eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Grado en Ingeniería Informática

Computación

Trabajo de Fin de Grado

Exploración de Arquitecturas text-to-layout

Autor/a

Eneko Suarez Etxeberria

2022

eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Grado en Ingeniería Informática

Computación

Trabajo de Fin de Grado

Exploración de Arquitecturas text-to-layout

Autor/a

Eneko Suarez Etxeberria

Directore/a(s)

Gorka Azkune Galparsoro y Oier López de Lacalle Lecuona

Resumen

La tarea de generación de imágenes a partir de texto es excepcionalmente compleja. Es por ello que en algunos casos la tarea se divide en varias subtareas. Una de las posibles subtareas es la conocida como *text-to-layout*. Ésta sería la primera en la cadena y trata de la colocación de los diferentes objetos en la escena para crear la composición o *layout* de la imagen. El objetivo es facilitar la tarea de los siguientes pasos dándoles más información que el texto: posición y tamaño aproximado de los objetos principales de la escena.

En este proyecto nos centraremos en esta tarea: *text-to-layout*. Se tomarán las arquitecturas desarrolladas por Carlos Domínguez en su proyecto de fin de grado [Dominguez, 2021] y se actualizarán para hacer uso de los modelos más utilizados en los últimos años: *Transformers*.

Además se propone una nueva prueba para la evaluación de los modelos. Esta prueba es capaz de diferenciar la calidad entre diferentes modelos de manera igual o mejor que las métricas previamente propuestas. También es considerablemente más intuitiva.

Índice general

Resumen I

Índice general III

Índice de figuras VII

Índice de tablas XI

1. Introducción 1

2. Antecedentes 5

2.1. Arquitecturas seq2seq 5

2.2. Redes Recurrentes (RNN) 6

2.3. Redes Recurrentes para la Generación de Composición de Escena 7

2.3.1. Codificador 7

2.3.2. Decodificador 8

2.4. Transformers 8

2.4.1. Codificador Transformer (*Transformer Encoder*) 11

2.4.2. Decodificadores Transformer (*Transformer decoder*) 12

2.5. Pre-entrenamiento/*fine-tuning* 13

3. Soluciones Propuestas **15**

3.1. Formalización del Problema 15

3.1.1. Entrada 15

3.1.2. Salida 16

3.2. TRAN2LY 18

3.3. STRAN2LY 20

3.4. TRAN2TRAN 20

3.4.1. Codificador 20

3.4.2. Decodificador 20

4. Conjuntos de Datos Utilizados **23**

4.1. MSCOCO2014 23

4.2. Spatial Commonsense Dataset 24

4.2.1. Tamaño 24

4.2.2. Altura 26

4.2.3. Posición Relativa 27

4.2.4. Procesamiento de Salida 27

4.2.5. Pre-procesado 28

5. Experimentos y Resultados **31**

5.1. Entrenamiento 31

5.1.1. Pérdida 32

5.2. Métricas de Evaluación 34

5.3. Selección de Modelo 35

5.4. Nombramiento de Arquitecturas 36

5.5. Resultados en MSCOCO 36

5.5.1. Pérdidas en Entrenamiento y desarrollo 36

5.5.2. Resultados de Métricas en desarrollo . . . 39
5.5.3. Resultados en Test . . . 41
5.5.4. Ejemplos . . . 42
5.6. Resultados en Spatial CommonSense . . . 48
5.6.1. Ejemplos . . . 50

6. Conclusiones y Trabajo Futuro **59**
6.1. Conclusiones . . . 59
6.2. Trabajo Futuro . . . 60

Anexos

A. Seguimiento **65**
A.1. Descripción del Proyecto . . . 65
A.2. Objetivos . . . 66
A.3. Restricciones . . . 67
A.4. Plan de trabajo . . . 68
A.5. Metodología . . . 71
A.5.1. Entorno de trabajo . . . 71
A.5.2. Comunicación . . . 72
A.5.3. Horarios de trabajo y reuniones . . . 72
A.5.4. Provisión de potencia de cómputo . . . 73
A.6. Riesgos . . . 73

B. Optimizaciones **75**
B.1. Conversión de Coordenadas a *one-hot encoding* . . . 75
B.2. Redondeo de Coordenadas a la Rejilla de Salida . . . 76

C. Fine-Tuning Process **79**

C.1. TRAN2LY 79

C.1.1. *Pooling* 79

C.1.2. Temperatura 80

C.1.3. Tabla de parámetros 81

C.2. STRAN2LY 82

C.3. TRAN2TRAN 82

Bibliografía **85**

Índice de figuras

1.1. Ejemplo de la tarea de generación de imagen a partir de texto (*text to image*) con el paso intermedio de generación de composición (*text-to-layout*) 2

2.1. Esquema de la idea general de una arquitectura *Encoder-Decoder*. 6

2.2. Esquema de una RNN procesando una secuencia de N elementos 6

2.3. Visualización de la atención que le da el token "_it" al resto de tokens de la frase "The animal didn't cross the street because it was too tired". La imagen original es de [Alammar, 2020]. La visualización ha sido realizada utilizando la herramienta [Vig, 2021]. 9

2.4. Esquema del proceso del *self-attention* desde el punto de vista del token Perro con dos tokens en total en la secuencia. La imagen original es de [Alammar, 2020]. Ha sido adaptada al castellano y las palabras de los tokens cambiadas. 10

2.5. Visualización de la idea general del trabajo de un *transformer-encoder*. Cada vector inicial se ve representado por un color y se pueden ver los vectores finales como una "mezcla" entre el vector inicial y el resto de vectores. Por ejemplo, el vector de perro acaba siendo mezclado principalmente con el representante de blanco y juega. 11

2.6. Esquema del funcionamiento general de un Transformer-Decoder. Imagen tomada de [von Platen, 2020] 13

2.7. Esquema del funcionamiento general de un Transformer. Imagen tomada de [von Platen, 2020] 14

3.1. Ejemplo de composición de imagen con los diferentes datos de cada objeto marcados. El tipo (clase) de los objetos: *person* (persona), *dog* (perro), *hairdryier* (secador de pelo); posiciones (con las coordenadas normalizadas entre 0 y 1) y tamaños: alturas y anchuras (normalizadas entre 0 y 1). La descripción de la imagen se ha traducido al castellano para facilitar la comprensión. En un ejemplo real la descripción sería en inglés. 17

3.2. Ejemplo del proceso de *max-pooling* fusionando cuatro vectores 19

3.3. Esquema de la arquitectura de TRAN2LY 19

4.1. Tres ejemplos de posibles salida a la entrada ”Un pájaro y una silla”. Cada salida está marcada como correcta o incorrecta. 25

4.2. Tres ejemplos de posibles salida a la entrada ”Un sofá y un humano”. Cada salida está marcada como correcta o incorrecta. 27

5.1. Ejemplo de una mala puntuación IoU para una composición correcta. Los *bounding-box* del *ground-truth* en amarillo y los predichos en verde. La composición es idéntica al *ground-truth* y sigue siendo coherente con el texto pero no consigue buena puntuación por estar desplazada 34

5.2. Pérdidas de $RNN2LY_F$ en entrenamiento (azul) y en desarrollo (naranja) . 36

5.3. Pérdidas de $RNN2LY_{UF}$ en entrenamiento (azul) y en desarrollo (naranja) 37

5.4. Pérdidas de $TRAN2LY_F$ en entrenamiento (azul) y en desarrollo (naranja) 37

5.5. Pérdidas de $TRAN2LY_{UF}$ en entrenamiento (azul) y en desarrollo (naranja) 37

5.6. Pérdidas de $STRAN2LY_F$ en entrenamiento (azul) y en desarrollo (naranja) 37

5.7. Pérdidas de $STRAN2LY_{UF}$ en entrenamiento (azul) y en desarrollo (naranja) 38

5.8. Pérdidas de $TRAN2TRAN_F$ en entrenamiento (azul) y en desarrollo (naranja) 38

5.9. Pérdidas de $TRAN2TRAN_{UF}$ en entrenamiento (azul) y en desarrollo (naranja) 38

5.10. Puntuación (eq. 5.2) de las arquitecturas $RNN2LY_F$, $RNN2LY_{UF}$, $TRAN2LY_F$, $TRAN2LY_{UF}$, $STRAN2LY_F$, $STRAN2LY_{UF}$, $TRAN2TRAN_F$ y $TRAN2TRAN_{UF}$ durante sus respectivas 100 épocas de entrenamiento. 39

5.11. Mujer y niño en una cocina blanca. ID de imagen: 141017 43

5.12. Un jugador de béisbol bateando la peloto con otros dos detrás de él. . . . 44

5.13. Foto de un retrovisor que muestra un perro jadeando fuera de la ventana de un coche. 45

5.14. un hombre montando un *kiteboard* sobre el océano bajo un cielo nublado 46

5.15. Un chico y una chica sentados en una mesa de madera con el chico mirando a la chica. 47

5.16. Un rebaño de ovejas pastando en el campo. 48

5.17. Un vaso y un microondas. 51

5.18. Un perro y una botella. 52

5.19. Un pájaro y un avión. 52

5.20. Un pájaro y una silla. 53

5.21. Un caballo y un camión. 54

5.22. ”Un pájaro y una jirafa” (izquierda) y ”Una botella y un camión” (derecha). 54

5.23. ”Una botella y una silla” (izquierda) y ”Una botella y un sofá” (derecha). 55

5.24. ”Un hombre alimentando el caballo” (izquierda) y ”Una mujer monta el caballo” (derecha). 56

5.25. ”Un hombre limpia el coche” (izquierda) y ”Una mujer conduce el coche” (derecha). 56

5.26. ”Un hombre reparando una bicicleta” (izquierda) y ”Un hombre montando una bicicleta” (derecha). 57

5.27. ”Una chica chuta el balón de fútbol.” (izquierda) y ”Un chico está chutando el balón de fútbol.” (derecha). ”Un hombre la da un cabezazo al balón.” (debajo) 58

6.1. Dos ejemplos de *ground-truth* y resultado. Ambos resultados se evaluan con dos métodos: uno categórico (izq.) y uno diferenciable (der.) que se podría utilizar como pérdida. 61

A.1. Paquetes de trabajo identificados durante el trabajo. En verde los paquetes que estaban planeados desde un inicio y se han realizados. En rojo los paquetes planeados inicialmente que no se han realizado. En azul los paquetes que no estaban planificados pero se han realizado. 68

A.2. Fechas en las que se planeó realizar cada paquete de trabajo. 70

A.3. Fechas en las que se han realizado los diferentes paquetes de trabajo. En verde las fechas que coinciden con el plan inicial y en rojo las que no. . . 71

C.1. Ejemplo del proceso de *max-pooling* fusionando cuatro vectores 80

C.2. Ejemplo del proceso de *average-pooling* fusionando cuatro vectores . . . 81

Índice de tablas

4.1. Todos los niveles de tamaño con sus objetos. 25

4.2. Todos los niveles de altura con sus objetos. 26

5.1. Mejor puntuación conseguida por cada arquitectura. *Época* indica la época más tardía en la que ha conseguido la mejor puntuación. *Rango* $\pm$ *0.03* indica el rango de épocas mínimo que incluye todas las épocas que han conseguido un resultado que está a menos de 0.03 puntos de distancia del mejor. 40

5.2. Resultados de cada arquitectura de las diferentes métricas explicadas en 5.2 en el subconjunto de test de MSCOCO. Se añade también como referencia los resultados que consiguió [Dominguez, 2021] y [Li et al., 2019]. Las arquitecturas de [Dominguez, 2021] son $RNN2LY_{PT}$: *RNN2LY* con codificador pre-entrenado, $RNN2LY_{FT}$: *RNN2LY* con codificador pre-entreando y *fine-tuneado* y *GNC2LY*: una arquitectura diferente que utiliza un codificador GCNN 41

5.3. Precisión de las diferentes arquitecturas en cada subconjunto de *Spatial Commonsense Test*. Los resultados de los modelos Best PLM, VinVL e ISM son los resultados tomados de [Liu et al., 2022]. Son modelos de generación de imágenes así que en su caso, hay que conseguir el resultado y detectar después los objetos en la imagen. Los resultados mostrados son con humanos detectando los objetos y precisión sobre el conjunto de datos completo. 49

A.1. Horas planeadas y horas utilizadas en cada paquete de trabajo. 71

C.1. Parámetros utilizados en la versión final de TRAN2LY 82

C.2. Parámetros utilizados en la versión final de STRAN2LY 82

C.3. Parámetros probados para TRAN2TRAN 83

C.4. Parámetros utilizados en la versión final de TRAN2TRAN 84

1. CAPÍTULO

Introducción

La generación de imágenes a partir texto es una tarea excepcionalmente compleja. Requiere de una variedad de habilidades propias de diferentes modalidades para el correcto cumplimiento de la tarea. Entre estas habilidades se encuentra el entendimiento de las formas de diferentes objetos, la relación de los objetos entre sí o la habilidad para imaginar diferentes detalles de la imagen a partir de información implícita. Además, el modelo tiene que poner en uso estas habilidades siguiendo las indicaciones del texto, haciendo que el modelo necesite también habilidades de procesamiento de lenguaje natural.

Es por esta complejidad que muchas de las arquitecturas que se encargan de generación de imágenes a partir de texto reparten la tarea en varias subtareas. El objetivo de esta división es el desacoplamiento de las diferentes habilidades necesarias para simplificar su realización. La subtarea de *text-to-layout* (fig. 1.1) es la más comúnmente desacoplada del resto del proceso. La tarea de *text-to-layout* consiste en generar la composición básica de la imagen a partir del texto. La arquitectura analiza el texto introducido y se encarga de colocar los diferentes objetos de la escena, darles un tamaño aproximado y especificar qué tipo de objeto es. Para ser más precisos, por cada objeto predicho en la escena se genera una *bounding-box* y una clase de objeto. La *bounding-box* contiene la información no solo del tamaño (altura y anchura) del objeto sino también de su posición en la imagen.

