Bootcamp Inteligencia Artificial
Nivel Innovador

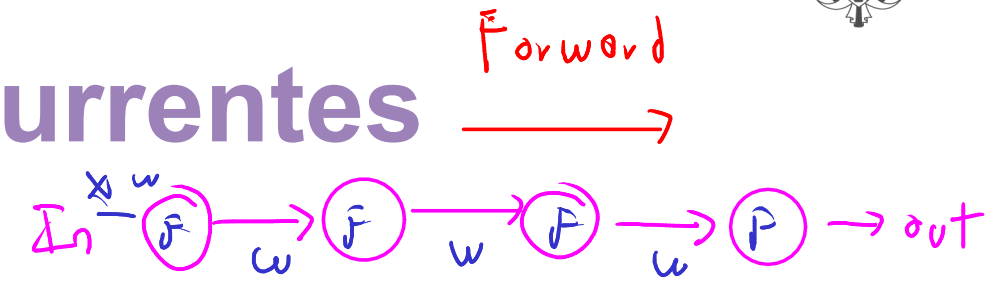
Master Class 3:

Aplicaciones de Deep Learning

Agenda

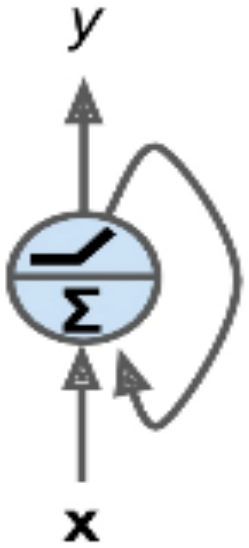
1. Redes Neuronales Recurrentes
2. Procesamiento de Lenguaje Natural
3. Autoencoders y GANs

1.1 Neuronas y Capas Recurrentes



Hasta ahora, todas las redes neuronales que hemos visto fluyen en una sola dirección: hacia adelante. La señal entra por un lado y es procesada por capas densas de neuronas, capas de convolución, capas de activación, capas de normalización, lo que gustes y mandes, y al final obtienes tus resultados del otro lado de la red.

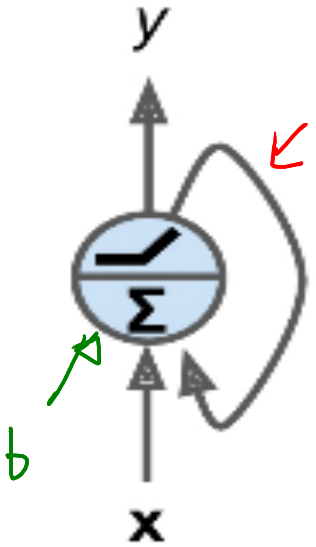
Pero ahora vamos a ver que en las Redes Neuronales Recurrentes (RNN), la señal fluye… repetidamente a través de la misma capa y las mismas neuronas. Veamos al RNN más sencillo de lo que nos podemos imaginar.



1.2 Neuronas y Capas Recurrentes

Primero, tenemos una sola neurona, con sus pesos lineales de un lado, su función de activación del otro. A esta Neurona le vamos a meter nuestra señal X – ahora en una red neuronal normal, esperaríamos que la procese su función lineal, luego su función de activación, y nos escupa los resultados por acá arriba.

No en esta RNN – va a pasar por los procesos que mencionamos antes, y la salida que escupe esta neurona vuelve a introducirse a la neurona, una y otra y otra y otra y otra vez.



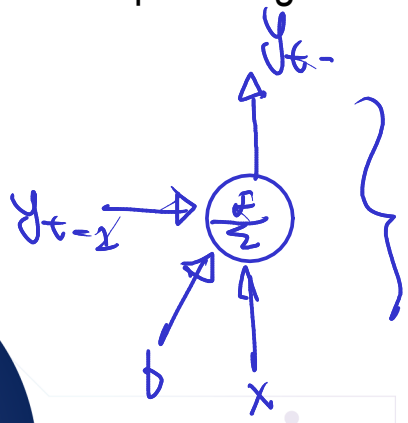
$$y = f(\bar{x})$$

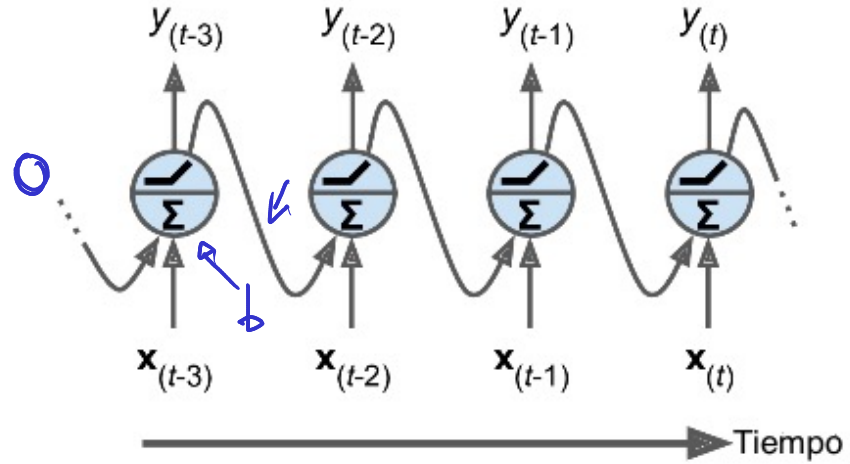
$$\bar{x} = wx + b + y_{t-1}$$

1.3 Neuronas y Capas Recurrentes

Se ve sencillo, pero representar las RNNs de esta manera no es práctico. Recuerda que al final de cuentas vas a estar leyendo papers sobre esta clase de temas, y las publicaciones científicas no van a traer animaciones profesionalmente hechas para ilustrar los más nuevos avances en RNNs.

Así que en realidad vamos a visualizar nuestros RNNs desenrollados a lo largo del tiempo: Imaginemos que haremos esta repetición 4 veces.





$$y_{t-3} = f(w_x x_{t-3} + b + w_y 0)$$

$$y_{t-2} = f(w_x x_{t-2} + b + w_y y_{t-3})$$

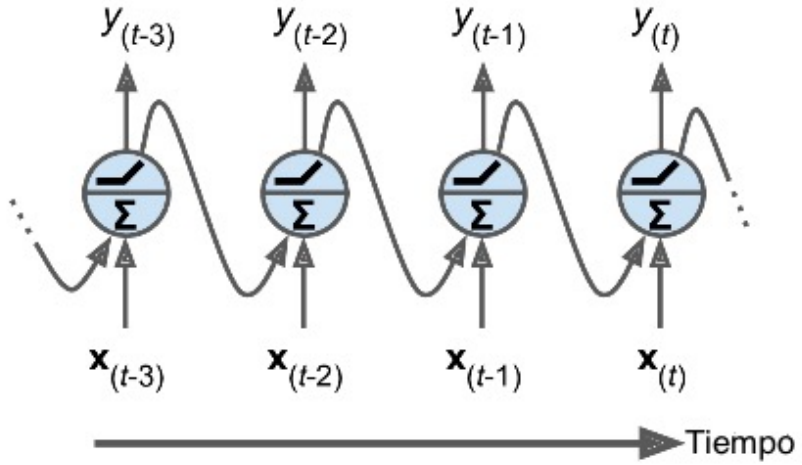
$$y_t = f(w_x x_t + b + w_y y_{t-1})$$

1.4 Neuronas y Capas Recurrentes

Entonces lo que haríamos sería dibujar nuestra neurona 1 vez y definiremos este dibujo como el cuadro o “frame” (en inglés, acostúmbrese que le diré frame) número t-3.

Esto por que como vamos a ciclar la señal, o repetir la señal, como quieran verlo, 4 veces, vamos a tener el cuatro t-3, t-2, t-1 y t.

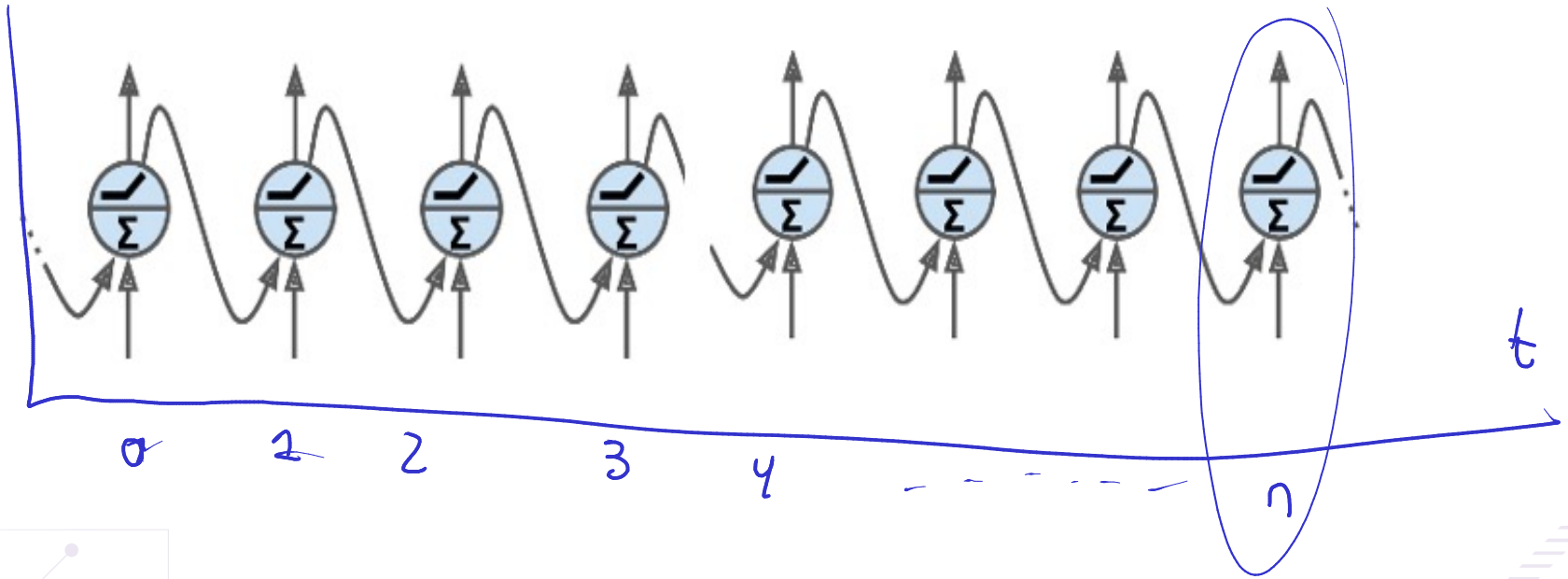
Tenemos nuestro frame t-3, le metemos la señal X(t-3) y nos va a escupir el resultado Y(t-3). Ahora dibujamos la misma neurona – mismos pesos (W), misma función de activación, aquí a la derecha. Vamos a meterle de nuevo nuestra señal original X (aunque ahora será t-2) y también recibirá la salida del frame anterior. Esto es lo mismo que, en el diagrama original, ciclar nuestra señal a través de la misma neurona.



1.5 Neuronas y Capas Recurrentes

Volviendo al frame t-2, la neurona nos va a escupir la señal. Pasamos ahora al frame t-2, donde tenemos nuestra misma neurona, y vuelve a recibir la exacta misma entrada X y aparte, el resultado del frame anterior t-3 … y por último repetimos el proceso en el momento t.

La exacta misma neurona, recibe la exacta misma entrá X y aparte el resultado de t-1 y finalmente nos escupe una salida. Podríamos, si quisiéramos, repetir este proceso más de 4 veces, en cuyo caso solo tendríamos que seguir agregando neuronas a nuestro diagrama.

