con LSTM
model_sentiment = Sequential()
model_sentiment.add(Embedding(input_dim=vocab_size, output_dim=100, input_length=max_len, weights=[embedding_matrix], trainable=False))
model_sentiment.add(LSTM(100))
model_sentiment.add(Dense(1, activation='sigmoid'))
model_sentiment.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

1.26 Importancia de los Embeddings Preentrenados

En este código, se utiliza una capa de embedding preentrenada con vectores GloVe. La ventaja de los embeddings preentrenados radica en la capacidad de capturar relaciones semánticas y contextuales entre palabras, incluso cuando el conjunto de datos de entrenamiento es limitado. Esto permite que el modelo comprenda mejor las sutilezas del lenguaje.

```python
from keras.models import Sequential
from keras.layers import Embedding, LSTM, Dense

# Crear modelo de clasificación de sentimientos con LSTM
model_sentiment = Sequential()
model_sentiment.add(Embedding(input_dim=vocab_size, output_dim=100, input_length=max_len, weights=[embedding_matrix], trainable=False))
model_sentiment.add(LSTM(100))
model_sentiment.add(Dense(1, activation='sigmoid'))
model_sentiment.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

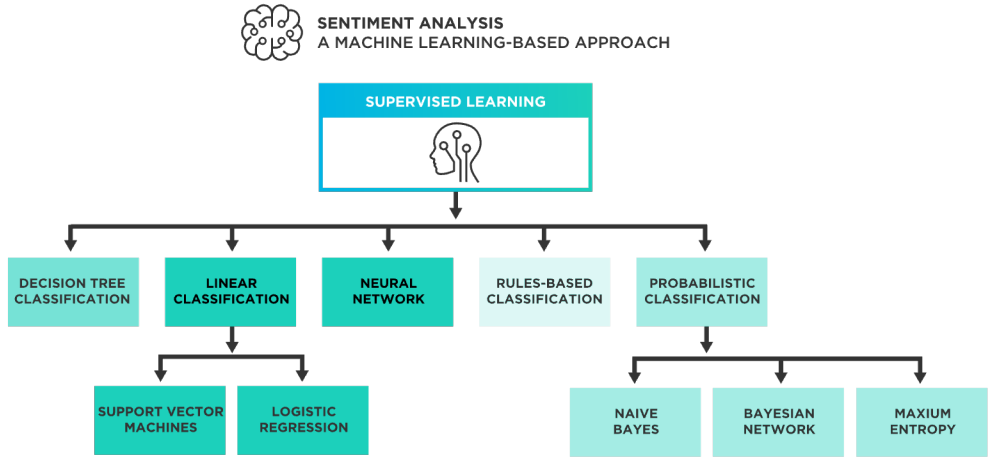
1.27 Consideraciones sobre Overfitting y Regularización

Dado que los modelos de clasificación de sentimientos pueden enfrentarse a conjuntos de datos desequilibrados y sesgados, es crucial considerar estrategias de regularización para evitar el sobreajuste. El uso de técnicas como Dropout durante el entrenamiento puede mejorar la generalización del modelo.

```
from keras.layers import Dropout

# Agregar Dropout para combatir el sobreajuste
model_sentiment.add(Dropout(0.2))
```

1.28 Generación de Texto - Desafios

La generación de texto implica la creación de contenido coherente y relevante. Es una tarea compleja que va más allá de simplemente predecir palabras en una secuencia; implica comprender y sintetizar información para producir texto significativo. Las arquitecturas de Transformer, como GPT (Generative Pretrained Transformer), han logrado avances significativos en esta área.

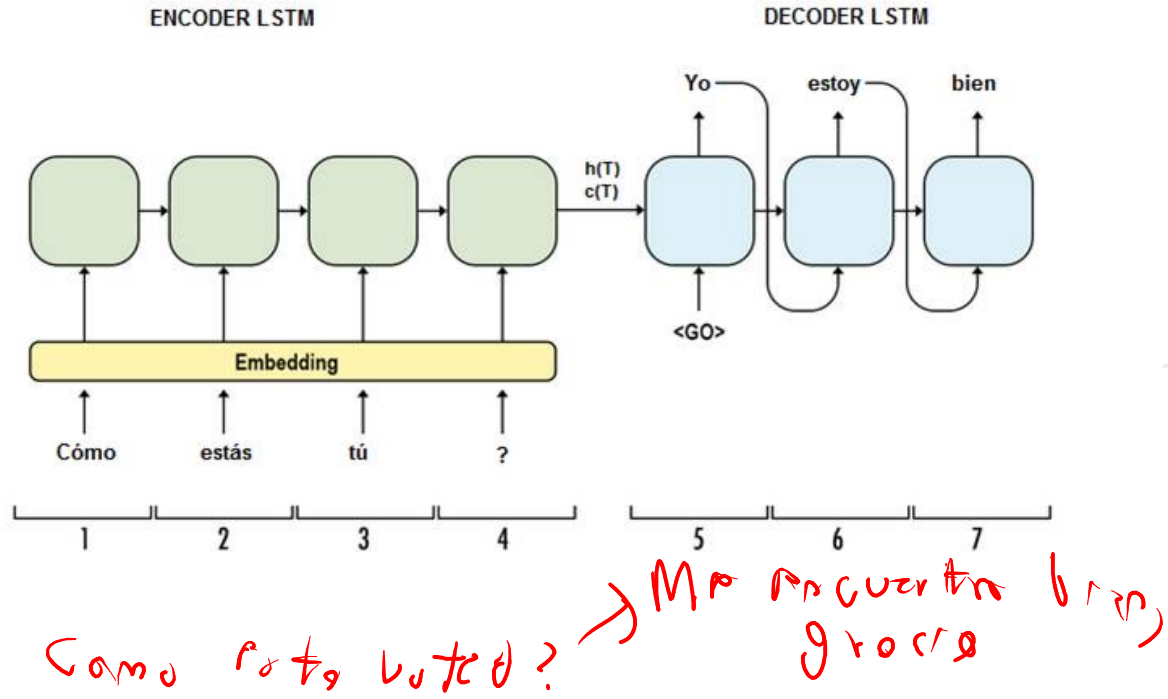


1.29 Modelo GPT para Generación de Texto

```
from transformers import GPT2LMHeadModel, GPT2Tokenizer

# Cargar modelo preentrenado GPT-2
tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
model_generation = GPT2LMHeadModel.from_pretrained("gpt2")

# Generar texto condicionalmente
input_text = "En un día soleado, un gato"
input_ids = tokenizer.encode(input_text, return_tensors='pt')
output = model_generation.generate(input_ids, max_length=100, num_return_sequences=1)
generated_text = tokenizer.decode(output[0], skip_special_tokens=True)
print(generated_text)
```

1.30 Transferencia de Estilo y Contenido

Los modelos de generación de texto avanzados no solo generan texto coherente, sino que también pueden transferir estilos y contenidos específicos. Esto se logra mediante la manipulación de vectores de atención, que indican qué partes del texto deben recibir más énfasis.



Preguntas