Diferentes arquitecturas habían conseguido resultados avanzados en la generación de imágenes para escenas simples con un solo objeto o pocos objetos [Reed et al., 2016] [Zhang et al., 2017] [Zhu et al., 2019]. Fue al intentar dar el salto a escenas más complejas que arquitecturas más actuales empezaron a hacer uso de *text-to-layout* [Hong et al., 2018]

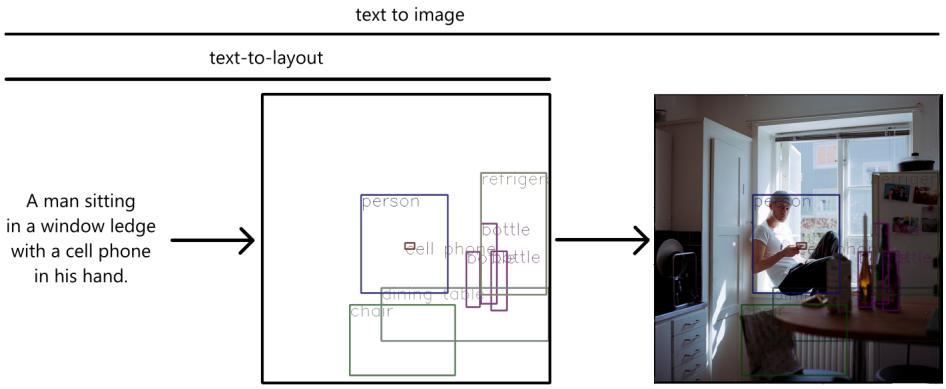

Figura 1.1: Ejemplo de la tarea de generación de imagen a partir de texto (*text to image*) con el paso intermedio de generación de composición (*text-to-layout*)

[Johnson et al., 2018]. Quizá la más conocida es ObjGAN [Li et al., 2019]. Desde entonces otras arquitecturas [Zakraoui et al., 2021] [Liang et al., 2022], especializadas en escenas excepcionalmente complejas siguen utilizando la técnica.

La separación del paso *text-to-layout* del resto del proceso consigue desacoplar habilidades de procesamiento de lenguaje natural y el entendimiento básico de las relaciones posicionales y de tamaños de los objetos. Además, este paso se encargará de darle dirección general a la imagen. Es el paso que toma las decisiones de composición de imagen. Siendo el primer paso, *text-to-layout* forma la base sobre la que el resto de subtareas construirán.

Dado que recae sobre él la ejecución de diferentes habilidades además de ser la base para el resto, *text-to-layout* es un paso crucial del proceso. Es por ello que la optimización del modelo que realice esta tarea es de gran importancia en las arquitecturas de generación de imagen a partir de texto.

En 2021 Carlos Domínguez publicó su trabajo de fin de grado [Dominguez, 2021] donde desarrolló una arquitectura para *text-to-layout*. Sus arquitecturas superaban a las utilizadas en el proceso de generación de imágenes de arquitecturas del estado del arte. En este proyecto hemos explorado la posibilidad de editar sus arquitecturas para que utilicen modelos *transformer* [Vaswani et al., 2017] y conseguir resultados superiores a los suyos.

Además de las evaluaciones en la tarea de *text-to-layout*, también hemos añadido un test diferente que permite medir el conocimiento espacial básico que han conseguido las arquitecturas. Evalúa por ejemplo que las arquitecturas sepan que un tipo de objeto es más grande que otro (cama >libro) o qué posición relativa deberían de tener bajo diferentes

acciones. Es un test que evalúa habilidades generales manteniéndose sencillo e intuitivo sin ser demasiado fácil para las arquitecturas.

Todo el código utilizado está subido a este repositorio: kek

2. CAPÍTULO

Antecedentes

En este apartado se explicarán diferentes conceptos relacionados con el aprendizaje profundo (en inglés *deep learning*) necesarios para la correcta comprensión del resto del documento.

2.1. Arquitecturas seq2seq

Se les llama *seq2seq* a las arquitecturas capaces de procesar secuencias de longitud arbitraria como entrada y su salida es otra secuencia de longitud arbitraria. Las longitudes de las dos secuencias no tienen por qué coincidir.

Esta cualidad se consigue con arquitecturas de tipo *Encoder-Decoder* (fig. 2.1). Éstas tienen dos partes: el codificador, que se especializa en procesar la entrada y (generalmente) representarla en un vector de tamaño fijo y el decodificador que, tomando la representación dada, genera una nueva secuencia a partir de ella.

Este desacoplamiento de tareas es el que permite de forma sencilla que la longitud (y a veces el tipo) de la secuencia de salida sea diferente a la de entrada. Sin esta separación uno tendría que decidir cómo relacionar cada elemento de la entrada a cada uno de la salida. También tendría que navegar alrededor de las limitaciones de la mayor parte de los modelos: solo podrían relacionar un número fijo de elementos de la entrada con uno fijo de la salida.

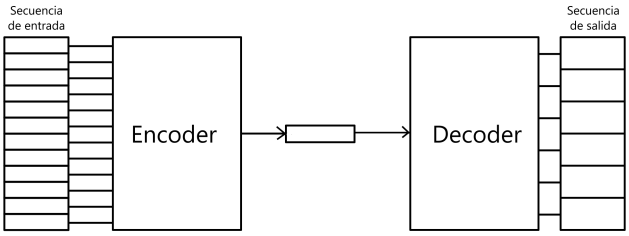

Figura 2.1: Esquema de la idea general de una arquitectura *Encoder-Decoder*.

2.2. Redes Recurrentes (RNN)

Las redes neuronales recurrentes (RNN por sus iniciales en inglés) (fig. 2.2) son redes neuronales hechas para procesar secuencias de datos. Procesan cada dato de la secuencia de uno en uno. Por cada dato procesado, produce dos resultados: un *output-state* y un *hidden-state*.

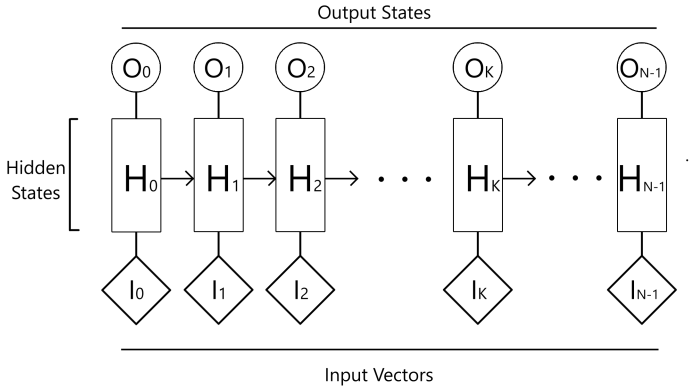

Figura 2.2: Esquema de una RNN procesando una secuencia de N elementos

El *output-state* es la salida de la red en ese paso. En el caso de ser utilizado como codificador (procesado de secuencia), podría ser una versión mejorada del vector de entrada. En caso de ser utilizado como decodificador (generación de secuencia), podría ser lo que el modelo predice como el siguiente dato.

El *hidden-state* es un vector que mantiene la información de la secuencia hasta el momento. Este vector se va transmitiendo de una iteración a otra. De esta manera, el quinto paso recibiría el *hidden-state* del cuarto paso. Este quinto paso añadiría al *hidden-state* información sobre el vector que está procesado en ese momento. El *hidden-state* dado por el quinto paso se le daría al sexto. Esta cadena seguiría hasta el final de la secuencia.

Así cada paso tiene como contexto la información acumulada sobre la secuencia hasta el momento.

Al final de la secuencia se puede tomar el último *hidden-state* como el resumen de la secuencia entera. Si se está utilizando la RNN como codificador, en vez de tomar el grupo de *output-state*s como la salida de la red, se puede tomar solo el *hidden-state* para tener un solo vector que represente la secuencia entera.

Además de las RNN, también existen las LSTM (*Long-Short Term Memory*) que funcionan de manera similar. Las LSTM son generalmente consideradas una versión superior (y más compleja) de las RNN. Por otro lado también existen métodos para hacer que la acumulación de información en el *hidden-state* no se realice solo de principio a fin sino también desde el final al principio. A modelos que utilizan este método se les llama bi-direccionales (bi-RNN o bi-LSTM). En este documento no se profundizará sobre estos temas ya que solo es necesitará el conocimiento de la existencia de los *hidden-state* y *output-state* para la comprensión de las arquitecturas explicadas.

2.3. Redes Recurrentes para la Generación de Composición de Escena

En este apartado se explicará el funcionamiento de la arquitectura *RNN2LY* desarrollada por [Dominguez, 2021]. Es una arquitectura desarrollada para la realización de la tarea *text-to-layout* y es la arquitectura tomada como referencia y de la que derivan las propuestas de este proyecto. Tanto el codificador como el decodificador son LSTMs en esta arquitectura.

2.3.1. Codificador

El codificador primero convierte cada palabra de la frase en un token utilizando un tokenizador sencillo con una entrada por cada palabra presente en el conjunto de datos MSCOCO. Después de que el LSTM codificador procese todos los tokens de entrada, se toma su último *hidden-state* como salida. De esta manera se consigue como salida de la secuencia entera un único vector de tamaño fijo.

2.3.2. Decodificador

El decodificador, para conseguir utilizar la información del codificador, toma el el vector de salida del codificador y lo utiliza como *hidden-state* inicial en su primer paso.

Para conseguir generar un nuevo objeto para la secuencia de salida, se toman los siguientes pasos:

1. Se convierte la información del anterior objeto (clase, posición y tamaño) en un vector numérico. Esto se consigue pasando cada una de las tres partes de la información del objeto por una red densa que lo transforma en un vector. Después estos tres vectores se concatenan en uno solo.
2. Se introduce el vector conseguido en el LSTM (junto al último *hidden-state*). Consiguiendo un nuevo *hidden-state* y un *output-state*.
3. Se toma el *output-state* del LSTM y se introduce en diferentes redes densas para conseguir la nueva clase, la nueva posición y el nuevo tamaño para generar el nuevo objeto. Primero se consigue la clase pasando el *output-state* por una capa densa. Después se consigue la posición. La posición depende tanto de la clase como del *output-state*. Por último, el tamaño depende tanto de la clase, como de la posición como del *output-state*.

Como se puede observar, para conseguir un nuevo objeto hace falta partir de la información del anterior. Para conseguir el primer objeto se introduce un objeto especial: el objeto de inicio de secuencia (SOS por sus iniciales en inglés). En nuestro caso, este SOS tiene la clase 1, la posición (0,0) y el tamaño también a (0,0).

La secuencia se da por terminada cuando se llega al límite impuesto de objetos o el modelo genera el objeto de final de secuencia (EOS por sus iniciales en inglés). En nuestro caso, este EOS tiene la clase 2, la posición (0,0) y el tamaño también a (0,0).

Para más detalles en la forma que tiene la salida de la clase, posición y tamaño al salir del decodificador, leer la sección 3.1.

2.4. Transformers

Los transformers son modelos introducidos por primera vez por Google en el artículo [Vaswani et al., 2017]. Como el título "*Attention is all you need*" indica, el modelo utiliza

principalmente mecanismos de atención.

Los mecanismos de atención son mecanismos que intentan dar más valor (atención) a partes de datos con mayor importancia y menos a las partes menos útiles. Mecanismos de atención propia o *self-attention* son mecanismos que calculan la importancia de los datos desde el punto de vista de cada dato. En la figura 2.3 se ve cómo el token ”_it” le da más atención (importancia) a los tokens ”The_” (el) y ”animal_” (animal). Este ejemplo está tomado a partir de atenciones calculadas por un modelo ya entrenado. Es por ello que se pueden encontrar relaciones con sentido como la del ejemplo en la figura 2.3, donde el pronombre ”it_”, que se refiere al animal, le está prestando atención a los tokens que conforman ”el animal” (The_ Animal_).

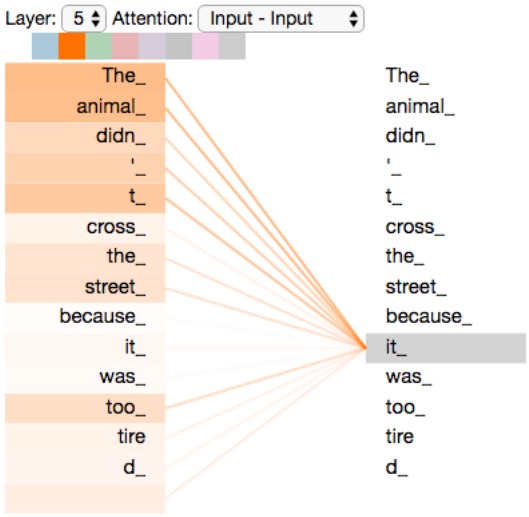

Figura 2.3: Visualización de la atención que le da el token ”_it” al resto de tokens de la frase ”The animal didn’t cross the street because it was too tired”. La imagen original es de [Alammar, 2020]. La visualización ha sido realizada utilizando la herramienta [Vig, 2021].

Cada token, además de calcular la importancia que le da a otros tokens, será transformado utilizando esta información. Cada token se va a ”mezclar” con el resto. Esta mezcla tendrá en cuenta la atención calculada y le dará más peso a los tokens que hayan conseguido mayor atención. De esta manera, en el ejemplo de la figura 2.3, el token ”it_” pasará a ser una mezcla entre sí mismo y el resto de tokens. Entre el resto de tokens, los que más influencia van a tener son los tokens ”The_” y ”animal_”.

Tanto el cálculo de la atención como la ”mezcla” de vectores empieza con el cálculo de tres vectores por cada token: *Key*, *Query* y *Value*. Para conseguir el valor de atención que le da el token A al token B, se compara el vector *Query* del token A con el vector *Key*

del token B. Esta comparación se realiza calculando el *dot product* de los vectores. Para realizar la mezcla de cada token, primero se normalizan las atenciones que les ha dado a todos los tokens de tal manera que la suma de las atenciones normalizadas resulte en 1. Después, el token pasa a ser la suma ponderada por las atenciones normalizadas de los vectores *Value* de cada token.

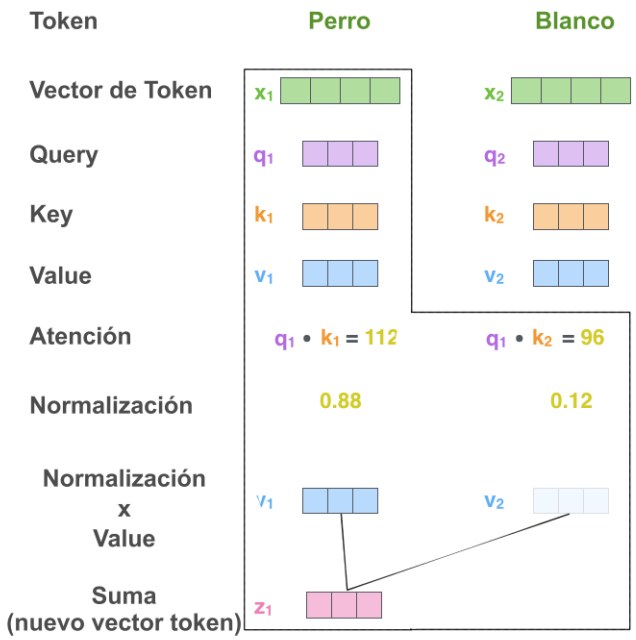

Figura 2.4: Esquema del proceso del *self-attention* desde el punto de vista del token Perro con dos tokens en total en la secuencia. La imagen original es de [Alammar, 2020]. Ha sido adaptada al castellano y las palabras de los tokens cambiadas.