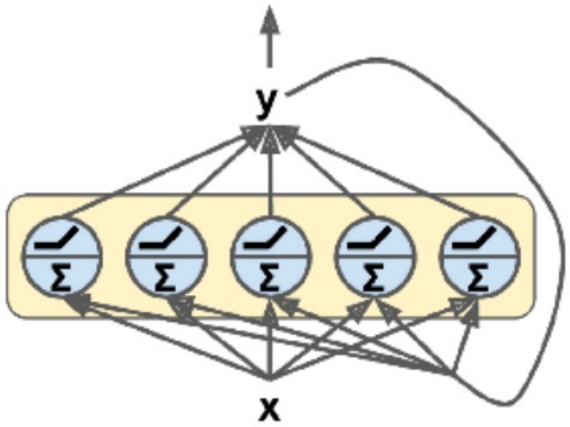


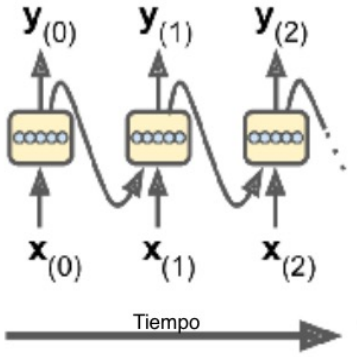
1.6 Neuronas y Capas Recurrentes

Aunque en esencia, es solo estar ciclando la señal a través de la misma neurona tantas veces como lo creamos necesario. Este modelo es sencillo – solo es una neurona. Vamos a echarle un poquito de peligro.

¿Qué pasa si yo quiero una capa entera con 5 neuronas? Fácil, voy a plantear mis 5 neuronas aquí.

Ingresando toda a la misma entrada X – las neuronas las procesaran con sus respectivos pesos W y su función de activación, y obtendremos nuestra salida Y. Esta salida es un vector.



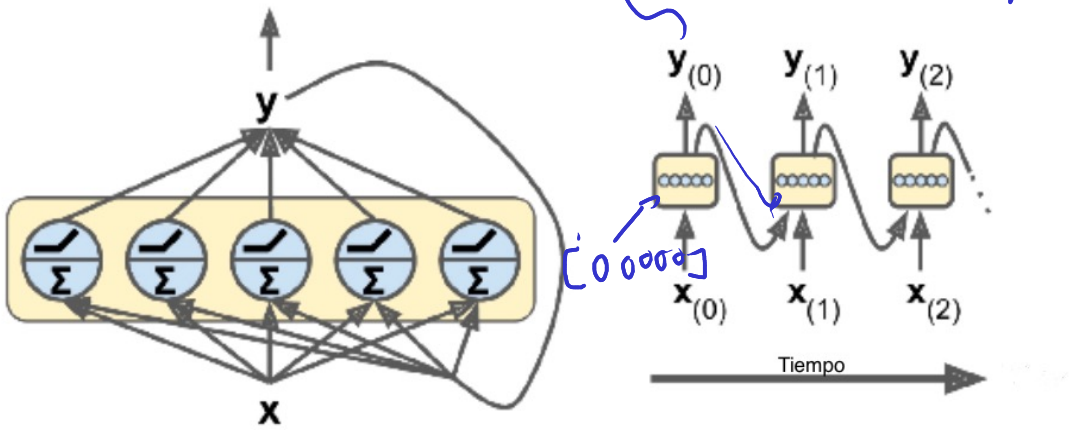


1.7 Neuronas y Capas Recurrentes

Para este primer ciclo obtendremos nuestro vector de salida Y, y lo usaremos para volver a alimentar a nuestras mismas neuronas. Y entonces comienza el ciclo de repetición, donde en cada fase, estaremos alimentando las entradas con el mismo vector de salida Y. Así repetimos hasta que terminemos todos los ciclos de repetición.

¿Cómo dibujamos esta clase de redes neuronales recurrentes en su diagrama por frames? Fácil, en vez de poner la neurona, simplemente diagramamos la capa. El proceso es el mismo, solo que ahora, por decisión mía totalmente, en vez de usar la nomenclatura t-1, vamos a comenzar en el paso 0.

Así que esta es nuestra capa en el paso 0, entra el vector X y sale el Vector Y0 – luego pasamos a la misma capa en el frame 1, donde entra el vector X1, y la salida anterior que habíamos obtenido (Y0).



$$y(1) = F(X_1 w_x + y(0) w_y + b)$$

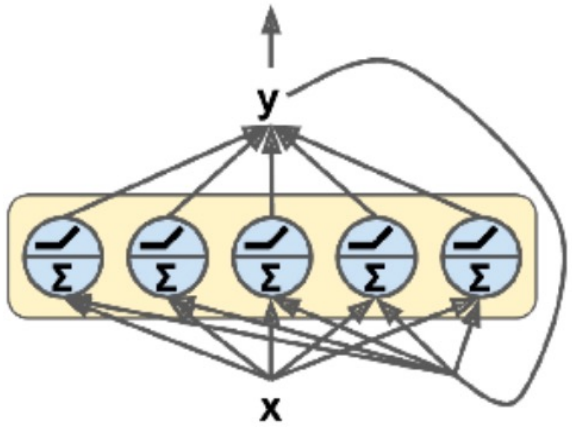
$$y(i) = F(X_i w_x + y(i-1) w_y + b)$$

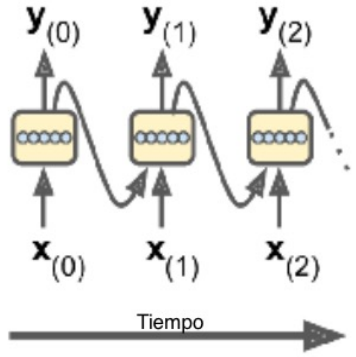
1.8 Neuronas y Capas Recurrentes

Y luego pasamos a Y2 donde hacemos el mismo procedimiento, y así consecutivamente hasta completar el número de repeticiones necesarias.

Cada neurona recurrente tiene 2 sets de pesos: una para las entradas x_yotra para las salidas que recibió del paso de tiempo anterior y_((t-1)). Los vectores se van a llamar los pesos w_x y w_y

Si consideramos toda la capa recurrente entonces podemos colocar todos los pesos de cada una de las neuronas en 2 matrices de pesos W_x y W_ypero con mayúsculas.





1.9 Neuronas y Capas Recurrentes

El vector de salida de toda la capa recurrente entonces puede ser computado como lo vemos en la siguiente ecuación:

$$y_t = \phi(W_x^\intercal x_t + W_y^\intercal y_{(t-1)} + b)$$

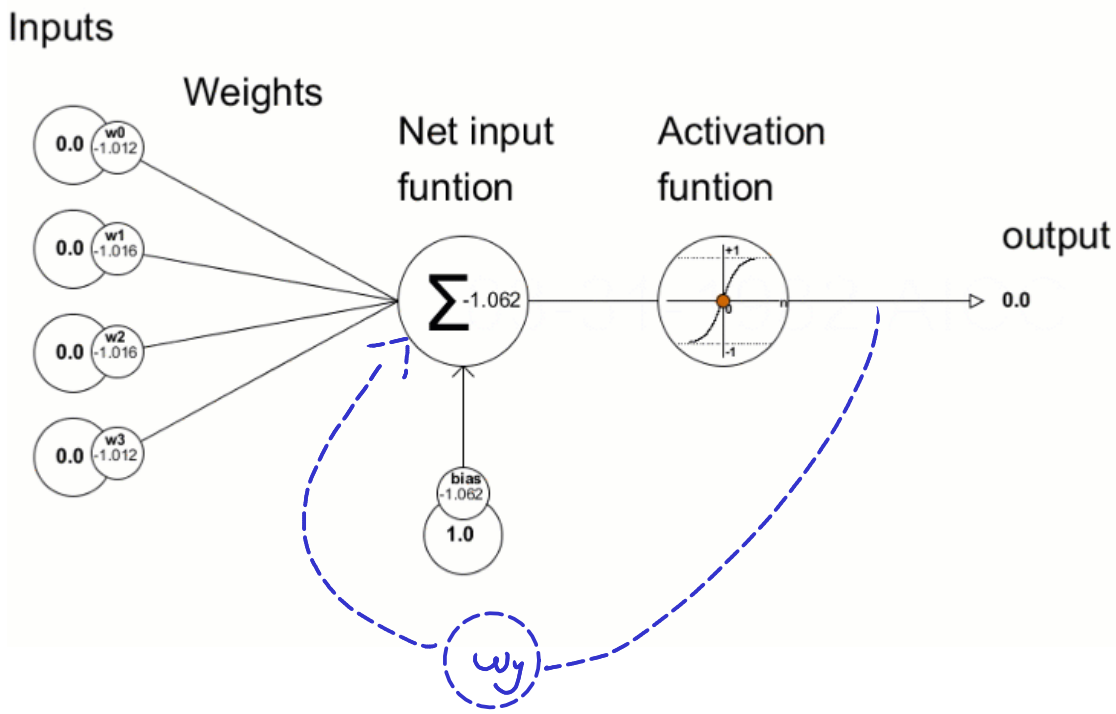
Agarramos los valores de W_x y obtenemos su producto punto con respecto a las entradas de x. Luego agarramos los valores de W_y y obtenemos su producto punto con y(t-1). Sumamos el resultado de esas 2 operaciones y aparte le sumamos el término de bias que nunca puede faltar.

Al final, obtenemos y_t de resultado, que se utilizará en la siguiente iteración del ciclo para alimentar a la misma capa.

1.10 Celdas de memoria

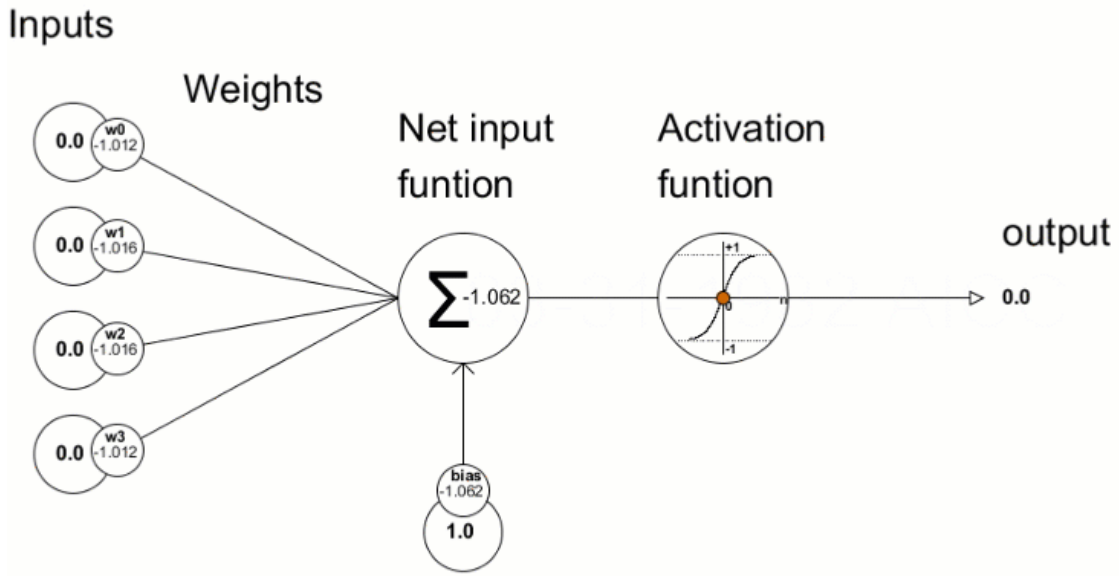
Vamos pensando en la última neurona de una red neuronal convolucional. Su resultado va a depender de la entrada X que le metamos, y al mismo tiempo de la entrada Y que viene de la neurona anterior.

Asimismo, el resultado de esa neurona anterior va a depender de la entrada que le meta la neurona anterior, o sea esta neurona que está a 1 paso removida también está influyendo nuestro resultado de la neurona final. Y, al mismo tiempo, el resultado de la neurona anterior anterior va a depender de lo que sea que le pase la neurona que va en el paso antes de este, y así consecutivamente.