En la figura 2.4 se puede ver un ejemplo con tan solo dos tokens. La figura es un ejemplo del proceso de *self-attention* del token ”Perro”. En este caso el token perro le da 112 de atención a sí mismo y 96 al token Blanco. Después de terminar la normalización de los vectores el token *Value* de Perro acabará constituyendo un 88 % del resultado final del siguiente vector de Perro.

Como se ve, todo el proceso depende del cálculo de los vectores *Query*, *Key* y *Value*. Esto es precisamente lo que los modelos *transformer* tienen que aprender. Sus parámetros son los diferentes números que conforman las matrices de transformación que transforman los vectores iniciales a vectores *Query*, *Key* y *Value*.

Los modelos tienen varias capas que realizan este proceso una detrás de la otra. En cada capa además se realiza este proceso múltiples veces. Una por cada cabeza que tenga la

capa. Los resultados de las cabezas se juntan en un solo resultado para conseguir un solo resultado por capa.

2.4.1. Codificador Transformer (*Transformer Encoder*)

Como se ha dicho en el apartado de *seq2seq*, el trabajo del codificador es procesar la entrada y comunicarle la información conseguida al decodificador. En este caso, la información no se va a resumir a un vector de tamaño fijo sino que se conseguirá una mejor representación de cada token entrante.

Cada vector resultante no solo representará el significado de su token, sino ese token dentro de la secuencia. El resultado se dice que está contextualizado. Por ejemplo: En la frase ”El perro blanco juega en el jardín” el vector inicial del token perro simplemente alude al tipo de animal, pero una vez contextualizado no debería referirse a cualquier perro, sino a uno blanco que está jugando (fig. 2.5).

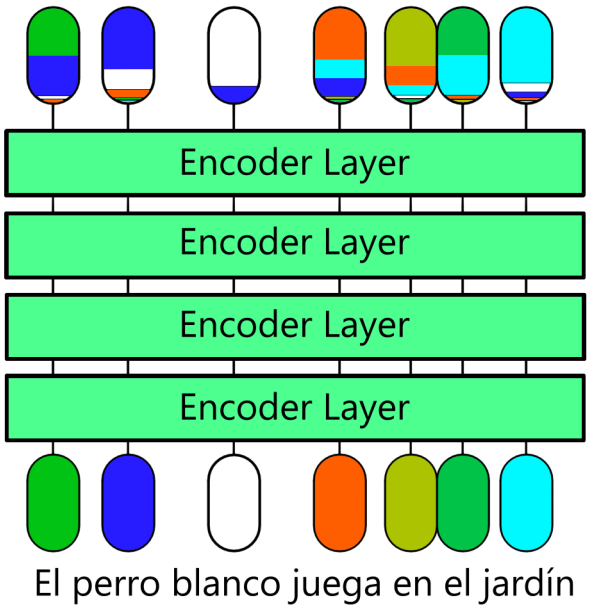

Figura 2.5: Visualización de la idea general del trabajo de un *transformer-encoder*. Cada vector inicial se ve representado por un color y se pueden ver los vectores finales como una ”mezcla” entre el vector inicial y el resto de vectores. Por ejemplo, el vector de perro acaba siendo mezclado principalmente con el representante de blanco y juega.

Durante el entrenamiento, el modelo aprende a cómo mezclar los vectores de manera correcta para conseguir vectores contextualizados con sentido. En modelos de procesado

del lenguaje natural, el modelo tendría que aprender a juntar los sustantivos con sus respectivos adjetivos (y solo los suyos), los pronombres con sus respectivos sustantivos o los verbos con sus sujetos, entre otros. Hay que denotar que, aunque decimos que **tendría que** aprender estas relaciones, en realidad aprenden de manera automática y solo llegarán a aprender estas relaciones si le son útiles en su tarea. En arquitecturas como BERT se ha encontrado que son capaces de codificar relaciones gramáticas y semánticas [Rogers et al., 2020].

2.4.2. Decodificadores Transformer (*Transformer decoder*)

El decodificador tendrá que tomar estos vectores contextualizados y generar la secuencia de salida. Como ya se ha dicho en el apartado de Codificadores Transformer, la entrada del decodificador no es un solo vector por secuencia sino un vector por token. El funcionamiento sigue siendo de *transformer*: mezcla los vectores de entrada para generar nuevos. Hay tres diferencias principales entre el decodificador y el codificador:

1. El objetivo es completamente diferente. En vez de conseguir un vector que sea mejor representante del token de entrada, lo que busca es conseguir un vector que represente el siguiente token de la secuencia. Es decir, en una frase, en vez de buscar representar mejor la palabra, busca conseguir un vector que represente **la siguiente** palabra de la frase. Así, el resultado es la predicción que hace el modelo para el siguiente token. Para conseguir el siguiente token a este, simplemente introduciríamos el token recién predicho al modelo.

2. Vectores memoria. Los vectores memoria son los vectores contextualizados introducidos por el codificador. El decodificador los utiliza en cada capa y puede mezclar los vectores con los que trabaja con vectores memoria. El decodificador los considera como vectores previos en la secuencia para la generación de los nuevos vectores. El decodificador no modifica los vectores memoria.

3. No todos los vectores se mezclan entre sí. Cada vector solo se mezcla con los anteriores de la secuencia y los vectores memoria. Esto se debe a que en la generación de, por ejemplo, el tercer token; solo se puede conseguir con la mezcla de los dos primeros tokens y los tokens de memoria

   En las siguientes iteraciones, aunque se introduzcan más vectores, para mantener la consistencia, cada vector se mezcla solo con los anteriores. Es decir, el tercer vector

de salida será el mismo tanto en la tercera iteración cuando se generó y solo estaban los dos anteriores y los de memoria que en la quinta iteración cuando hay ya cuatro vectores generados y los de memoria.

Quizá la manera más sencilla de comprender la arquitectura sea a través del esquema en la figura 2.6. Se ve claramente el método de introducción de los vectores memoria (en verde) en cada capa y la conexión de cada vector solo con los vectores anteriores.

El esquema de la figura 2.7 enseña el funcionamiento general del transformer-decoder trabajando con el transformer-encoder. También se ve mejor la introducción recurrente del último token predicho de vuelta en el decoder.

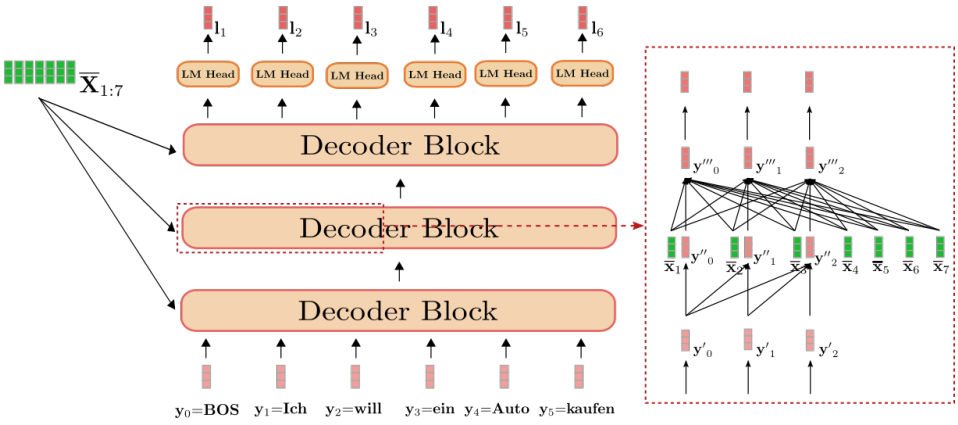

Figura 2.6: Esquema del funcionamiento general de un Transformer-Decoder. Imagen tomada de [von Platen, 2020]

2.5. Pre-entrenamiento/*fine-tuning*

Aunque la idea de pre-entrenar los modelos ya se demostró útil antes de su uso en el procesamiento del lenguaje natural, ha sido estos últimos años con los enormes modelos del lenguaje que ha tomado un rol central en el mundo del *Deep Learning*.

Para la realización de cualquier tarea los modelos tienen que desarrollar diferentes habilidades. Por ejemplo, para la tarea de análisis de sentimientos el modelo primero tendrá que familiarizarse con el inglés (o el idioma en el que estén escritos los mensajes) y después, una vez sabe procesar el contenido, a juzgar los mensajes por su sentimiento.

La idea de pre-entrenar el modelo es conseguir un modelo que tenga una habilidad base genérica. Así al aprender a realizar otras tareas más específicas partirá de una base y tendrá

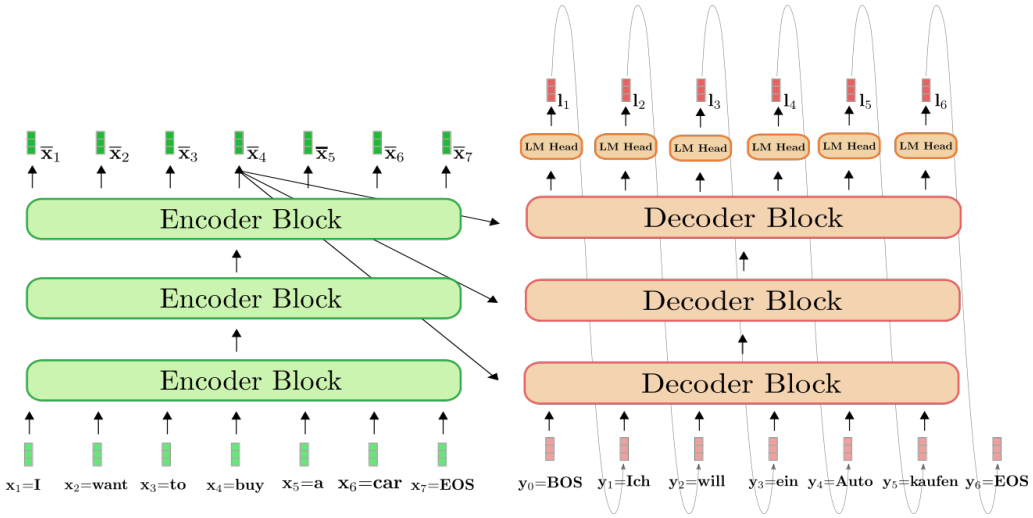

Figura 2.7: Esquema del funcionamiento general de un Transformer. Imagen tomada de [von Platen, 2020]

más facilidad. Al entrenamiento en las tareas genéricas se le llama pre-entrenamiento y en las específicas *fine-tuning*. En el ejemplo anterior, entrenar un modelo para que se acostumbre al inglés sería el pre-entrenamiento y aprender a juzgar el sentimiento de los mensajes el *fine-tuning*. De esta manera, al partir de un modelo ya reconoce diferentes patrones del inglés, será más fácil que aprenda a juzgar el sentimiento del mensaje.

En los modelos de procesamiento del lenguaje modernos se han utilizado técnicas de pre-entremiento que permiten la generación de ejemplos de forma automática y prácticamente infinitas. Este ha sido uno de los avances que ha propulsado a modelos modernos como GPT-3 [Brown et al., 2020] o LaMDA [Thoppilan et al., 2022] al nivel de entendimiento del lenguaje que poseen. Durante el proyecto se hará uso de diferentes modelos pre-entrenados como BERT [Devlin et al., 2018].

3. CAPÍTULO

Soluciones Propuestas

En este apartado se explicarán las arquitecturas desarrolladas y su funcionamiento.

3.1. Formalización del Problema

El primer paso en la resolución de cualquier problema es su formalización. En este apartado se explicarán las decisiones tomadas sobre la forma tanto de la entrada como de la salida del problema a resolver.

3.1.1. Entrada

La entrada es una descripción de imagen corta en inglés. Esta descripción entra en una sola frase sencilla y describe la escena general de la imagen. No describe cada objeto de la imagen. Como máximo describe los objetos principales y las acciones que están tomando.

El dato dado a la arquitectura es el texto plano representando la frase. Cada arquitectura utiliza un tokenizador para trocear la frase en tokens[1]. Cada token tiene su propio índice y la arquitectura se encarga de convertir ese índice en un vector numérico del tamaño indicado para el codificador.

[1] Generalmente cada token representa una palabra entera pero en algunos casos hay palabras que se dividen en múltiples tokens.

Existen otros tipos de descripciones de imagen que son utilizadas a veces para entrenar este tipo de arquitecturas. Estas descripciones se llaman descripciones densas (*dense captions*) e intentan describir todos los objetos presentes en la imagen. Este tipo de descripciones son mucho más largas que las utilizadas en este proyecto. Si utilizásemos este tipo de descripciones el objetivo del entrenamiento cambiaría significativamente. Con las descripciones utilizadas, las arquitecturas tienen tareas adicionales como interpretar información implícita en la descripción para llenar partes de la imagen que no se han descrito de forma explícita. También tiene más libertad que si se utilizasen descirpciones densas.

3.1.2. Salida

El objetivo de las arquitecturas es la generación de la composición de la imagen. Esta composición no es más que una serie de objetos. Cada objeto tiene una posición (coordenadas x,y), tamaño (anchura, altura) y tipo. En la figura 3.1 se puede ver un ejemplo visualizado de un resultado. Las arquitecturas generan estos objetos de uno en uno. Es por ello que solo hace falta desarrollar una representación de las tres características de los objetos.

En este proyecto nos hemos basado en la representación de *counding-box* utilizada por [Dominguez, 2021]. Después de revisarla, no se ha encontrado que sea una representación problemática. Diferentes aspectos de ella y diferentes opciones fueron ya sopesadas por él. Además, mantener el formato de la salida permite la reutilización de múltiples partes de su código, ahorrando una cantidad de tiempo considerable.

La representación de cada parte de objeto es la siguiente:

- Clase: La clase del objeto a de ser una de las clases de MSCOCO. Como es común en las salidas categóricas, las arquitecturas calculan un vector con tantos valores como clases existen. En este caso: 84. Cada valor de este vector de salida se normaliza a valores entre 0 y 1 utilizando la función *softmax* y se interpretan como la probabilidad de que el objeto sea de cada clase. Más tarde se elige la clase con mayor probabilidad.

- Tamaño (anchura, altura): Tanto la altura como la anchura se representan como un número. Las arquitecturas generan dos números normalizados entre 0 y 1. Donde 0 sería que no tiene altura o anchura y 1 que es tan ancho o alto como la imagen entera.

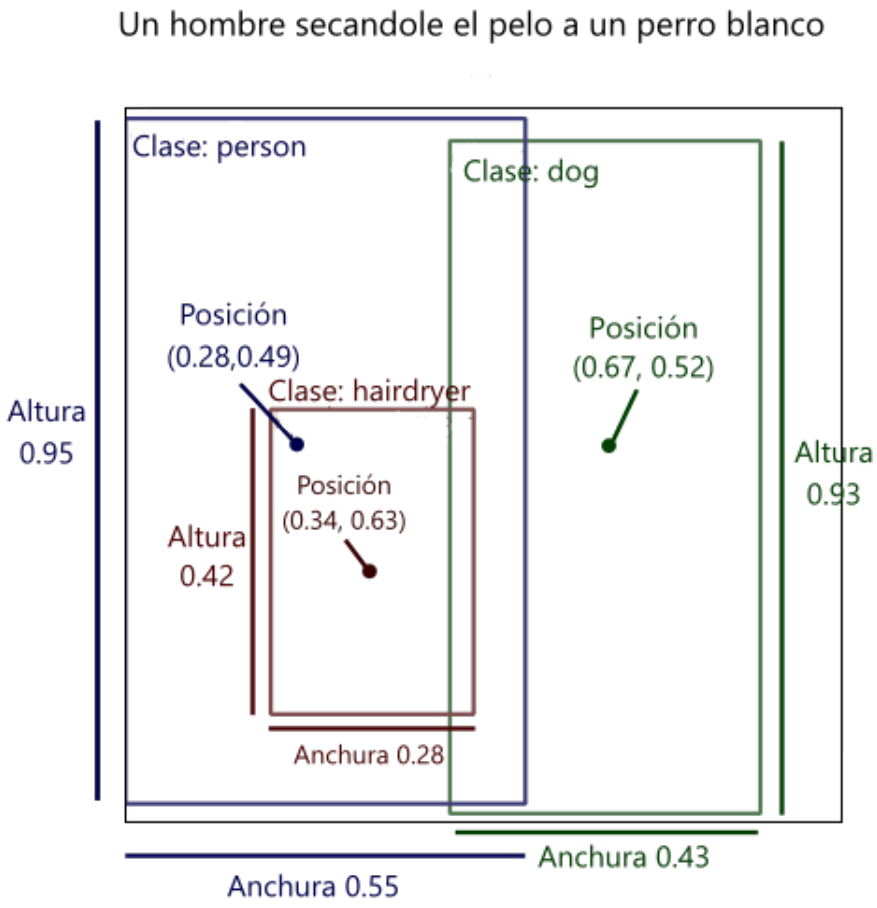

Figura 3.1: Ejemplo de composición de imagen con los diferentes datos de cada objeto marcados. El tipo (clase) de los objetos: *person* (persona), *dog* (perro), *hairdryier* (secador de pelo); posiciones (con las coordenadas normalizadas entre 0 y 1) y tamaños: alturas y anchuras (normalizadas entre 0 y 1). La descripción de la imagen se ha traducido al castellano para facilitar la comprensión. En un ejemplo real la descripción sería en inglés.

- Posición (coordenadas x,y): En vez de representar las coordenadas con dos números, se utiliza una técnica parecida a la de la clase La imagen se reparte en una cuadrícula de tamaño controlable por parámetro (durante el proyecto se ha utilizado una cuadrícula de 32x32). Las arquitecturas generan un vector con tantos valores como casillas tiene la cuadrícula. Cada valor se normaliza a valores entre 0 y 1 utilizando un *softmax* con temperatura y se interpreta como la probabilidad de que el objeto se encuentre en la casilla correspondiente.

  Al elegir qué casilla utilizar como resultado, no se toma simplemente la de mayor probabilidad. Para generar resultados más interesantes y variados, se selecciona de forma aleatoria una casilla pero la probabilidad de que salga cada casilla es su respectivo resultado.

3.2. TRAN2LY

Esta arquitectura toma como base la arquitectura RNN2LY de [Dominguez, 2021] explicada en el apartado 2.3. Para esta arquitectura se ha reemplazado el codificador RNN por un *Transformer-Encoder*. Para ser más precisos, se ha reemplazado por un modelo BERT pre-entrenado [Devlin et al., 2018]. El decodificador se ha mantenido.

Este BERT ha sido pre-entrenado en el idioma inglés para conseguir representaciones contextuales de cada token. Es decir, vectores que representan cada token dentro de la secuencia.

El único problema de la introducción del modelo es la forma de su salida. El RNN decodificador espera un solo vector de tamaño fijo por sentencia. Pero BERT da un vector por token. Hay diferentes maneras de fusionar estos vectores en uno solo. Para este caso nos hemos decantado por un *max-pooling* (visualizado en fig. 3.2). Con el método de *max-pooling* el vector resultante es del mismo tamaño que los vectores a fusionar. Cada elemento del resultado es el elemento de mayor magnitud entre los elementos de la misma posición entre los vectores a fusionar.

De esta manera conseguimos adaptar la salida de un *transformer-encoder* a la entrada requerida por un decodificador RNN. La arquitectura queda como se muestra en la figura 3.3.

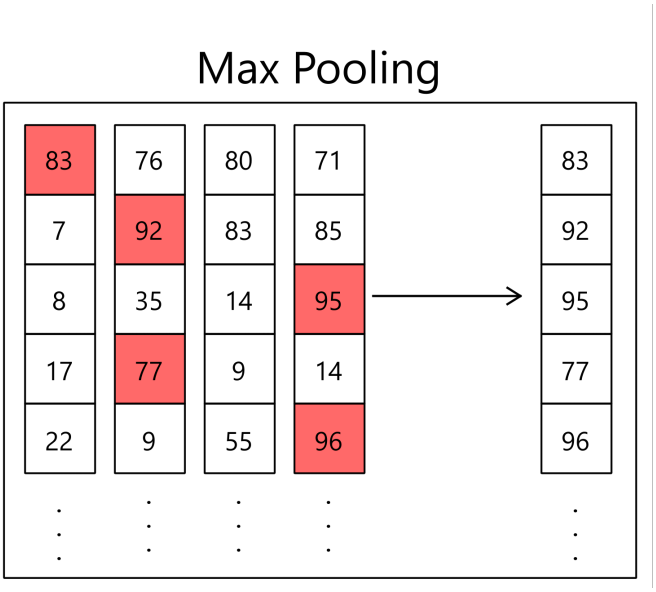

Figura 3.2: Ejemplo del proceso de *max-pooling* fusionando cuatro vectores

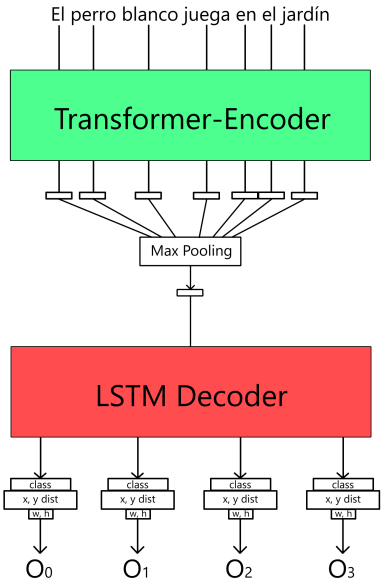

Figura 3.3: Esquema de la arquitectura de TRAN2LY

3.3. STRAN2LY

En esta arquitectura, el codificador es un *sentence-transformer*. Como ya se ha mencionado en el apartado de TRAN2LY, hay veces que se necesita una salida de tamaño fijo por cada sentencia. En el caso de los coldificadores transformer, esto requiere algún tipo de modificación. Al ser un problema común, ha sido solucionado de diferentes maneras y ha resultado en los llamados *sentence-transformers*.

Además de proponer la arquitectura, estos transformers son entrenados para optimizar la representación de sentencias completas. De esta manera, la tarea para la que han sido entrenados es la misma que se utilizan en el proyecto.

Para esta arquitectura se ha escogido el modelo all-mpnet-base-v2 basado en [Song et al., 2020][2]. Se ha escogido por su rendimiento en la tarea *Sentence-Embeddings* (representación de sentencias).

El decodificador RNN de RNN2LY se mantiene en la arquitectura STRAN2LY.

3.4. TRAN2TRAN

En esta arquitectura se ha sustituido también el decodificador. Con este cambio se sustituyen todas las partes del modelo RNN2LY de Carlos.

3.4.1. Codificador

Los decodificadores transformer están diseñados para funcionar con los codificadores transformer. Es decir, no necesitan un vector de tamaño fijo por sentencia sino uno por token. Por ello el codificador de esta arquitectura vuelve a ser el mismo BERT de TRAN2LY pero sin capa *max-pooling*.

3.4.2. Decodificador

El decodificador transformer ha sido entrenado para, por cada objeto de la secuencia de entrada, predecir el siguiente objeto. Es decir, si se le da una secuencia de los primeros

[2]Se ha escogido de entre la selección de la página https://www.sbert.net/docs/pretrained_models.html

cuatro objetos (0,1,2,3), la salida serian los cuatro objetos seguidos empezando desde el segundo (1,2,3,4). En general, dados los objetos [0, N], devuelve los objetos [1, N+1]. De esta salida, los objetos [1, N] no son de interés (ya teníamos la información). El dato importante es el objeto N+1: el modelo ha predicho el siguiente objeto de la secuencia. Añadiendo este nuevo objeto a la secuencia inicial ([0, N+1]) y volviendo a introducirlo, conseguiríamos el objeto N+2 (al final de la secuencia [1, N+2]).

Como se ve, para conseguir un nuevo objeto hace falta una secuencia de todos los anteriores objetos. Para conseguir el primer objeto se introduce una secuencia con un objeto especial: el objeto de inicio de secuencia (SOS en inglés). Este es un objeto con una clase especial (1) y tanto las coordenadas como las dimensiones puestas a 0. A partir de esta secuencia el modelo genera el primer objeto y se inicia la cadena de generación.

Por otro lado, la secuencia se para cuando el último objeto de la salida es el objeto de fin de secuencia (EOS en inglés) o se llega al límite predeterminado de diez objetos. El objeto EOS es cualquier objeto con la clase 2.

Hasta ahora en esta sección se ha mencionado que introducimos objetos al decodificador. En realidad al decodificador se le introducen vectores numéricos y su salida es también una serie de vectores numéricos. En el caso de nuestras versiones, estos vectores numéricos contienen 768 elementos. Los objetos, por otro lado, están constituidos por su clase (1 entero), posición (2 números) y dimensiones (2 números).

Es por ello que antes de introducirlos al modelo hace falta pasar los objetos a un vector de 768 números y en la salida, convertir los vectores numéricos a una clase, posición y dimensiones.

En el caso de la entrada, cada parte del objeto (clase, posición, dimensiones) se introduce en una red neuronal con capas totalmente conectadas (*fully conected*) para conseguir un vector numérico. Los tres vectores numéricos se concatenan para conseguir un único vector representante del objeto completo.

En la salida se utilizan también redes neuronales con capas totalmente conectadas para conseguir cada parte del objeto de uno en uno. Primero se consigue la clase. La clase depende del vector de salida del decodificador y se produce el vector de probabilidades de cada clase de MSCOCO. Después se genera el vector de probabilidades de la posición que, para cada casilla de la imagen, indica la probabilidad de que el objeto esté en esa casilla. La posición no depende solo de la salida del codificador sino también de la salida de la clase recién generada. Por último se producen las dimensiones. Estos dos números dependen tanto del vector de salida del decodificador como de la clase como de la

posición.

4. CAPÍTULO

Conjuntos de Datos Utilizados

En este apartado se introducirán los diferentes conjuntos de datos utilizados. Se empezará explicando el conjunto de datos utilizado para la tarea principal de *text-to-layout*. Esta contiene ejemplos de parejas de datos de entrada y de salida (descripción y composición). Después se describirá el conjunto de datos que hemos utilizado para intentar descubrir si las arquitecturas han desarrollado un sentido común respecto al tamaño que deberían tener los diferentes objetos y las posiciones que deberían tomar. Esta nueva tarea se utilizará como una nueva manera de comparar las arquitecturas finales. Además de describir el objetivo de las tareas de cada conjunto también se darán datos sobre ella y se explicará el pre-procesado al que se ha sometido cada una.

4.1. MSCOCO2014

Microsoft Common Objects in COntext (MSCOCO) [Lin et al., 2014] es un conjunto de datos que contiene un total de 328.000 imágenes y 2.500.000 instancias de objetos. Es un conjunto de datos utilizado frecuentemente en visión por computador. Esto se debe a que por cada imagen el conjunto contiene hasta 5 descripciones de imagen con una descripción general; *bounding-box*, clase y mapa de segmentación de cada objeto presente en la imagen.

En su proyecto [Dominguez, 2021] efectúa diferentes tipos de pre-procesado. El más sencillo es la limitación de número de objetos por imagen a 10, cambio que en este proyecto se mantendrá. Los objetos que se mantienen son los que mayor área ocupan, es decir, los

principales de la imagen. A partir de los 10 objetos, los restantes son objetos pequeños de poca importancia que arriesgan introducir ruido en el entrenamiento (a raíz de darles demasiado peso).

También calcula grafos AMR y SGP para cada descripción de imagen pero en este proyecto no son necesarios ya que ninguna de las arquitecturas utiliza ninguno de los dos. Menciona la posibilidad de filtrar descripciones de imagen que no son útiles por su especificidad o uso de nombres propios (de lugares, por ejemplo). Ni en su proyecto se realizó ni este se ha realizado este filtrado ya que requeriría la revisión manual de cada descripción de imagen, dejando poco tiempo para el resto de tareas.

El último pre-procesado que se hace es la eliminación de los ejemplos que no contienen ningún objeto.

El conjunto se divide en conjuntos de entrenamiento, desarrollo y test. Después de los pre-procesados realizados, cada uno contiene 73870, 8211 y 40504 imágenes, respectivamente. De cada imagen sólo se utilizará una descripción de imagen para evitar dar más peso a imágenes con más descripciones de imagen.

4.2. Spatial Commonsense Dataset

El *Spatial Commonsense Dataset* [Liu et al., 2022] es un conjunto de datos creado para evaluar el conocimiento espacial básico de diferentes modelos de manera sencilla e intuitiva. En este proyecto, se ha utilizado el total del conjunto como conjunto de evaluación. No se ha dividido en subconjuntos para entrenamiento ni desarrollo. El conjunto contiene información de diferente tipo: tamaño, altura y posición relativa.

El conjunto de datos está hecho en inglés y se ha utilizado en inglés pero para facilitar la comprensión en este documento se han traducido al castellano los datos y los ejemplos.

4.2.1. Tamaño

En este apartado del conjunto se encuentra información sobre el tamaño de diferentes objetos. Más específicamente, se definen cinco niveles diferentes de tamaño y se colocan todos los objetos en su respectivo nivel de tamaño. Los diferentes objetos en sus respectivos niveles se pueden ver en la tabla 4.1. El primer nivel contiene los objetos más pequeños y el último los más grandes.

Tamaño	
1	hormiga, moneda, nuez, bala, dado
2	pájaro, taza, concha, botella, billetera
3	neumático, silla, microondas, perro, maleta
4	humano, sofá, estantería, tigre, cama
5	casa, cine, montaña, camión, avión

Tabla 4.1: Todos los niveles de tamaño con sus objetos.