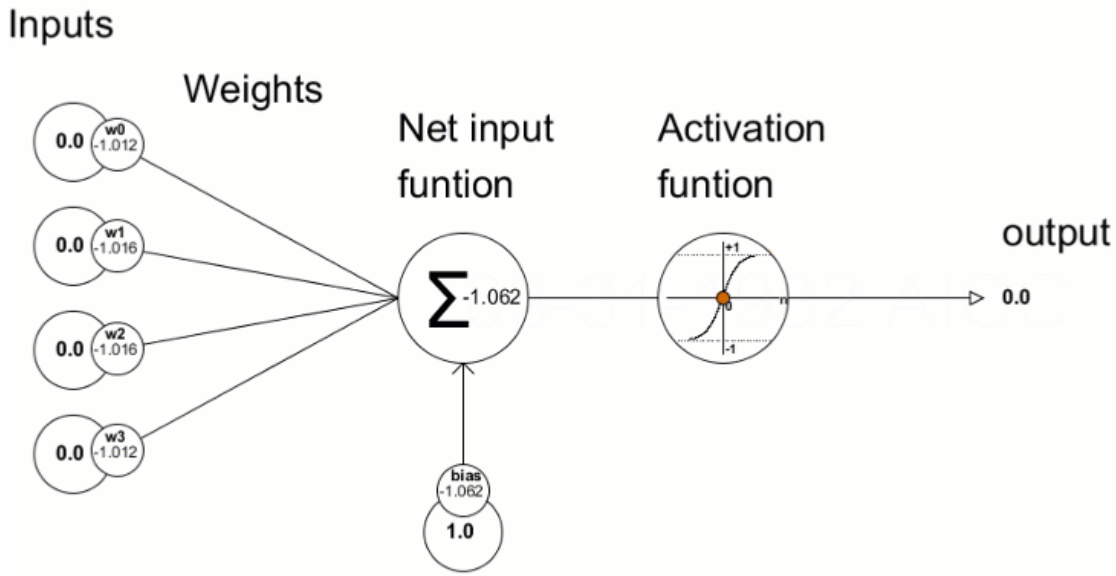
1.11 Celdas de memoria

El resultado de la neurona final va a ser influenciado por todas las neuronas, o todos los pasos anteriores, cada vez en menor medida claro, pero la influencia está ahí. Incluso si tenemos una red neuronal recurrente de 20 ciclos, la neurona que representa el ciclo uno le manda una señal a nuestra última neurona que, muy débilmente, está representada en nuestra salida.



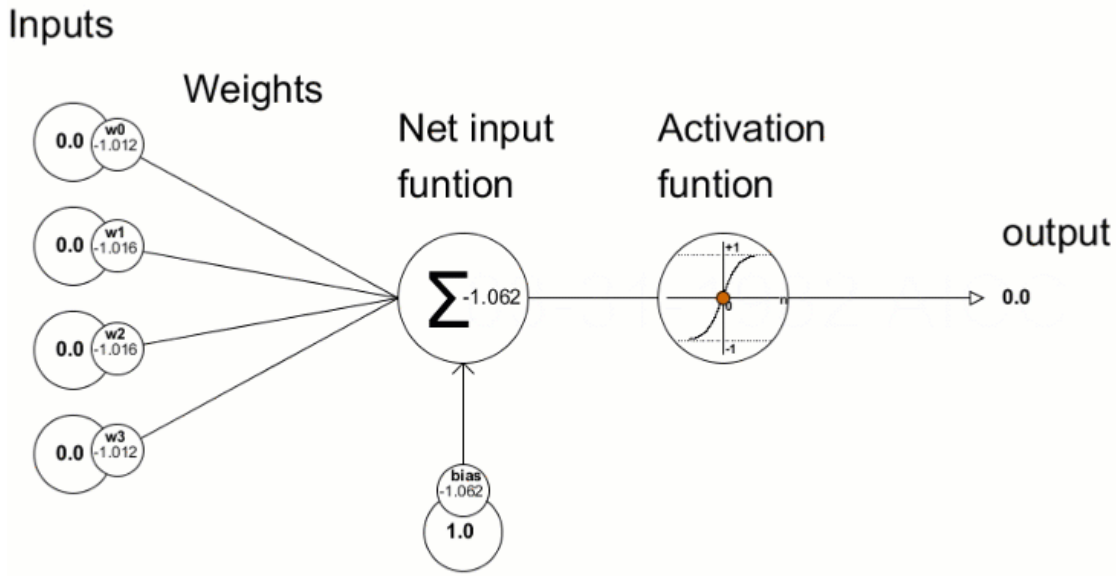
1.12 Celdas de memoria

Tomemos nuestra neurona – ya sabemos que la entrada es x y la salida es y. Pero el estado oculto, ese ciclo de volverle a meter su salida en sí misma, la conoceremos como h_t que es el estado de h en algún momento t (recuerda que la t denota el ciclo de recurrencia en el que vamos) es una función de algunas entradas en ese momento (como x_t) y su estado en el momento previo h_(t-1) entonces podríamos decir que h_t=f(h_(t-1) ,x_t).



1.13 Celdas de memoria

Su salida y_t, es una función del estado previo y sus entradas actuales. ¿Por qué todo este lío con el álgebra? Porque en las celdas básicas que vimos en la introducción, la salida y simplemente es igual al estado h, ambas son salidas de la neurona aplicando su función lineal de pesos y su función de activación. Pero en unos momentos veremos que NO en todos los casos aplica que y es igual a h.

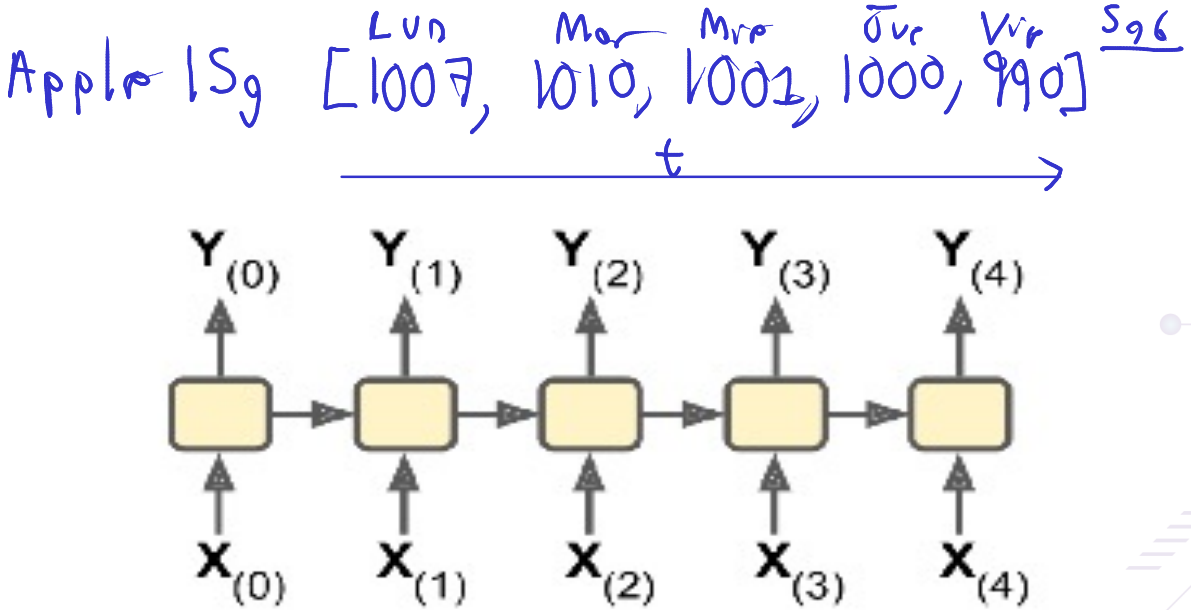


1.14 Secuencias de Entrada y de Salida

Imaginemos que estamos intentando utilizar nuestras redes neuronales para predecir el mercado. Tenemos la lista de precios de Amazon (AMZN) a lo largo de los últimos 5 días de trading y queremos que la red neuronal recurrente nos diga cómo va a amanecer al día siguiente.

Para entrenar nuestra red neuronal le meteremos una secuencia de precios a lo largo de los últimos 5 días, y esperaríamos de regreso los precios, pero adelantados un día hacia el futuro, o sea desde hace 4 días hasta mañana.

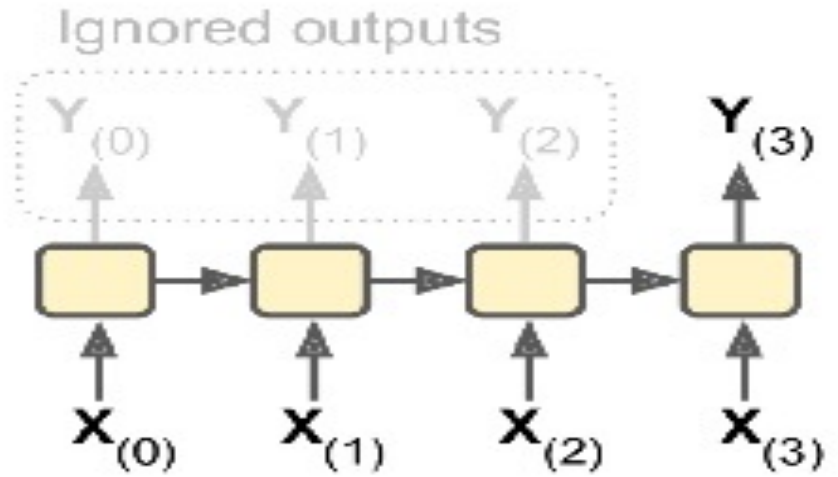
Esta clase de RNN básicas son las redes neuronales que se les conoce como secuencia a secuencia. Le metemos una secuencia, le sacamos una secuencia.



1.15 Secuencias de Entrada y de Salida

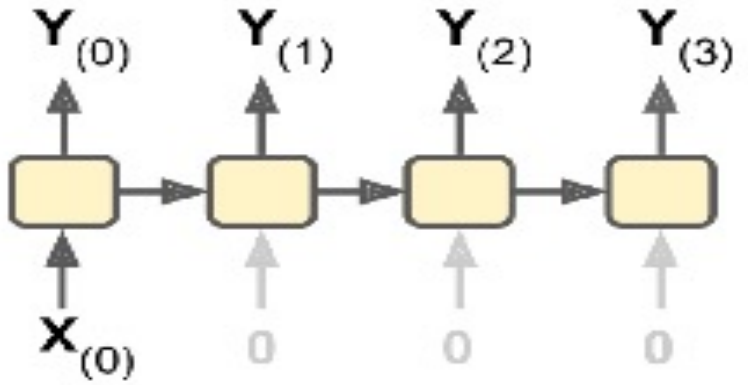
Por otro lado, tal vez queremos hacer un medidor de sentimiento – alimentarle, por ejemplo, las noticias del día y que nos diga si son noticias positivas (+1) o negativas (-1). En este caso usaríamos una arquitectura que se llama secuencia a vector.

Lo que hacemos es alimentar la RNN con una secuencia de entradas e ignoramos todas las salidas excepto la última. Al final de cuentas solo necesitamos una conclusión – si la noticia es buena o mala.



1.16 Secuencias de Entrada y de Salida

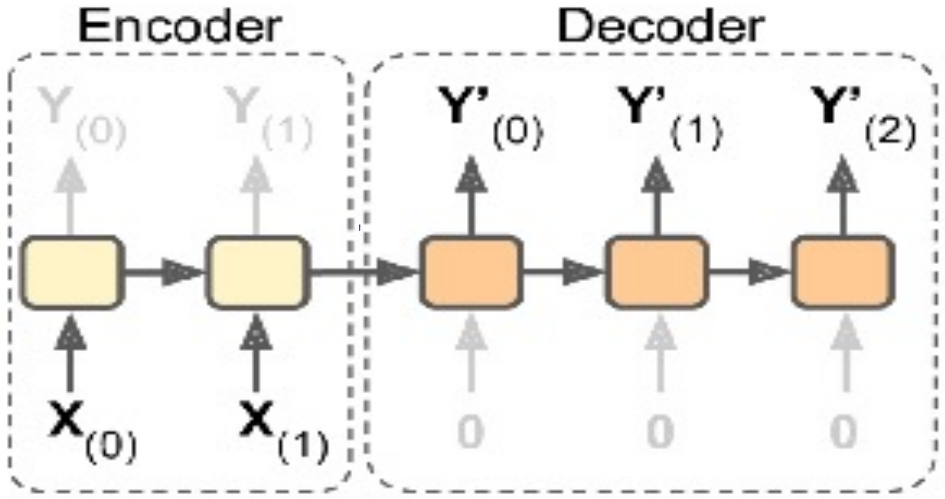
Ahora, vamos a suponer que queremos hacer lo contrario – vamos a alimentarle un vector una sola vez y dejar que saque una secuencia. Este modelo de vector a secuencia sería útil si le damos una imagen y queremos que nos escupa una descripción de la imagen.

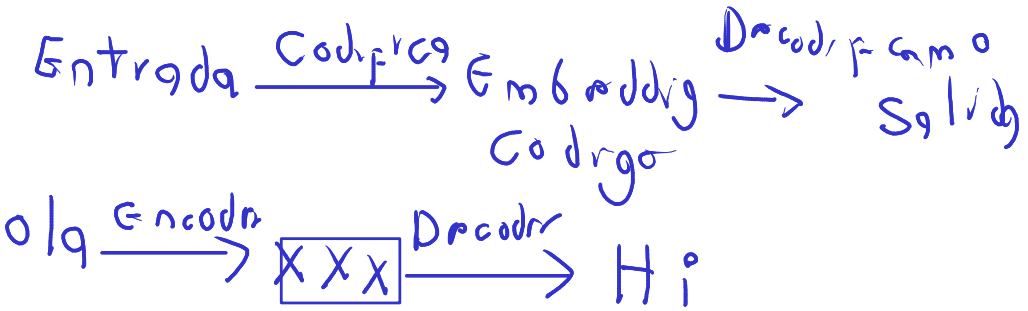


1.17 Secuencias de Entrada y de Salida

Y, por último, viene la arquitectura más interesante de todas: los codificadores – decodificadores.

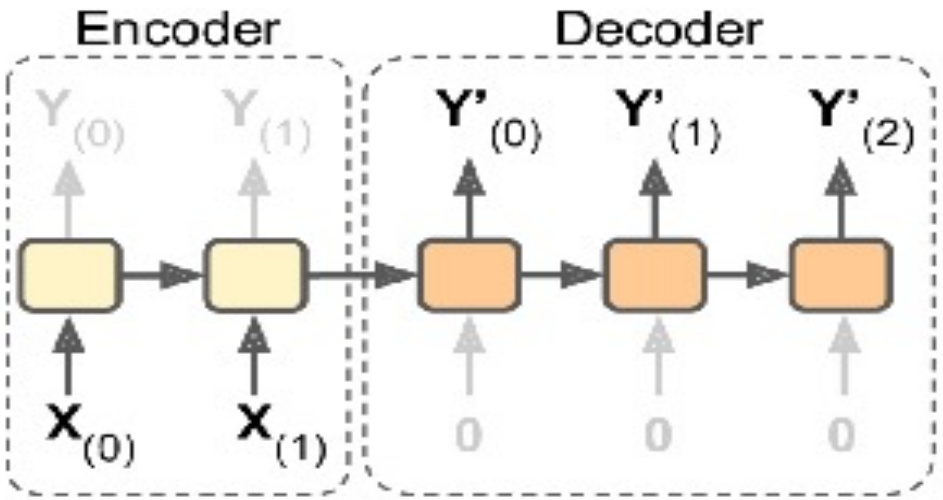
Esta estructura se usa para hacer traducciones. En la parte de codificación de la RNN le daríamos una oración en español, por ejemplo, y el RNN la convertiría en un simple vector. Después la pasaría a la sección descodificadora de la red, en donde la RNN agarra el vector de la oración en español y lo decodifica en una oración en inglés.





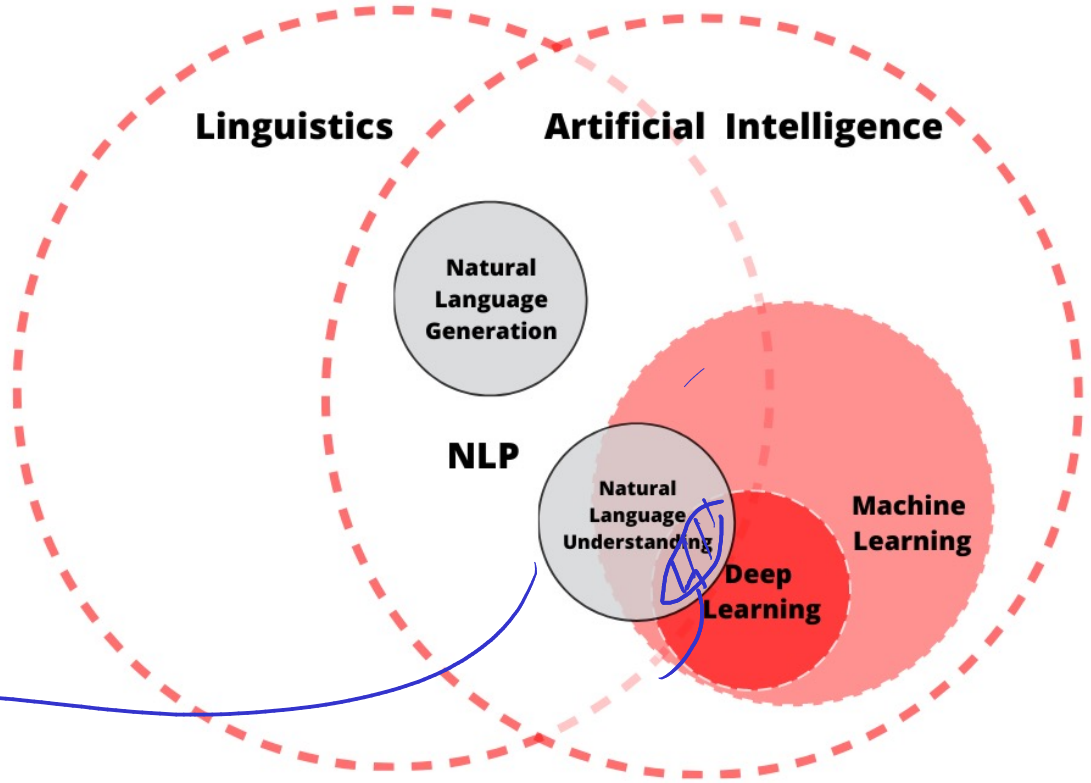
1.18 Secuencias de Entrada y de Salida

Esto funciona mucho mejor que intentar traducir en tiempo real palabra por palabra porque la gramática y estructura de los idiomas son diferentes. Por ejemplo, en el alemán la palabra que dices al final de una oración puede cambiar completamente el significado del mensaje – así que tienes que esperarte al final de la oración para asegurarte que captaste todo.



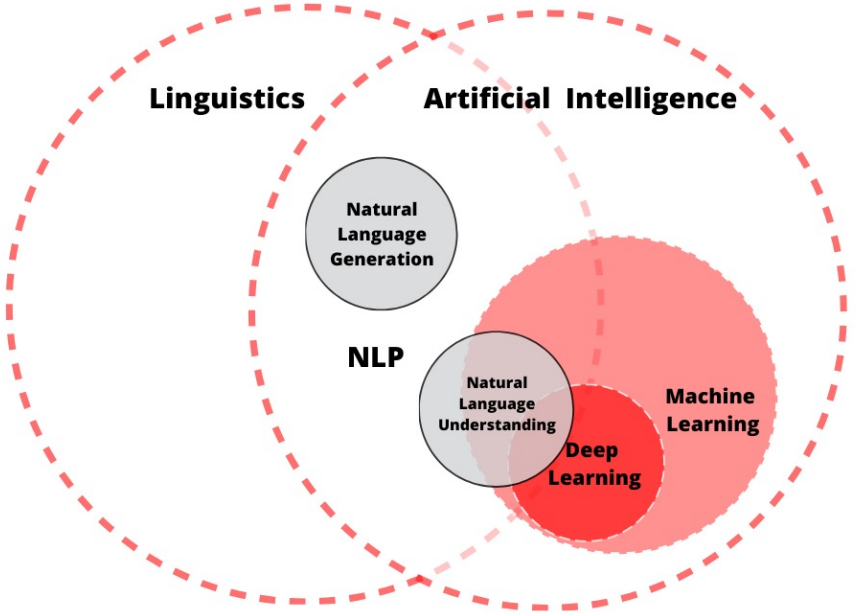
2.1 Introducción

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.



2.2 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.



2.3 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.

2.4 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.**

Limitaciones one hot encoding
1. El número de columnas es mucho
2. No es posible mapear relaciones entre palabras

id	color
1	red
2	blue
3	green
4	blue

One Hot Encoding

id	color_red	color_blue	color_green
1	1	0	0
2	0	1	0
3	0	0	1
4	0	1	0

2.5 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.

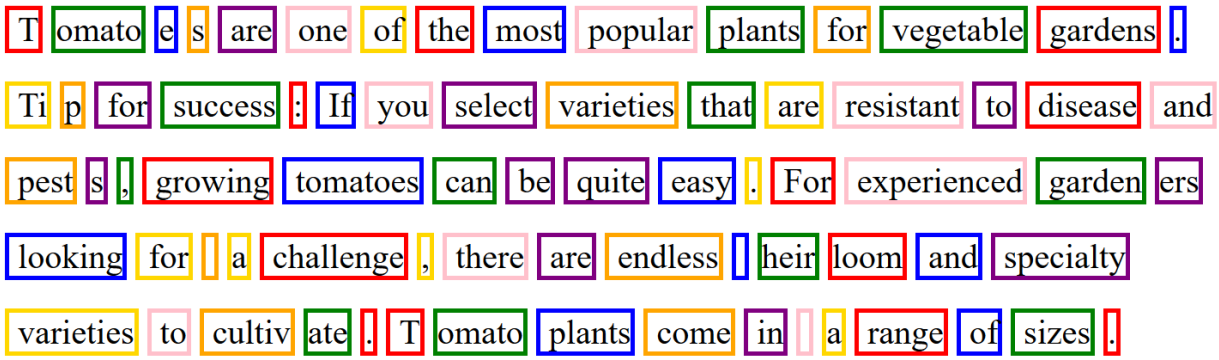
En este código, estamos entrenando un modelo Word2Vec con la lista de tokens que obtuvimos previamente. Luego, podemos acceder al embedding de una palabra específica, en este caso, 'procesamiento', para ver la representación vectorial aprendida.

```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'])
```

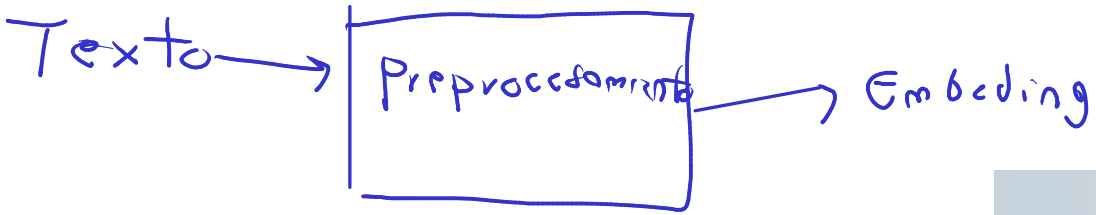
2.6 Tokenización y Representación de Palabras

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.



2.7 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.

2.8 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.