Los niveles están elegidos de forma que la diferencia de tamaño entre objetos de diferentes niveles es siempre significativa. De esta manera, si le preguntásemos a una persona cual es mayor, la persona debería de ser capaz de responder decididamente que el objeto en el nivel mayor es más grande. Por ejemplo, cualquier persona sabe que una estantería (nivel 4) es más pequeña que un avión (nivel 5).

A partir de esta información se generan ejemplos con resultados conocidos. Se emparejan objetos de diferentes niveles y se apunta por cada pareja cual de los dos objetos debería de ser mayor. Por cada pareja, se produce también una frase para darle como entrada a la arquitectura.

Por ejemplo, se podrían emparejar pájaro (nivel 2) y silla (nivel 3). Se apuntaría como resultado correcto que silla es el objeto más grande. Por último, se generaría el texto a introducir en la arquitectura: ”Un pájaro y una silla”. De esta manera, se esperaría que la arquitectura generase un pájaro y una silla y después habría que comprobar si la silla generada es más grande que el pájaro (fig. 4.1).

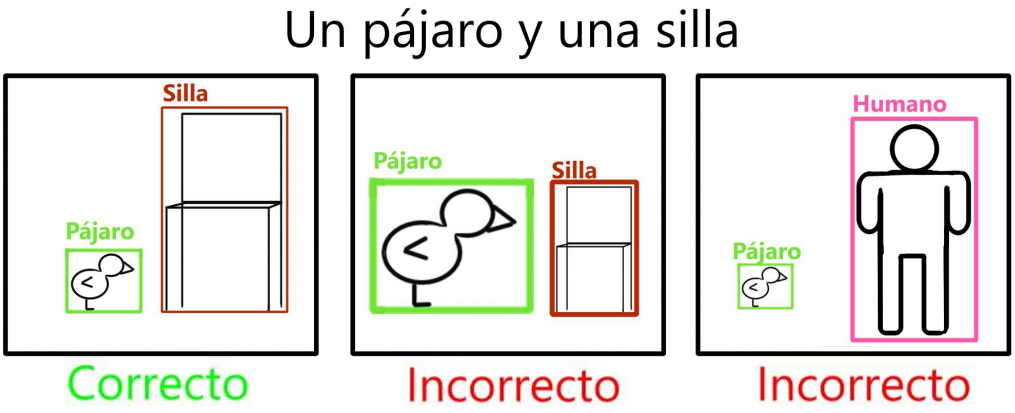

Figura 4.1: Tres ejemplos de posibles salida a la entrada ”Un pájaro y una silla”. Cada salida está marcada como correcta o incorrecta.

Cabe notar que por cada pareja se genera también una pareja simétrica. Es decir, se genera

tanto ”Un pájaro y una silla” como ”Una silla y un pájaro”. Es para evitar puntuar de forma injusta a arquitecturas que tengan preferencia a generar siempre el primer objeto más grande o viceversa.

El conjunto incluye tanto la información de los niveles y los objetos que hay en ellos como todas las parejas ya generadas con sus textos.

4.2.2. Altura

El apartado de altura es idéntico al de tamaño pero la información que guarda es sobre la altura de los objetos en vez de su tamaño. Los objetos que aparecen en este apartado no son los mismos que en el de tamaño (aunque si comparten muchos).

En este apartado se definen cinco niveles diferentes de alturas y se colocan todos los objetos en su respectivo nivel de altura. Todos los objetos en su respectivo nivel se pueden ver en la tabla 4.2. El primer nivel contiene los objetos más bajos y el último los más altos.

Altura	
1	hormiga, insecto, gota de agua, bala, dado
2	pájaro, vaso, zapato, botella, teléfono móvil
3	mesa, silla, cubo de basura, sofá, maleta
4	humano, caballo, estantería, camello, puerta
5	apartamento, teatro, jirafa, camión, farola

Tabla 4.2: Todos los niveles de altura con sus objetos.

Al igual que en el apartado de tamaño, los niveles están elegidos para que haya una diferencia significativa de altura entre objetos de diferentes niveles. Así la diferencia es suficientemente evidente como para que cualquier persona no tenga dudas al elegir qué objeto es más alto al presentarle dos objetos de diferentes niveles.

De la misma manera que para los tamaños, los ejemplos se generan emparejando objetos de diferentes niveles, apuntando como resultado cual de los dos es más alto y generando texto a partir de la pareja. Por ejemplo, al emparejar sofá (nivel 3) y humano (nivel 4) se apuntaría que el objeto más alto es humano y se generaría el texto ”Un sofá y un humano”. Algunos ejemplos de posibles generaciones a partir de este texto y su resultado se ven en la figura 4.2.

De la misma manera que con los tamaños, para cada pareja se genera también su pareja simétrica.

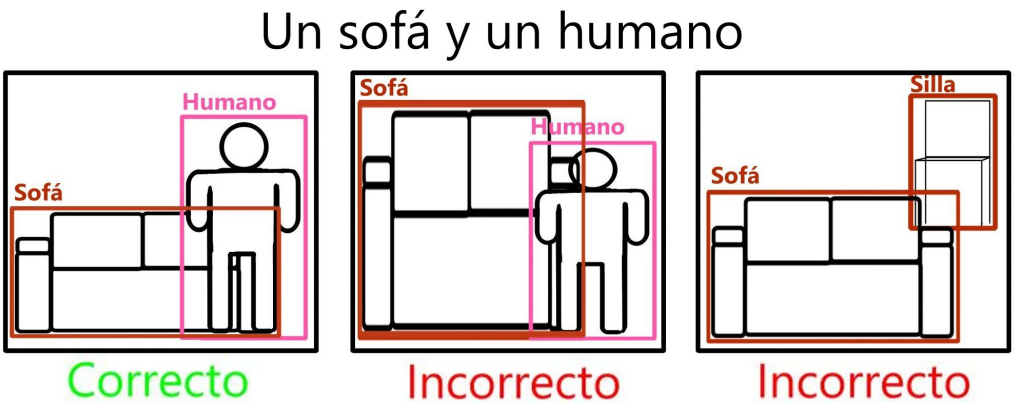

Figura 4.2: Tres ejemplos de posibles salida a la entrada ”Un sofá y un humano”. Cada salida está marcada como correcta o incorrecta.

El conjunto incluye tanto la información de los niveles y los objetos que hay en ellos como todas las parejas ya generadas con sus textos.

4.2.3. Posición Relativa

Este apartado es el más diferente al resto.

Este apartado recoge una colección de descripciones de diferentes situaciones. Por ejemplo: ”Un niño andando en bicicleta”. Cada situación está compuesta por dos objetos interactuando entre sí. En el ejemplo anterior los dos objetos son ”niño” y ”bicicleta”. Por cada situación, se recoge también qué posición tiene el primer objeto respecto al segundo: dentro, encima, debajo o al lado. En el ejemplo de antes, el resultado correcto sería ′encima′, porque el primer objeto (niño) debería de estar encima del segundo (bicicleta). En el ejemplo ”Un hombre conduce un coche”, el resultado sería ′dentro′ porque el hombre está dentro del coche.

4.2.4. Procesamiento de Salida

El conjunto está pensado para ser utilizado con diferentes tipos arquitectura. Al ser tan genérico, hay veces que hace falta algo más de procesado para extraer la información necesaria del resultado para compararlo con los datos del conjunto. Es decir, la respuesta apuntada por el conjunto es (por ejemplo) el hecho de que el primer objeto es más grande que el segundo. Sin embargo las arquitecturas dicen explícitamente qué objeto es más

grande. Simplemente, generan una composición o una imagen, así que hay que extraer información de tamaños y posiciones de ella.

En nuestro caso, este procesado es mínimo ya que por cada objeto de la composición tenemos su tipo, posición y dimensiones. Es por ello que podemos saber fácilmente toda la información necesaria:

- Podemos saber si ambos objetos están en la composición porque tenemos las clases de todos los objetos.
- Podemos saber su posición relativa porque tenemos ambas posiciones (y su tamaño para calcular si uno está dentro de otro).
- Podemos calcular si un objeto es más alto que otro porque tenemos la altura de ambos objetos.
- Podemos calcular si un objeto es más grande que el otro porque tenemos las dos dimensiones de cada. En nuestro caso tomamos tamaño como área del *bounding-box* que representa el objeto.

Como detalle: en caso de que haya más de un objeto con la misma clase, solo se tendrá en cuenta el más grande de ellos. Consideramos el objeto más grande como el principal de la imagen. Así, si el ejemplo es ”Un hombre limpiando un coche” sería de esperar que la persona principal de la imagen fuese el hombre que limpia el coche. Además cualquier objeto adicional que no sea de las clases esperadas será ignorado. Es decir, no hay penalización por generar objetos que no aparezcan explícitamente en la descripción dada.

Los autores del conjunto en su artículo [Liu et al., 2022] evalúan arquitecturas de generación de imágenes. Como tal, para conseguir la información del tipo, tamaño y posición de los objetos, hace falta detectarlos dentro de la imagen. En el artículo, realizan esta detección tanto de forma automática como manual y publican los resultados de ambos métodos. También evalúan arquitecturas de procesado del lenguaje natural. Para este tipo de arquitecturas el proceso de evaluación cambia más que para las arquitecturas de generación de imagen. En su artículo [Liu et al., 2022] se explica con mayor detalle el proceso específico para arquitecturas de procesamiento de lenguaje natural.

4.2.5. Pre-procesado

Antes de poder utilizar el conjunto se han de efectuar diferentes pre-procesados.

Como este es un conjunto independiente de MSCOCO, puede contener (y contiene) palabras que no están presentes en el conjunto de entrenamiento. Como las arquitecturas se han entrenado en MSCOCO, no se puede esperar que funcionen correctamente con palabras que no están presentes en ningún ejemplo visto. Por ello, el primer filtro aplicado es la eliminación de cualquier ejemplo que contenga al menos una palabra que no esté presente en algún descripción de imagen utilizado de MSCOCO.

El segundo problema es que hay que emparejar los objetos presentes en el ejemplo del conjunto con objetos en la imagen. Para ello necesitamos que cualquier objeto de los ejemplos puedas ser representado por alguna clase de MSCOCO. En este segundo preprocesado se mappean todos los objetos del conjunto a una clase de MSCOCO y en caso de que no se puedan mappear se elimina el ejemplo que lo contenga. Es decir, si el ejemplo original era ”un hombre conduciendo un coche” donde los objetos eran ”hombre” y ”coche”, la frase se mantendrá igual pero los objetos se cambiaran a ”persona” y ”coche” (ambas clases presentes en MSCOCO).

Este mapeo se ha hecho de forma manual. El primer paso ha sido hacer las uniones evidentes (como las del ejemplo dado). Una vez hechas estas, se han revisado el resto de objetos sin mappeo. Para cada uno se han buscado descripciones de imagen en MSCOCO que contengan la palabra para ver si en sus respectivas imágenes había *bounding-boxes* para el objeto a buscar. En general este último paso no ha dado fruto y las clases asignadas en el primer paso son casi las únicas que se han conseguido mappear. Por ejemplo en este conjunto de datos un posible objeto era camello, pero la única descripción de imagen que contenía la palabra era ”un hombre montado en un camello” y la imagen no tenía ningún *bounding-box* para representar el camello. Otros objetos como ”bala” aparecían más a menudo en MSCOCO pero con significados completamente diferentes: en MSCOCO se referían a trenes bala mientras que en este se referían a balas de arma.

Después de estos pre-procesados los subconjuntos de tamaño, altura y posición relativa pasan de tener 500, 500 y 224 ejemplos a tener 106, 104 y 104 respectivamente. Aunque el número es significativamente menor, consideramos que siguen siendo suficientes para la valoración de las arquitecturas.

5. CAPÍTULO

Experimentos y Resultados

En este apartado se describirán diferentes detalles sobre el entrenamiento de las arquitecturas.

Además de su entrenamiento se describirá también el método de evaluación que se ha llevado a cabo. Se presentarán los resultados de las comparaciones de nuestras arquitecturas con las de [Dominguez, 2021] y otras arquitecturas del estado del arte.

Primero se evaluará el rendimiento de las arquitecturas en la tarea principal *text-to-layout* utilizando la partición de test de MSCOCO. Después se evaluarán sus habilidades de sentido común espacial generales utilizando el conjunto de datos *Spatial Commonsense Dataset*. Aunque la tarea no es para la que se han entrenado los modelos, este test nos da una idea de las habilidades espaciales generales que han conseguido las arquitecturas.

También se mostrarán ejemplos de diferentes composiciones generadas por nuestras arquitecturas y se evaluarán de forma cualitativa.

5.1. Entrenamiento

El entrenamiento se ha llevado acabo en la partición de entrenamiento de MSCOCO. Como se ha mencionado en el apartado 4.1, ésta parte del conjunto de datos contiene un total de 74504 ejemplos.

Mientras se consiguen los resultados de entrenamiento, las arquitecturas también se evalúan en el conjunto de desarrollo. Los resultados en desarrollo no se utilizan para entrenar

las arquitecturas sino que sirven como referencia al rendimiento de las arquitecturas en ejemplos no antes vistos por la misma.

Cada arquitectura se ha entrenado durante 100 épocas. Además cada arquitectura se ha entrenado manteniendo el codificador congelado[1] y sin congelar. En los resultados se indicará que una arquitectura tiene el codificador congelado añadiendo $_F$ a su nombre. Se indicará que el codificador no está congelado con $_{UF}$

5.1.1. Pérdida

Las pérdidas (*loss* en inglés) utilizadas no se ha cambiado del proyecto de Carlos [Dominguez, 2021]. Aunque habría sido interesante revisarlos, no ha habido suficiente tiempo para ello. Como las pérdidas de Carlos eran funcionales, el cambio no era prioritario.

Las pérdidas comparan el resultado de una sola *bounding-box* con su respectiva *bounding-box* de *ground-truth*. Por cada ejemplo de entrenamiento se calculan estas perdidas para cada *bounding-box*.

Las pérdidas consisten en:

- Clase: *CrossEntropyLoss*. La pérdida de entropía cruzada (*CrossEntropyLoss*) se utiliza en tareas de clasificación. Lo que hace es comparar el vector de probabilidades dado por la arquitectura (donde a cada posible tipo se le da una probabilida) y se compara con la verdadera probabilidad de cada tipo en el *ground-truth*. La entropía cruzada (*cross-entropy*), simplificando, mide cómo de similares son dos vectores entre sí (como de correlacionados están). Para nuestro caso, es más fácil ver por qué funciona viendo su fórmula (eq. 5.1)

$$\text{Pérdida} = -\sum_i gt_i \cdot \ln y_i \tag{5.1}$$

Donde gt_i es la probabilidad de que el objeto sea de la clase i en el *ground-truth* e y_i en el resultado dado por la arquitectura para la clase i.