El mira un pajaro

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

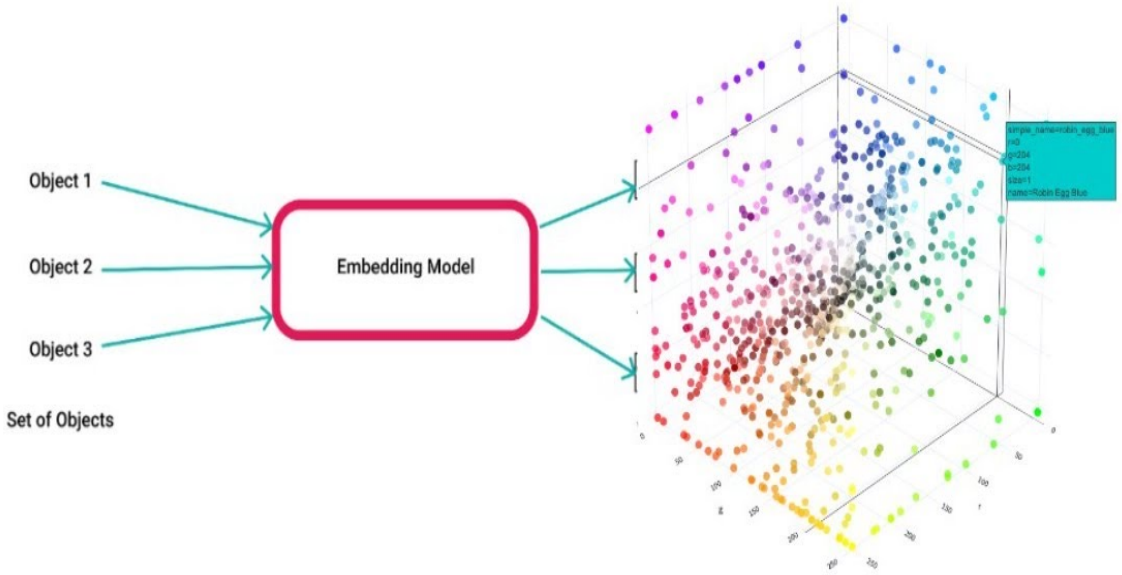
Las LSTM son una variante de las redes neuronales recurrentes (RNN) diseñada para superar el problema de desvanecimiento de gradientes en secuencias largas. Su arquitectura permite aprender dependencias a largo plazo y ha sido ampliamente utilizada en tareas de modelado de lenguaje.

```
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'))
```

2.10 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



2.11 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'])
```

2.12 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
```

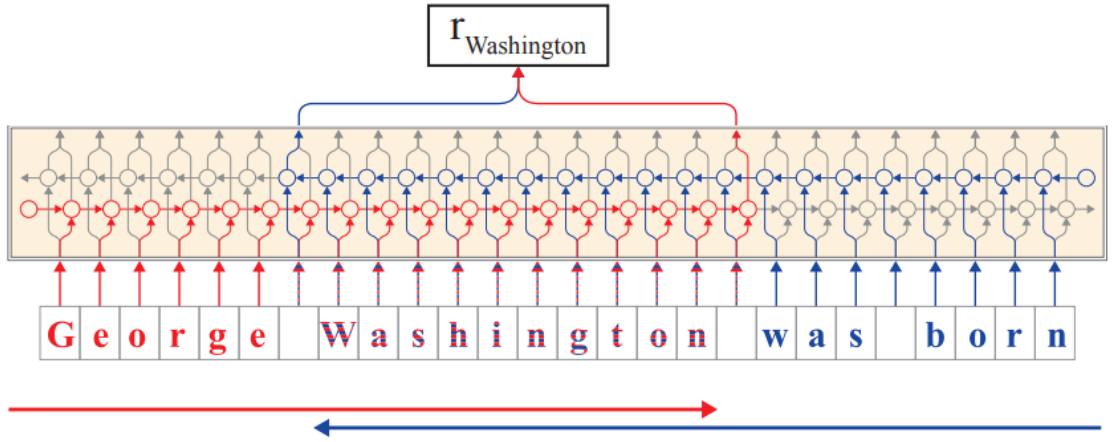
2.13 Uso de Embeddings en Modelos

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))
```

2.14 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.



2.15 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
```

2.16 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)
```

2.17 Evaluación y Afinación de Modelos de Lenguaje

hiperparametros: Hay que elegirlos sabiamente

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.

2.18 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)
```

2.19 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)
```

2.20 Evaluación y Afinación de Modelos de Lenguaje

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.

```python
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)
```

2.21 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)
```

2.22 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.")
```

2.23 Afinación de Modelos de Lenguaje

Fine-Tuning y Transfer Learning

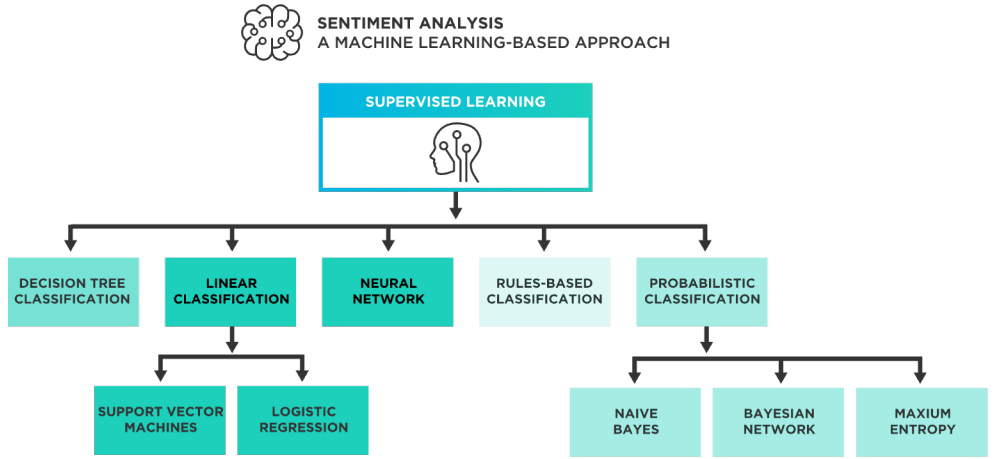
Fine-tuning implica ajustar un modelo preentrenado en un conjunto de datos específico para una tarea particular. Transferir el conocimiento de un modelo entrenado en una tarea relacionada a otra puede acelerar el aprendizaje.

```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))
```

2.24 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.

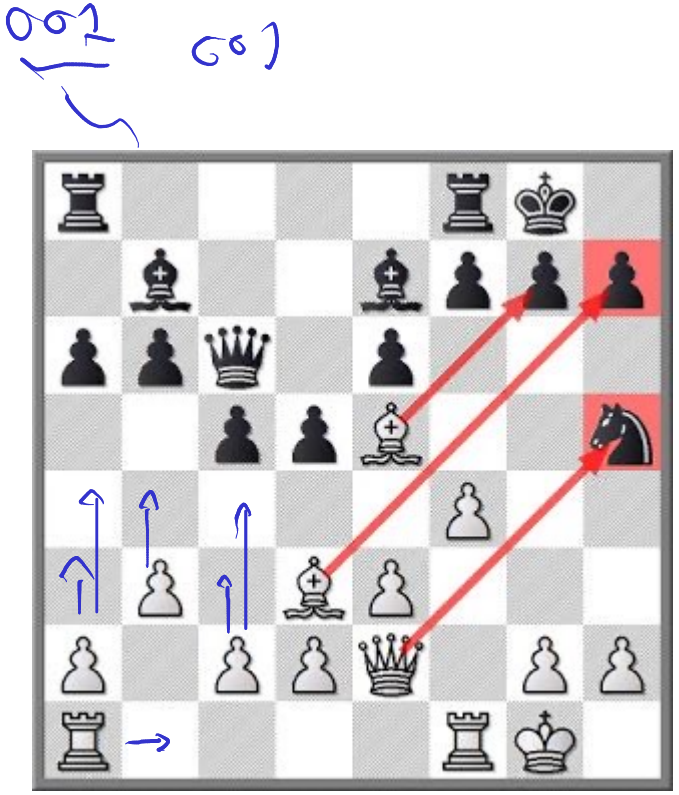


.1 Introducción

Una computadora no tiene problemas de memoria como nosotros los humanos – las computadoras pueden memorizar secuencias extraordinariamente largas, para eso las usamos.

Pero mediante el autoencoder, obligamos a la computadora a que NO memorice, sino a que busque algún patrón o alguna regla que le permita rearmar los datos desde el principio.

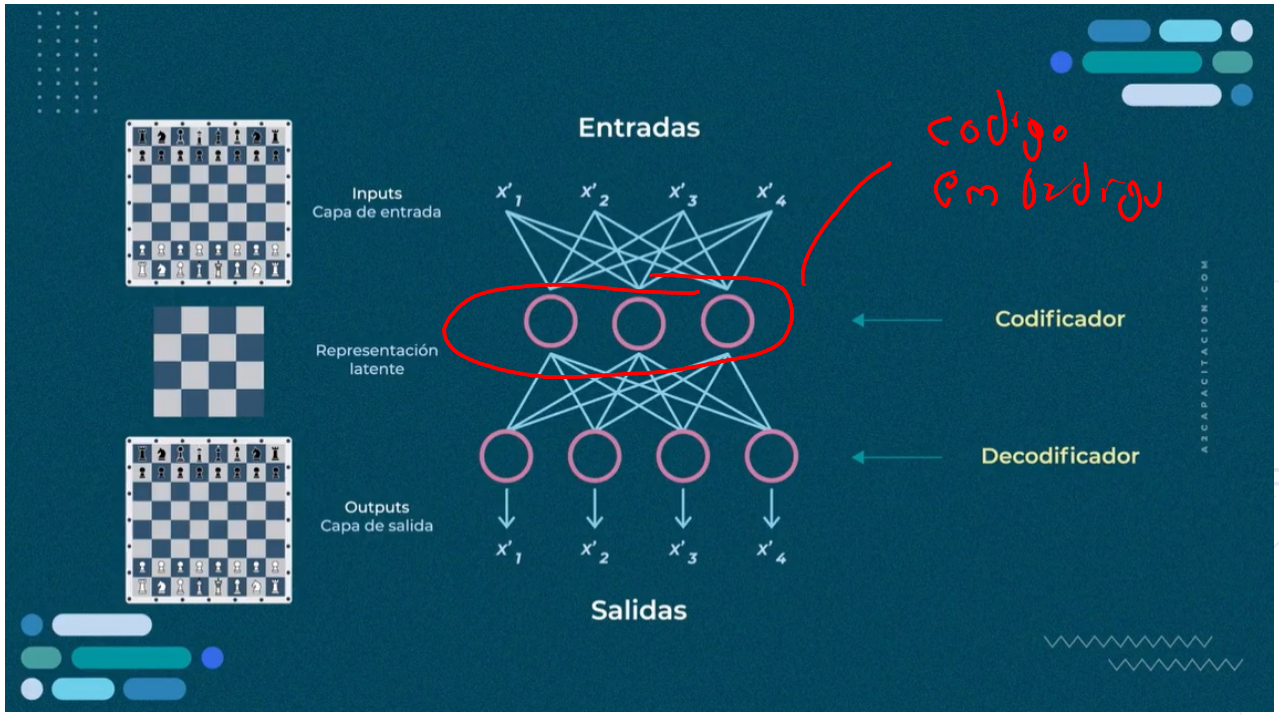
La relación entre patrones y memoria está muy estudiada en el ajedrez. Los jugadores expertos de ajedrez son capaces de memorizar fácilmente las posiciones de todas las piezas en un juego solo de ver el tablero unos segundos.



3.2 Introducción

Un autoencoder generalmente tiene la misma arquitectura que una red neuronal Densa, excepto que el número de neuronas en la capa de salida tiene que ser igual a la cantidad de entradas.

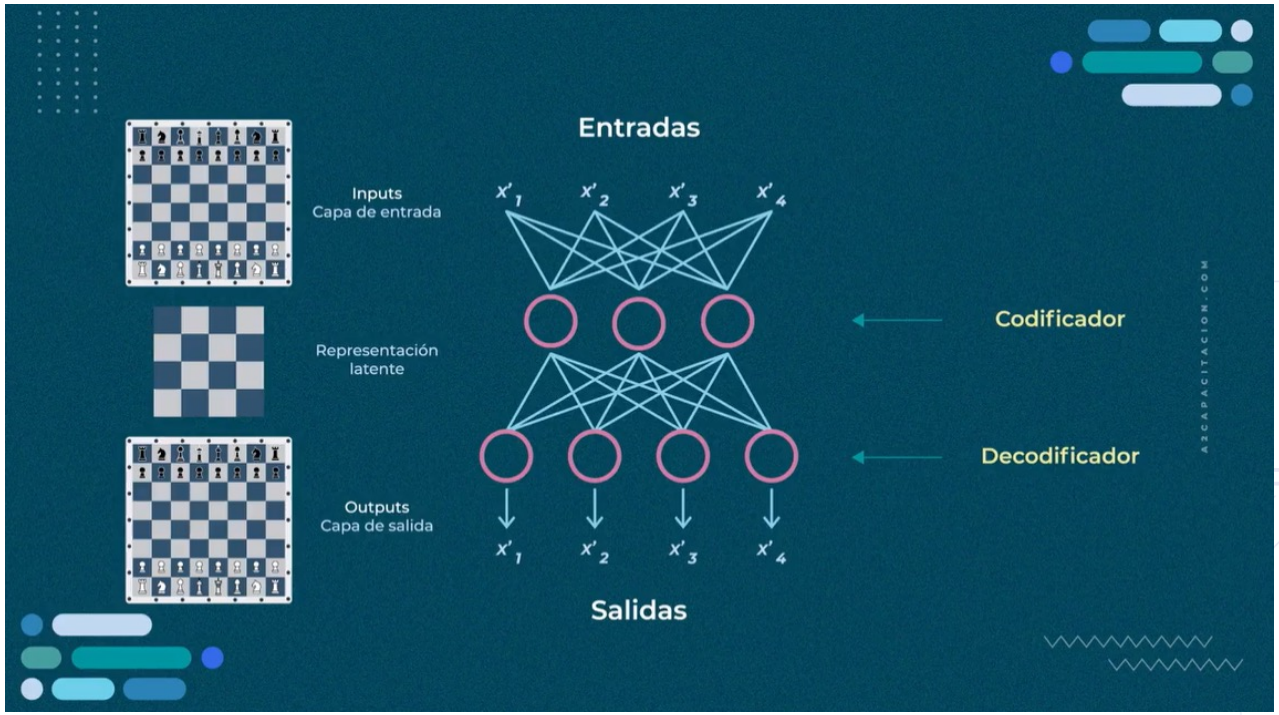
En este ejemplo hay 4 entradas, una capa oculta de 3 neuronas, que sería el encoder, y una capa de salida de 4 neuronas, el decoder. Las salidas se llaman reconstrucciones por que el autoencoder intenta reconstruir las entradas, y la función de costo contiene una pérdida de reconstrucción que penaliza el modelo cuando las reconstrucciones son diferentes de las entradas.



3.3 Introducción

El autoencoder se dice que es sub completo por que la representación interna es de menor dimensión que los datos de entrada.

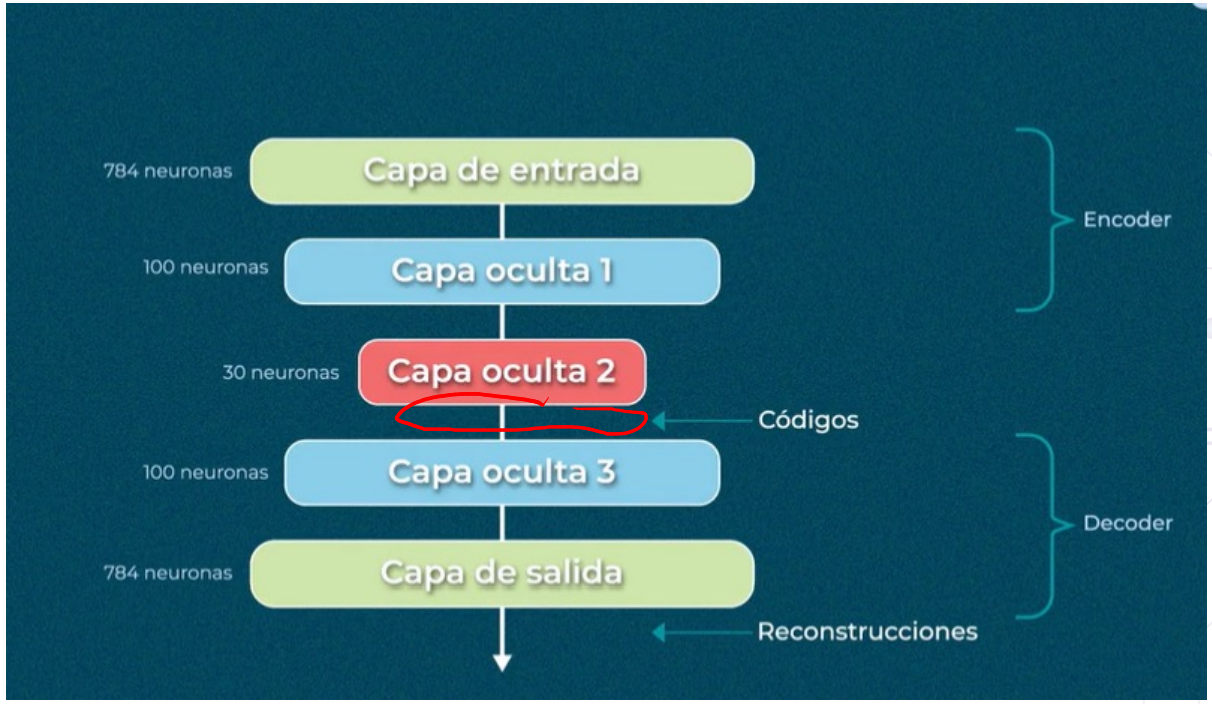
Un autoencoder sub completo no puede simplemente copiar las entradas y pegarlas como si fueran la salida y ya. Tiene que aprender los patrones de la entrada para poder reconstruirla y escupirte la salida como el autoencoder la entendió. Por lo mismo, se ve obligado a entender las características más importantes de la entrada.



3.4 Autoencoders Apilados

Igual que cualquier otra red neuronal que hemos discutido hasta ahora, los autoencoders pueden tener capas ocultas múltiples. Cuando esto ocurre, les llamamos stacked autoencoders o ENCODERS APILADOS (o también, encoders profundos).

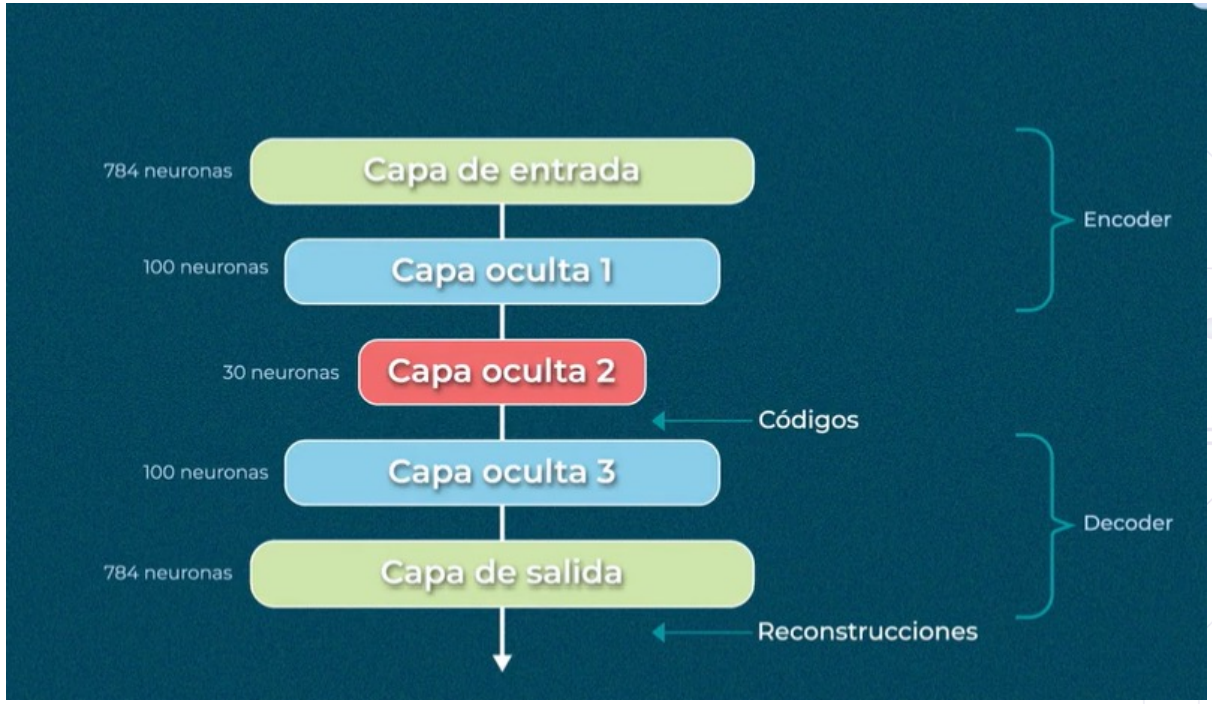
Agregarle más capas ayuda al autoencoder a aprender patrones más complejos. Dicho eso, recuerda que no estamos buscando hacer el autoencoder MUY poderoso. Imagina un encoder tan potente que simplemente mapea cada entrada a un número arbitrario, y el decoder aprende la inversa.



3.5 Autoencoders Apilados