En nuestro caso el *ground-truth* sabe exactamente de qué clase es el objeto así que la probabilidad de gt_k será 100 % (k siendo la clase correcta) y el resto tendrán un

[1] Se le llama congelar a no permitir que sus parámetros cambien de valor. Se mantienen con los valores de inicio.

0 %. Esto da como resultado un vector *one-hot encoding* donde todos los valores en el vector son cero menos uno. Sabiendo esto, la fórmula 5.1 se reduce a 5.2.

$$\text{Pérdida} = -1 \cdot \ln y_k \tag{5.2}$$

Donde y_k es la probabilidad que la arquitectura le ha dado a la clase correcta.

Así vemos como la pérdida depende únicamente de cuánta probabilidad le da la arquitectura a la clase correcta. Cuando y_k más se acerca a 1, $\ln y_k$ se acerca a 0 haciendo que la pérdida se anule. Sin embargo cuando y_k tiende a cero, lny_k tiende a menos infinito, haciendo que la pérdida tienda a infinito.

- Tamaño: *MSE_loss*. En este caso la pérdida es más sencilla. Simplemente se toma la diferencia entre los valores del *ground-truth* y los resultados. Estas diferencian se elevan después al cuadrado (eq. 5.1.1). Así el error es siempre positivo y se castigan más errores mayores que menores. Es decir, la diferencia de pérdida entre un error de 4,5 y 5 es 4,75 mientras que la diferencia de pérdida entre un error de 2 y uno de 2,5 es de 2,25.

$$\text{Pérdida} = \sum_i (gt_i - y_i)^2 \tag{5.3}$$

Donde gt son los valores del *ground-truth* e 'y' son los resultados dados por la arquitectura.

- Posición: *CrossEntropyLoss*. En este caso se está tratando la generación de una posición como un problema de clasificación. La imagen se divide en una cuadrícula y el problema pasa a ser el de acertar en qué casilla se encuentra el objeto. De esta manera utilizamos la misma pérdida que para el tipo. Pero y_i pasa a ser la probabilidad que se le ha dado al objeto de estar en la casilla i y el objetivo es maximizar la probabilidad de la casilla en la que se encuentra el objeto en el *ground-truth*.

5.2. Métricas de Evaluación

La métrica más utilizada para la evaluación de resultados relacionados con *bounding-boxes* es la Intersección sobre la Unión o IoU. Simplificando, esta métrica mide en cuánto área coinciden dos *bounding-boxes* entre ellas. Es especialmente utilizada en detección de objetos donde, en caso de que el resultado no esté en el mismo sitio que el *ground truth*, el resultado es erróneo. Este no es el caso para *text-to-layout*. Para la misma descripción múltiples resultados son posibles y las *bounding-boxes* no tienen por qué coincidir. El ejemplo 5.1 es un ejemplo de cómo un posible resultado correcto puede no coincidir con el *ground truth*.

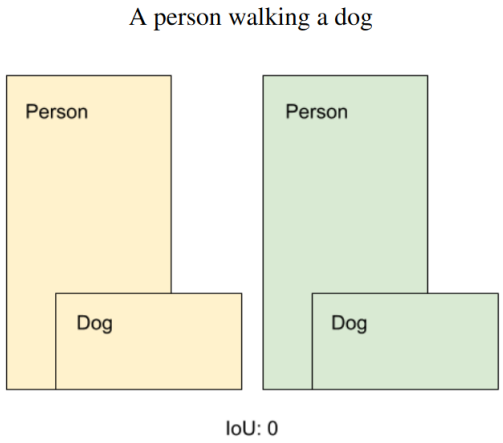

Figura 5.1: Ejemplo de una mala puntuación IoU para una composición correcta. Los *bounding-box* del *ground-truth* en amarillo y los predichos en verde. La composición es idéntica al *ground-truth* y sigue siendo coherente con el texto pero no consigue buena puntuación por estar desplazada

Es por ello que [Dominguez, 2021] desarrolló nuevas métricas para la evaluación de composiciones. Las métricas de que desarrolló son las siguientes:

- Posición Espacial Relativa Categórica (RSCP). Por cada par de objetos en la composición, se comprueba la posición relativa de uno respecto al otro: a su izquierda, a su derecha, encima o debajo. Este resultado se compara después a la posición relativa de los objetos correspondientes en el *ground truth*.

- Proporción (AR). Se compara la proporción de altura-anchura de cada *bounding-box* respecto al *bounding-box* respectivo en el *ground-truth*.

- *Class Matching*. Comprueba que las clases predichas por cada objeto sean iguales al *ground-truth*. Más concretamente, se calculan el F1-score (F1), la precisión (P) y el *recall* (R).
- Diferencia de Escala Relativa (RS). Por cada par de objetos, mide la diferencia de tamaño entre uno y otro. Después se comparan con los valores del *ground-truth*

Para evaluar cada arquitectura se toman estas métricas, se normalizan y se suman para conseguir una única métrica general:

$$\text{puntuación} = RSCP + (1 - AR) + (1 - softmax(RS)) + P + F1 + R \tag{5.4}$$

Hay que denotar que parte del proceso de cada métrica es el emparejamiento de cada objeto predicho con un objeto del *ground-truth*. En nuestro caso, los objetos del *ground-truth* están ordenados por tamaño de mayor a menor y las arquitecturas están entrenadas para que generen los objetos en ese mismo orden. Es por ello que el proceso de emparejado es trivial ya que se espera que el propio orden indique el emparejamiento.

En este apartado se ha hecho un resumen rápido de cada métrica. Para mayor detalle se puede revisar el apartado *5.2 Proposed Metrics* de [Dominguez, 2021].

5.3. Selección de Modelo

Como ya se ha comentado, cada arquitectura es entrenada durante 100 épocas. Para representar cada arquitectura queremos tomar la mejor versión de ella, que no tiene por qué ser la última época.

Para ello, por cada arquitectura, se ha elegido la época con mejor puntuación (eq. 5.2) total. Las métricas son calculadas utilizando los ejemplos y respectivos resultados de la partición de desarrollo. En caso de que más de una época consiga la misma puntuación, se toma la época más cercana al final del entrenamiento. Esto se debe a que después de un análisis cualitativo, se ha observado que en general las épocas más tardías suelen conseguir resultados ligeramente mejores incluso teniendo una puntuación igual.

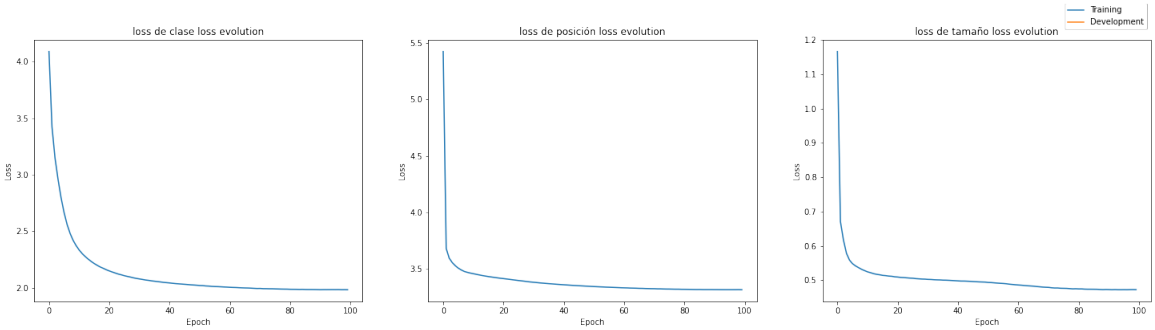

Figura 5.2: Pérdidas de $RNN2LY_F$ en entrenamiento (azul) y en desarrollo (naranja)

5.4. Nombramiento de Arquitecturas

Las arquitecturas desarrolladas durante el proyecto además de las versiones entrenadas de $RNN2LY$ tendrán su respectivo nombre en los resultados ($TRAN2LY$, $STRAN2LY$, $TRAN2TRAN$). Cada uno tiene además dos versiones: una congelando el codificador y otra con el codificador descongelado. Las versión con el codificador congelado irán marcadas con un $_F$ y las no congeladas con $_{UF}$.

Las arquitecturas externas a este proyecto tendrán el nombre dado en su respectivo artículo y se explicará el significado de las marcas adicionales.

5.5. Resultados en MSCOCO

En esta sección se mostrarán y comentarán los diferentes resultados conseguidos en las diferentes particiones de MSCOCO.

5.5.1. Pérdidas en Entrenamiento y desarrollo

Los resultados de las pérdidas durante el entrenamiento tanto en el subconjunto de entrenamiento como en el de desarrollo de cada arquitectura se muestran en las figuras 5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 5.8 y 5.9

Como se puede comprobar, los resultados en entrenamiento parecen buenos: las arquitecturas parecen aprender. Sin embargo los resultados no parecen tan buenos en desarrollo. Este fenómeno (pérdidas atascándose en desarrollo) también le ocurría a [Dominguez, 2021].

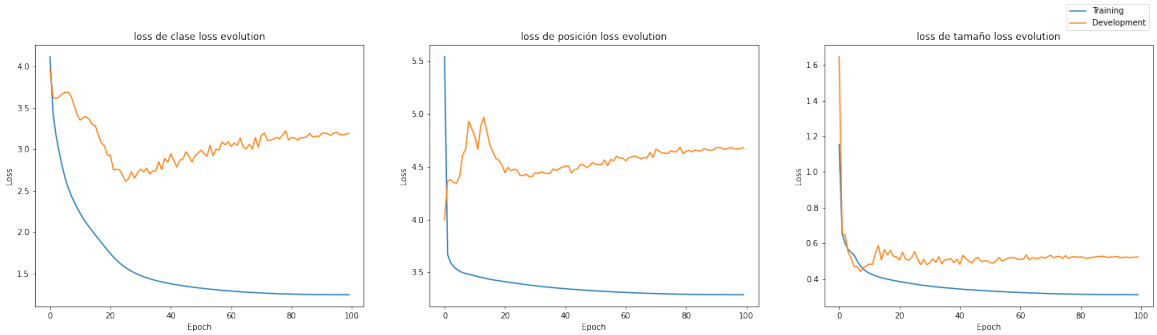

Figura 5.3: Pérdidas de $RNN2LY_{UF}$ en entrenamiento (azul) y en desarrollo (naranja)

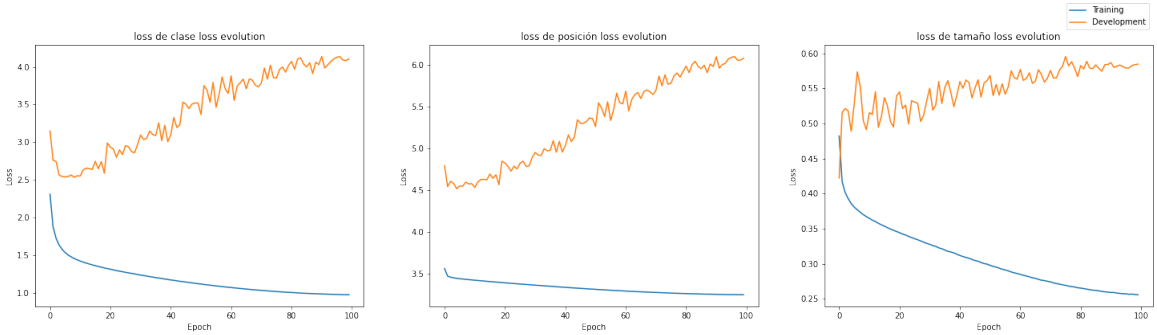

Figura 5.4: Pérdidas de $TRAN2LY_F$ en entrenamiento (azul) y en desarrollo (naranja)

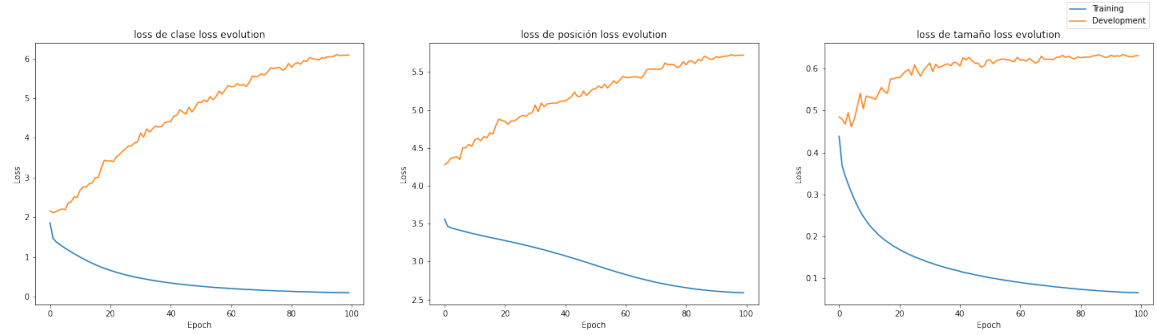

Figura 5.5: Pérdidas de $TRAN2LY_{UF}$ en entrenamiento (azul) y en desarrollo (naranja)

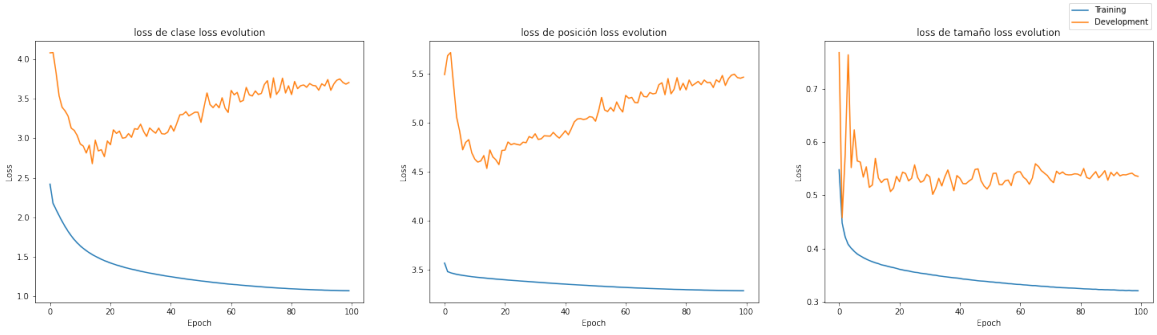

Figura 5.6: Pérdidas de $STRAN2LY_F$ en entrenamiento (azul) y en desarrollo (naranja)

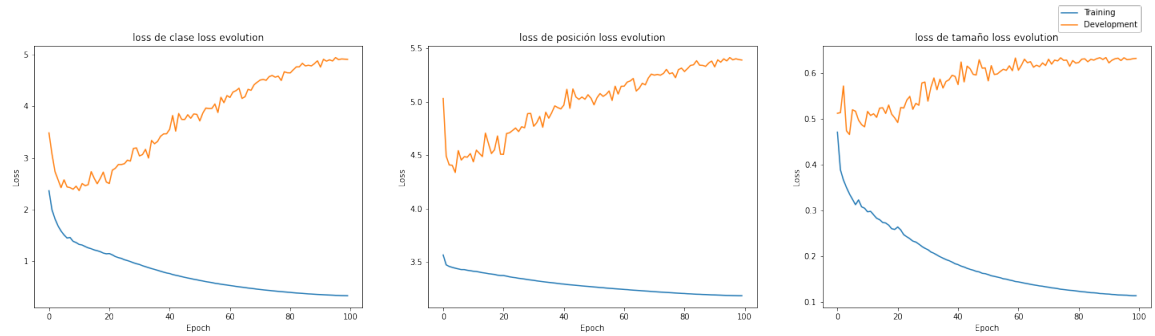

Figura 5.7: Pérdidas de $STRAN2LY_{UF}$ en entrenamiento (azul) y en desarrollo (naranja)

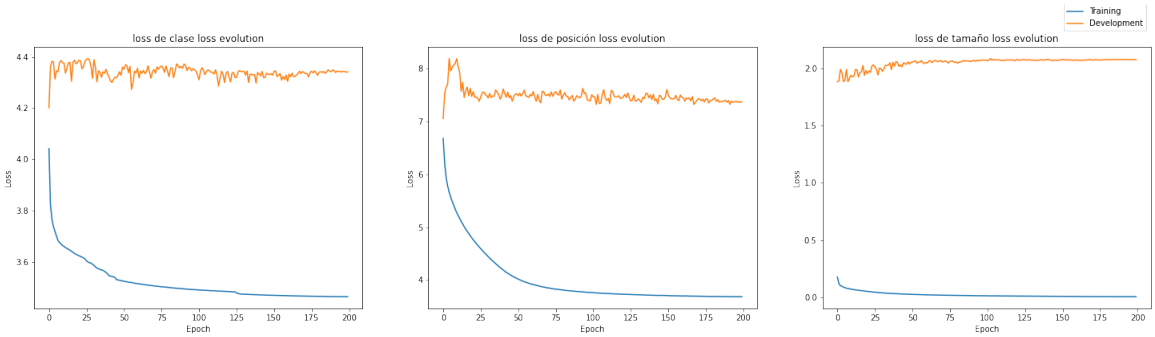

Figura 5.8: Pérdidas de $TRAN2TRAN_F$ en entrenamiento (azul) y en desarrollo (naranja)

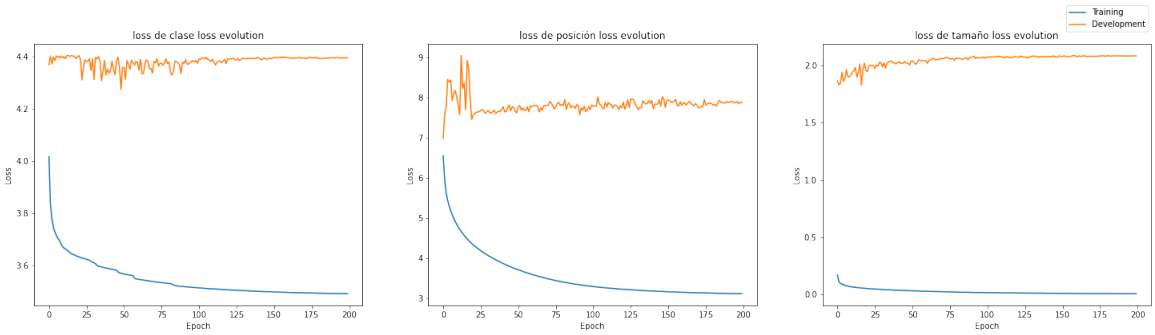

Figura 5.9: Pérdidas de $TRAN2TRAN_{UF}$ en entrenamiento (azul) y en desarrollo (naranja)

Sin embargo, como se mostrará a continuación, al evaluar los modelos utilizando diferentes métricas se ve que los modelos verdaderamente están mejorando en general, no solo en el subconjunto de entrenamiento.

5.5.2. Resultados de Métricas en desarrollo

Los resultados de las arquitecturas que se han entrenado en este proyecto se puede ver en la figura 5.10. Se muestra la evolución de la puntuación 5.2 de todas las arquitecturas durante las 100 épocas de entrenamiento.

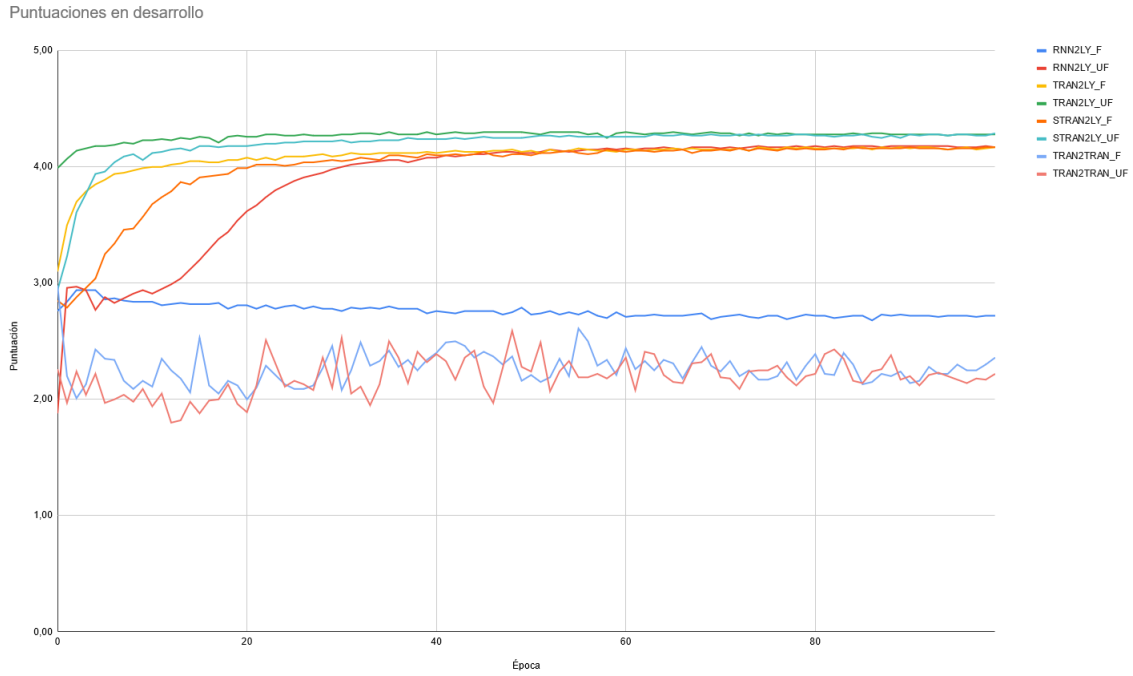

Figura 5.10: Puntuación (eq. 5.2) de las arquitecturas $RNN2LY_F$, $RNN2LY_{UF}$, $TRAN2LY_F$, $TRAN2LY_{UF}$, $STRAN2LY_F$, $STRAN2LY_{UF}$, $TRAN2TRAN_F$ y $TRAN2TRAN_{UF}$ durante sus respectivas 100 épocas de entrenamiento.

A primera vista se puede ver que $TRAN2TRAN_F$, $TRAN2TRAN_{UF}$ y $RNN2LY_F$ están claramente por debajo del resto. Estos primeros resultados ya indican los malos resultados conseguidos por estas arquitecturas. En el caso de $RNN2LY_F$ tiene sentido ya que el codificador ha sido congelado y no estaba pre-entrenado, de forma el resultado intermedio del codificador está siendo calculado de forma aleatoria.

En el caso de $TRAN2TRAN$ se debe a un fallo de la arquitectura. No se han conseguido mejores resultados con ella durante el proyecto.

Otra característica que se puede ver en el resto de arquitecturas es que la puntuación se estanca en épocas bastante tempranas. Las arquitecturas que utilizan codificadores pre-entrenados llegan a su máximo aún antes que el resto. Es decir, es probable que consigamos resultados muy similares entrenando las arquitecturas durante menos épocas.

Este estancamiento en las métricas hace que gran parte de las épocas se consideren de calidad igual o muy similar. Las épocas más tardías de cada arquitectura tienden a tener rendimiento algo mejor. Es decir, si tomamos dos épocas de la misma arquitectura con la misma puntuación, generalmente parece que la época más tardía es mejor. Nos hemos dado cuenta de este detalle después de hacer diferentes análisis cualitativos. Es por ello que los representantes de cada arquitectura son, dentro de las épocas con la puntuación máxima, las más tardías.

Aunque la figura 5.10 ayuda a ver la evolución de la puntuación de cada arquitectura, para ver los valores de cada una y compararlos es mejor la tabla 5.1. Esta muestra los mejores resultados de cada arquitectura en el conjunto de datos de desarrollo y el rango en el que se encuentran todas las épocas de cada arquitectura que tienen una puntuación cercana a la mejor obtenida por la arquitectura.

Arquitectura	Puntuación5.2	Época	Rango $\pm$ 0.03
$RNN2LY_F$	2.94	4	2-4
$RNN2LY_{UF}$	4.18	98	52-99
$TRAN2LY_F$	4.17	99	46-99
$TRAN2LY_{UF}$	4.30	69	19-99
$STRAN2LY_F$	4.17	99	54-99
$STRAN2LY_{UF}$	4.29	99	50-99
$TRAN2TRAN_F$	2.98	0	0-0
$TRAN2TRAN_{UF}$	2.59	48	48-48

Tabla 5.1: Mejor puntuación conseguida por cada arquitectura. *Época* indica la época más tardía en la que ha conseguido la mejor puntuación. *Rango $\pm$ 0.03* indica el rango de épocas mínimo que incluye todas las épocas que han conseguido un resultado que está a menos de 0.03 puntos de distancia del mejor.

En la tabla se ve que, aunque los resultados son muy parejos para todas las arquitecturas menos $RNN2LY_F$, $TRAN2TRAN_F$ y $TRAN2TRAN_{UF}$, parece que $STRAN2LY_{UF}$ y $TRAN2LY_{UF}$ son algo mejores que el resto.

Como se ha mencionado, por cada arquitectura se tomará la última época que haya conseguido la mejor puntuación (columna *Época* en la tabla 5.1) como representante de la arquitectura.

5.5.3. Resultados en Test

En este apartado se mostrarán los resultados de cada arquitectura en el subconjunto de datos de test de MSCOCO. Los resultados de cada arquitectura se juzgarán utilizando las métricas de la ecuación 5.2.

Arquitectura	RSCP ↑	AR ↓	RS ↓	P ↑	F1 ↑	R ↑	Punt.[a]↑
$Obj-GAN$	0.348	0.246	2216.491	0.866	0.566	0.499	-
$RNN2LY_{PT}$	**0.464**	**0.188**	7.587	0.907	0.609	0.534	-
$RNN2LY_{FT}$	0.459	0.190	**7.478**	**0.911**	0.606	0.529	-
$GCN2LY$	0.449	0.191	7.540	0.893	0.594	0.520	-
$RNN2LY_{F}$	0.43	0.25	7.61	0.37	0.2	0.17	2.92
$RNN2LY_{UF}$	0.38	0.19	8.6	0.89	0.59	0.52	4.18
$TRAN2LY_{F}$	0.37	0.2	9.79	0.89	0.59	0.52	4.16
$TRAN2LY_{UF}$	0.39	0.21	10.64	0.9	**0.64**	**0.58**	**4.30**
$STRAN2LY_{F}$	0.36	0.2	9.15	0.89	0.6	0.52	4.17
$STRAN2LY_{UF}$	0.41	0.21	10.83	0.89	0.63	0.57	4.28
$TRAN2TRAN_{F}$	0.42	0.22	7.65	0.37	0.20	0.18	2.95
$TRAN2TRAN_{UF}$	0.36	0.26	9.03	0.24	0.13	0.11	2.59

Tabla 5.2: Resultados de cada arquitectura de las diferentes métricas explicadas en 5.2 en el subconjunto de test de MSCOCO. Se añade también como referencia los resultados que consiguió [Dominguez, 2021] y [Li et al., 2019]. Las arquitecturas de [Dominguez, 2021] son $RNN2LY_{PT}$: *RNN2LY* con codificador pre-entrenado, $RNN2LY_{FT}$: *RNN2LY* con codificador pre-entreando y *fine-tuneado* y *GNC2LY*: una arquitectura diferente que utiliza un codificador GCNN[b] para analizar la entrada que ha sido previamente convertido a un grafo de escena. La arquitectura de [Li et al., 2019] es ObjGAN.

[a]Para el cálculo de la puntuación en la tabla 5.2 hace falta normalizar RS. Esta normalización depende de que haya múltiples épocas siendo evaluadas. En esta tabla solo hay una época siendo evaluada por cada arquitectura. Para el cálculo, se ha utilizado el valor de RS normalizado de desarrollo. En el caso de las arquitecturas externas ([Dominguez, 2021] y [Li et al., 2019]) no tenemos esta información.

[b]Red Neuronal Convolucional para Grafos (Graph Convolutional Neural Network)

Las puntuaciones de las arquitecturas de este proyecto se mantienen similares a las conseguidas en desarrollo. Es por ello que mantenemos las conclusiones conseguidas en desarrollo: $RNN2LY_{F}$, $TRAN2TRAN_{F}$ y $TRAN2TRAN_{UF}$ tienen resultados pobres, $STRAN2LY_{UF}$ y $TRAN2LY_{UF}$ son las mejores, seguidas cerca por las versiones congeladas y $RNN2LY_{UF}$.

El primer hecho que resalta al comparar nuestras arquitecturas con las externas es la diferencia entre los modelos *RNN2LY* entrenados por nosotros y los entrenados por [Dominguez, 2021]. Como las de [Dominguez, 2021] consiguen mejores resultados, serán esas las que se utilicen como punto de referencia para la arquitectura *RNN2LY* ya que parece haber conseguido sacar más probecho de ella.

$TRAN2LY_{UF}$ y $STRAN2LY_{UF}$ consiguen competir con las arquitecturas de [Dominguez, 2021]. En *F1 score* y *Recall* nuestras arquitecturas consiguen una ventaja considerable respecto a las de [Dominguez, 2021]. Sin embargo, ni $TRAN2LY_{UF}$ ni $STRAN2LY_{UF}$ consiguen superar las puntuaciones en Precisión, *RSCP*, *RS* o *AR* de [Dominguez, 2021].

Cabe mencionar que, aunque la métrica *RSCP* tenga completo sentido en teoría, los resultados positivos en esta métrica por parte de nuestras peores arquitecturas son una mala señal para el valor de la métrica.