Obviamente un autoencoder va a reconstruir los datos perfectamente, pero no habrá aprendido a hacer generalizaciones útiles en el proceso.

La arquitectura de un autoencoder apilado es típicamente simétrica con respecto a la capa central oculta (la capa de código). Para ponerlo de manera sencilla, se ve como un sándwich. Por ejemplo, el autoencoder para MNIST puede tener 784 entradas, seguido de una capa oculta de 100 neuronas, y luego una capa oculta de 30 neuronas y luego otra capa oculta de 100 neuronaes, y la salida con 784 neuronas.

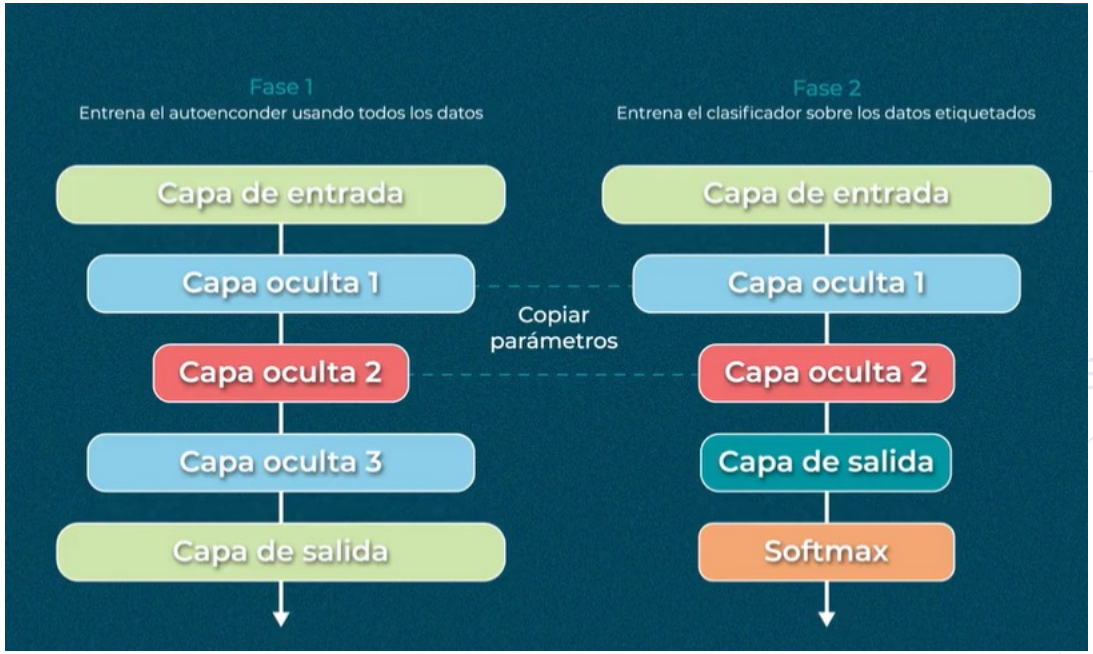


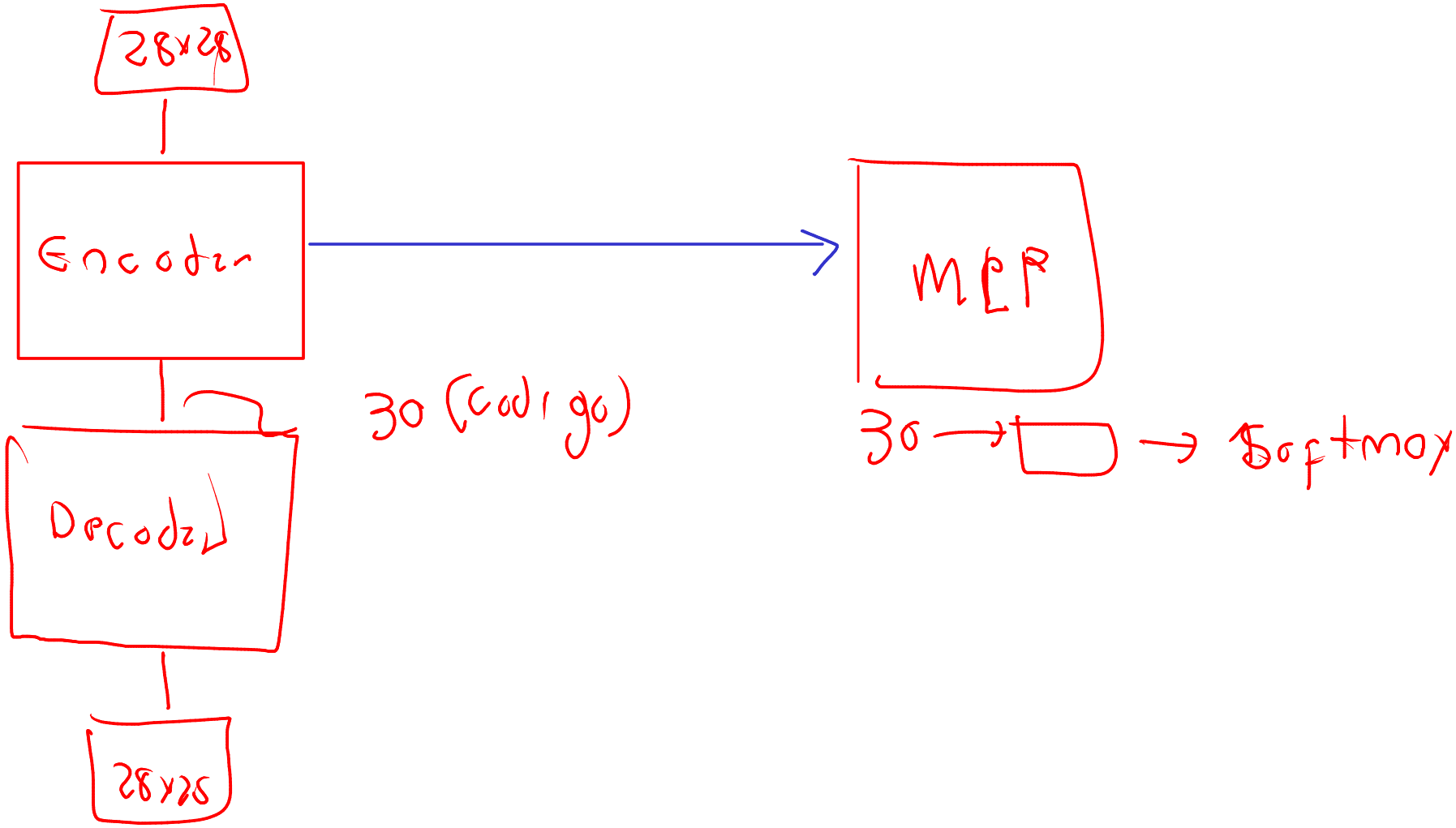
3.6 Preentrenamiento Sin Supervisión usando Autoencoders Encimados

Si estas resolviendo una tarea compleja supervisada pero no tienes un montón de datos de entrenamiento etiquetados, una solución es encontrar una red neuronal que realice una tarea similar y reúse sus capas inferiores.

Esto hace posible entrenar un modelo de alto desempeño usando pocos datos de entrenamiento por que tu red neuronal no tiene que aprender todas las variables de bajo nivel; simplemente va a reusar los detectores aprendidos por la red existente.

De manera similar, si tienes un dataset grande, pero todo es sin etiqueta, puedes primero entrenar un autoencoder apilado usando todos los datos, y luego reutilizar las capas inferiores para crear una red neuronal para tu tarea de verdad y entrenarla usando los datos etiquetados




28x28
Encoder
Decoder
28x28
30 (codigo)
MLP
30 → □ → Softmax

Tipos de encoder

1. Densos
Capaces de reconocer patrones en las imagenes a través del algoritmo MLP
Faciles de ajustar los hiperametros

2. Recurrentes
Capaces de reconocer secuencias dentro de las imagenes (especialmente util)
si las imagenes son una secuencia movimientos (Piensen ajedrez)
Dependen de los datos (que representen secuencias)

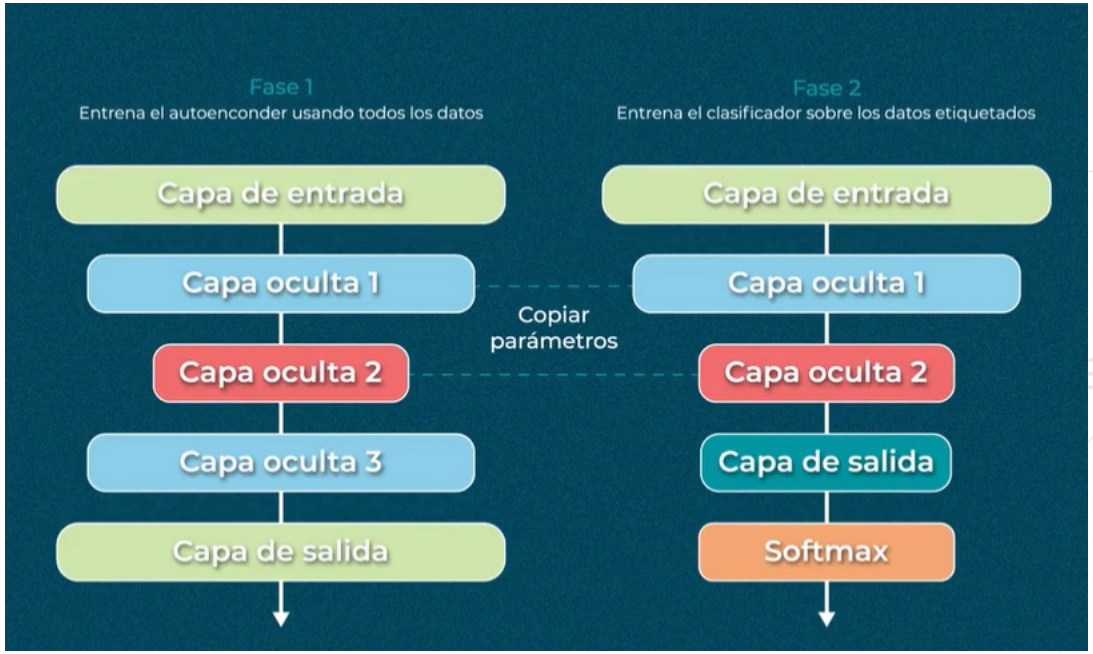
3. Convoluciones

Permiten extraer caracteristicas de las imagenes a través de las capas convolucionales y los paddings, son mas potentes para tareas de CLASIFICACION
Depende mucho de como se ajustan las capas convolucionales
Depende de los hiperparametros (aveces son mas dificiles ajustar)

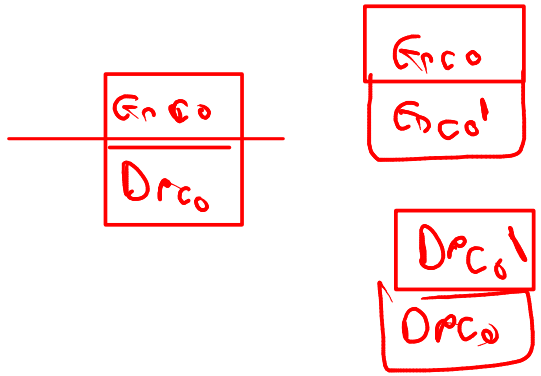
3.7 Preentrenamiento Sin Supervisión usando Autoencoders Encimados

Por ejemplo, puedes usar un autoencoder apilado para hacer preentrenamientos sin supervisión para una red neuronal de clasificación. Cuando entrenas un clasificador, si en realidad no tienes muchos datos etiquetados de entrenamiento, tal vez quieras congelar las capas pre entrenadas.

La implementación no tiene nada de chiste – simplemente entrena el autoencoder usando todos los datos de entrenamiento (etiquetados y sin etiquetar), y luego reúsa su capa de codificador para crear una nueva red neuronal.

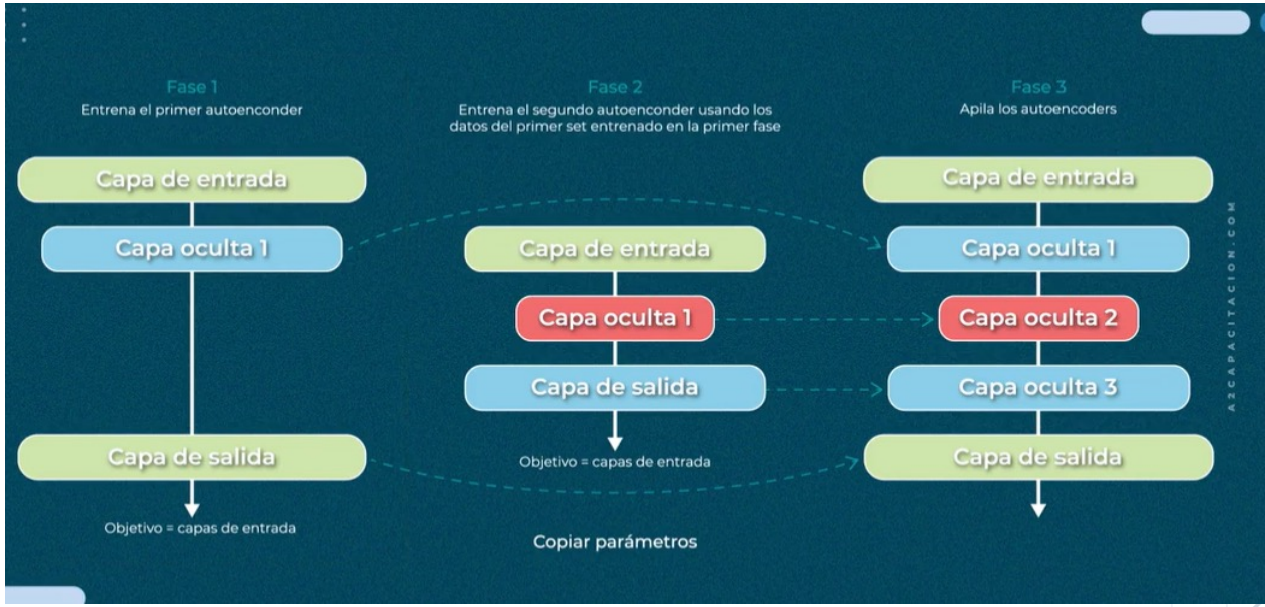


3.8 Un Autoencoder a la vez



En vez de entrenar todo el autoencoder apilado de un solo jalón como le acabamos de hacer, es posible entrenar un autoencoder estrecho a la vez, y luego apilarlos todos en un solo autoencoder apilado. Esta técnica no se usa mucho hoy en día, pero la vas a ver en papers referenciada como “greedy layerwise training”, así que vale la pena saber a qué se refieren.

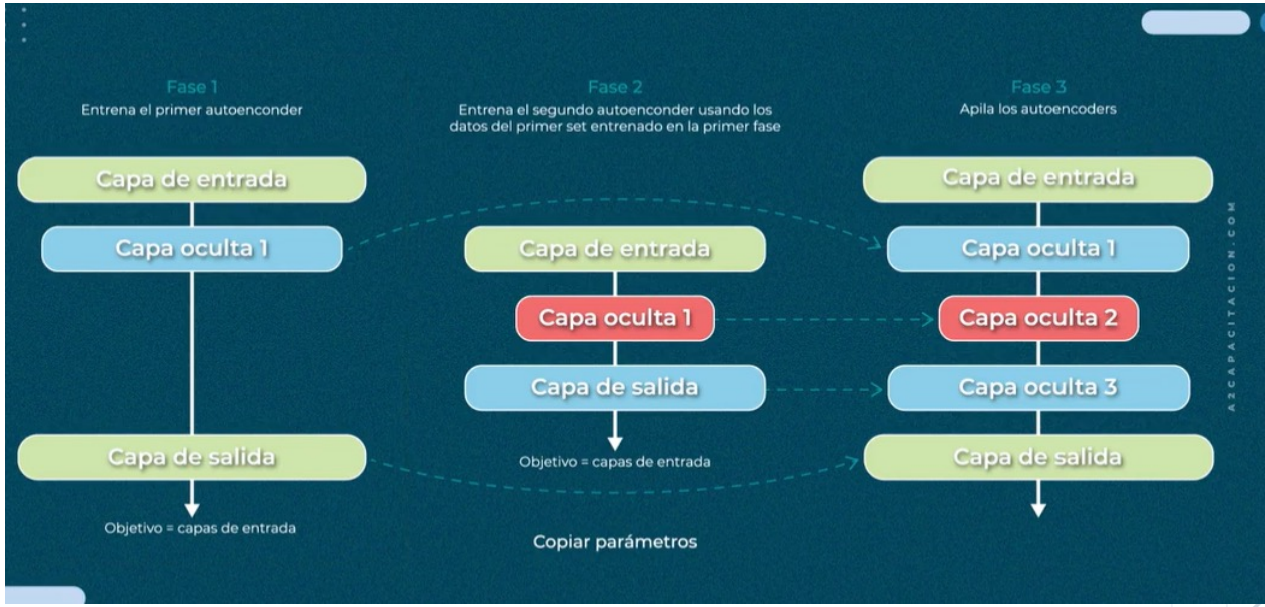
Durante la primera fase del entrenamiento, el primer autoencoder aprende a reconstruir las entradas. Luego codificamos todo el entrenamiento usando el primer autoencoder, y esto nos da un nuevo set de entrenamiento comprimido.



3.9 Un Autoencoder a la vez

Luego entrenamos un segundo autoencoder en este nuevo dataset. Esta es la segunda fase del entrenamiento. Finalmente, construimos un gran sándwich usando todos estos autoencoders.

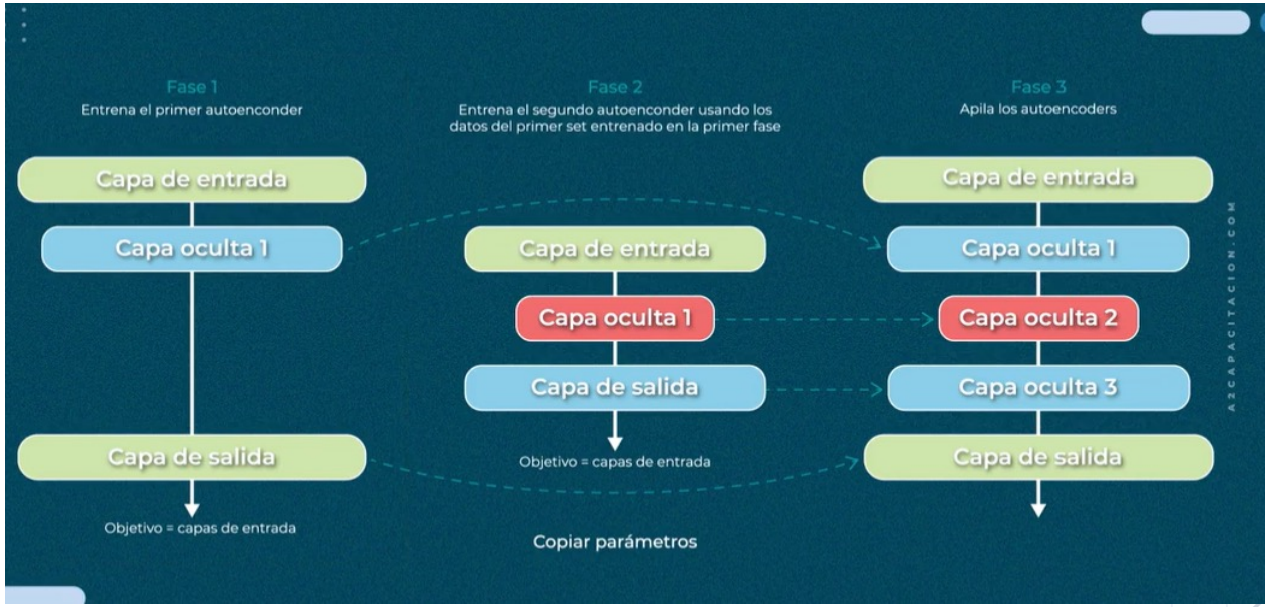
Esto nos da un autoencoder apilado final. Podrimos fácilmente entrenar más autoencoders de esta manera, construyendo un autoencoder apilado muy profundo.



3.10 Autoencoders Convolucionales

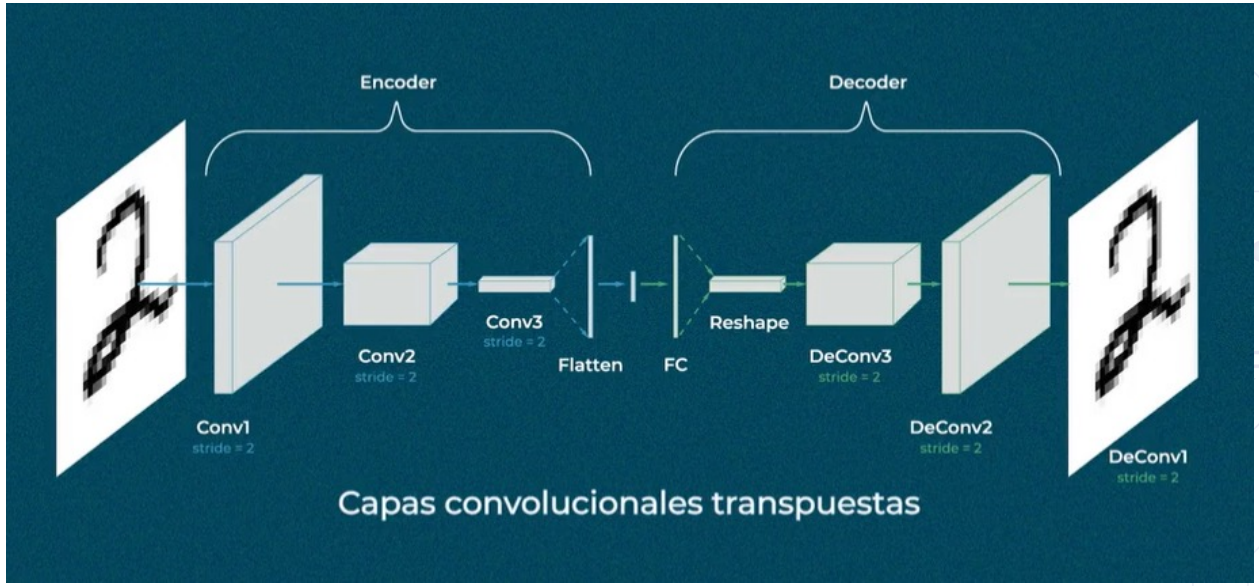
Luego entrenamos un segundo autoencoder en este nuevo dataset. Esta es la segunda fase del entrenamiento. Finalmente, construimos un gran sándwich usando todos estos autoencoders.

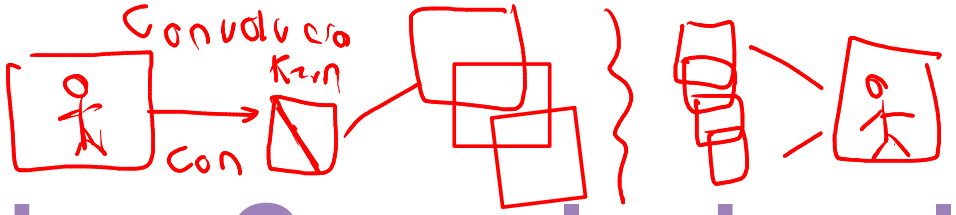
Esto nos da un autoencoder apilado final. Podrimos fácilmente entrenar más autoencoders de esta manera, construyendo un autoencoder apilado muy profundo.



3.11 Autoencoders Convolucionales

Si estas lidiando con imágenes, entonces los autoencoders que hemos visto hasta ahora no van a funcionar bien (salvo que sean imágenes muy sosas como las de MNIST). Como ya sabemos, la red por excelencia para lidiar con imágenes más complejas son las redes convolucionales.

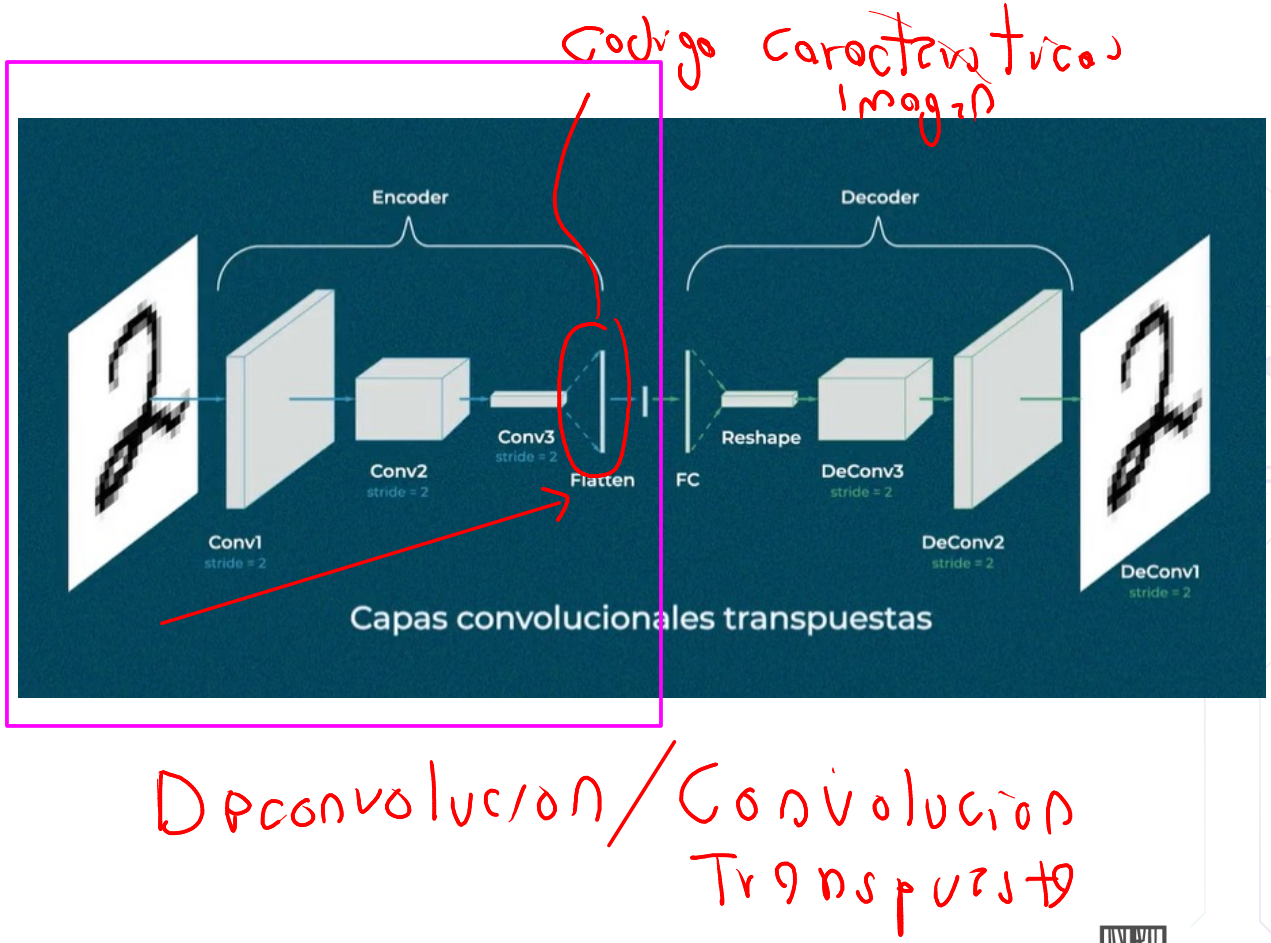




3.12 Autoencoders Convolucionales

Así que, si vas a construir un autoencoder para imágenes, pues simplemente vas a necesitar construir un autoencoder convolucional. El encoder es un CNN regular compuesto de varias capas convolucionales y capas de pooling.

Reduce la dimensionalidad espacial de las entradas, mientras incrementa la profundidad. El decoder debe de hacer lo inverso (aumentar la resolución de la imagen y reducir su profundidad de regreso a las dimensiones originales), y para esto puedes usar capas convolucionales transpuestas.



Preguntas