En general, los resultados conseguidos son parcialmente positivos. Al competir con los modelos de [Dominguez, 2021] estamos compitiendo con arquitecturas del estado del arte y superando con creces modelos como $Obj-GAN$. Sin embargo la diferencia entre los resultados de nuestras arquitecturas y las de [Dominguez, 2021] que intentábamos superar no son notables.

5.5.4. Ejemplos

En esta sección se hará un análisis cualitativo de diferentes resultados en la partición de test de MSCOCO. Se analizarán resultados de las mejores arquitecturas entrenadas: $TRAN2LY_{UF}$ y $STRAN2LY_{UF}$.

En la figura 5.11 se ve cómo ambas arquitecturas han conseguido crear el complejo escenario de una cocina. Han conseguido generar tanto frigorífico como horno, mesa, silla y fregadero. Además ambos han conseguido generar el correcto número de personas: 2 (la mujer y el niño). Ambas arquitecturas han generado además a una persona más pequeña que la otra. Es posible que haya sido porque han comprendido que el niño a de ser más pequeño que la mujer.

Por otro lado, el posicionamiento de algunos objetos es cuestionable. $TRAN2LY_{UF}$ ha colocado algunos objetos (dos cuencos y un vaso) en el aire. Aunque podrían estar en alguna balda, tendría más sentido que estuviesen en la mesa. $STRAN2LY_{UF}$ ha colocado una silla, el frigorífico y el fregadero en el mismo sitio. Además ambas arquitecturas han colocado una persona en el aire.

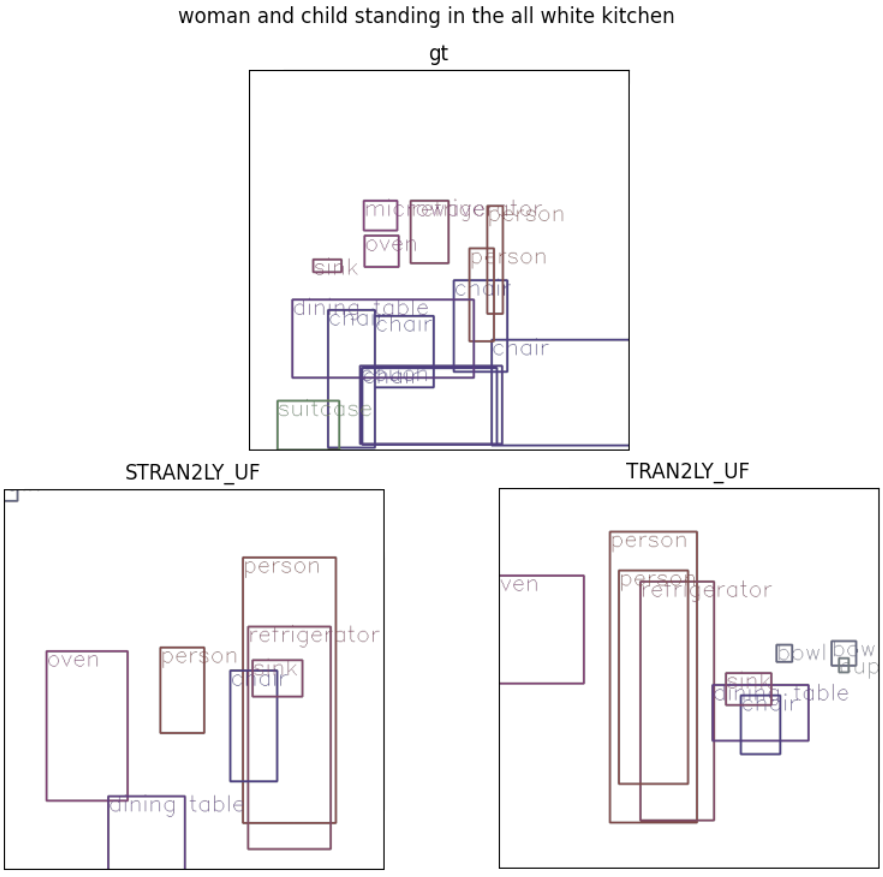

Figura 5.11: Mujer y niño en una cocina blanca. ID de imagen: 141017

En la figura 5.12 ambas arquitecturas consiguen generar todos los objetos del *ground-truth* exceptuando una persona por parte de $TRAN2LY_{UF}$. Ambos consiguen también mantener

el tamaño relativo de los objetos: Las personas son más grandes que los bates y los bates a su vez son más grandes que los guantes y las pelotas de béisbol.

$TRAN2LY_{UF}$ consigue posicionar el bate de béisbol y el guante de béisbol en sitios adecuados: el guante al lado de la persona del fondo y el bate dentro y hacia el costado de la del frente. Sin embargo $TRAN2LY_{UF}$ no consigue generar la tercera persona de la imagen. $STRAN2LY_{UF}$, aunque consigue generar las tres personas, no coloca de forma correcta el bate. Ambos colocan la pelota de béisbol correctamente.

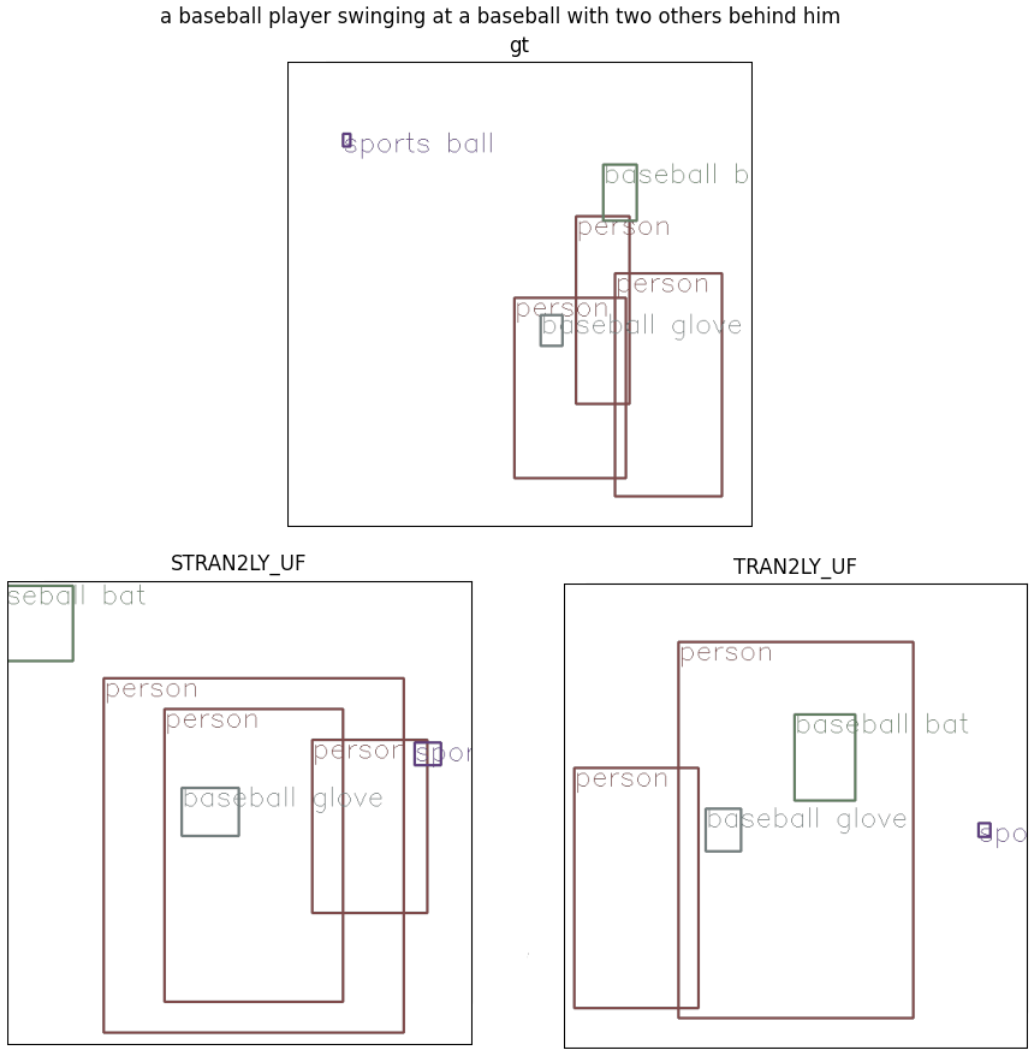

Figura 5.12: Un jugador de béisbol bateando la peloto con otros dos detrás de él.

En la figura 5.13 $STRAN2LY_{UF}$ parece comprender mejor la escena. Coloca al perro dentro del coche y añade un camión de fondo ya que supone que están en una carretera. El resultado de $TRAN2LY_{UF}$ es mucho más simple y no añade todos los objetos de la escena.

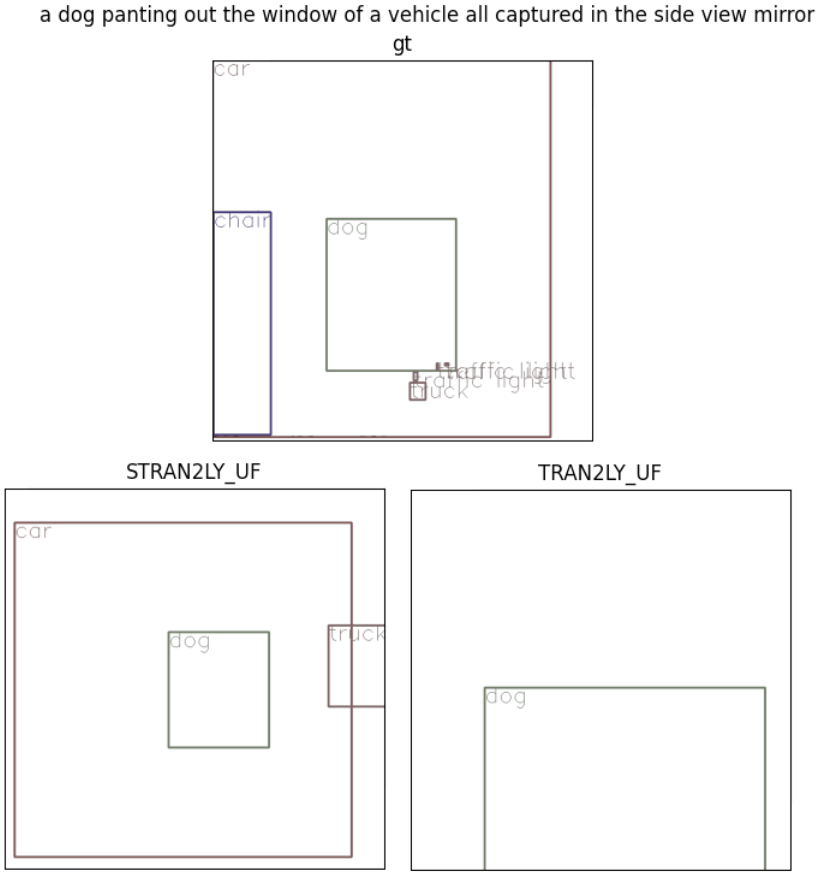

Figura 5.13: Foto de un retrovisor que muestra un perro jadeando fuera de la ventana de un coche.

En la figura 5.14 ambas arquitecturas han conseguido comprender el significado de *kiteboard* y han conseguido añadir una tabla de surf y una cometa aunque no hayan sido mencionados explícitamente. $TRAN2LY_{UF}$ consigue además colocar de forma correcta la cometa y la tabla respecto a la persona (encima y debajo respectivamente). $STRAN2LY_{UF}$, aunque consigue añadir los diferentes elementos de la imagen, coloca la cometa debajo de la persona.

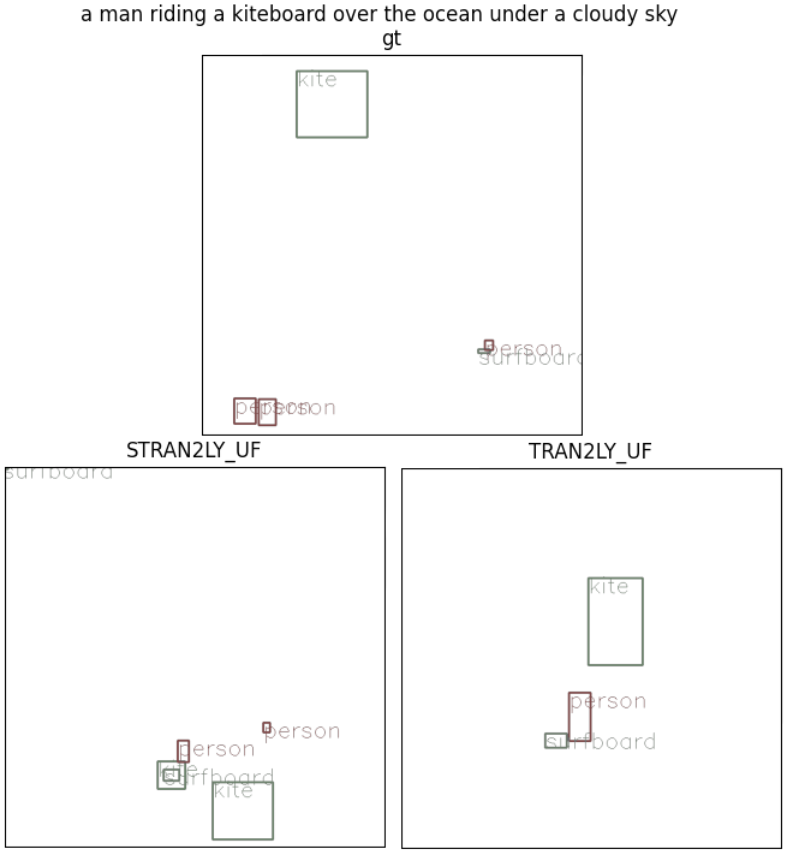

Figura 5.14: un hombre montando un *kiteboard* sobre el océano bajo un cielo nublado

En la figura 5.15 ambas arquitecturas consiguen generar una mesa y (al menos) dos personas. También añaden sillas que tienen sentido en el contexto de la descripción dada. El posicionamiento de las personas de $TRAN2LY_{UF}$ (una encima de otra) no es muy realista. En ese aspecto $STRAN2LY_{UF}$ consigue un mejor resultado. Por otro lado las dos arquitecturas generan objetos adicionales que, aunque no se hayan mencionado explícitamente en la descripción, son adecuados para la situación. $STRAN2LY_{UF}$ añade un vaso en la mesa y $TRAN2LY_{UF}$ añade una botella (además de las sillas mencionadas anteriormente).

Figura 5.15: Un chico y una chica sentados en una mesa de madera con el chico mirando a la chica.

En la figura 5.16 $STRAN2LY_{UF}$ ha conseguido comprender que un rebaño de ovejas se refiere a un grupo grande de ovejas. $TRAN2LY_{UF}$, por otro lado, tan solo ha generado tres ovejas y las ha posicionado todas en el mismo sitio.

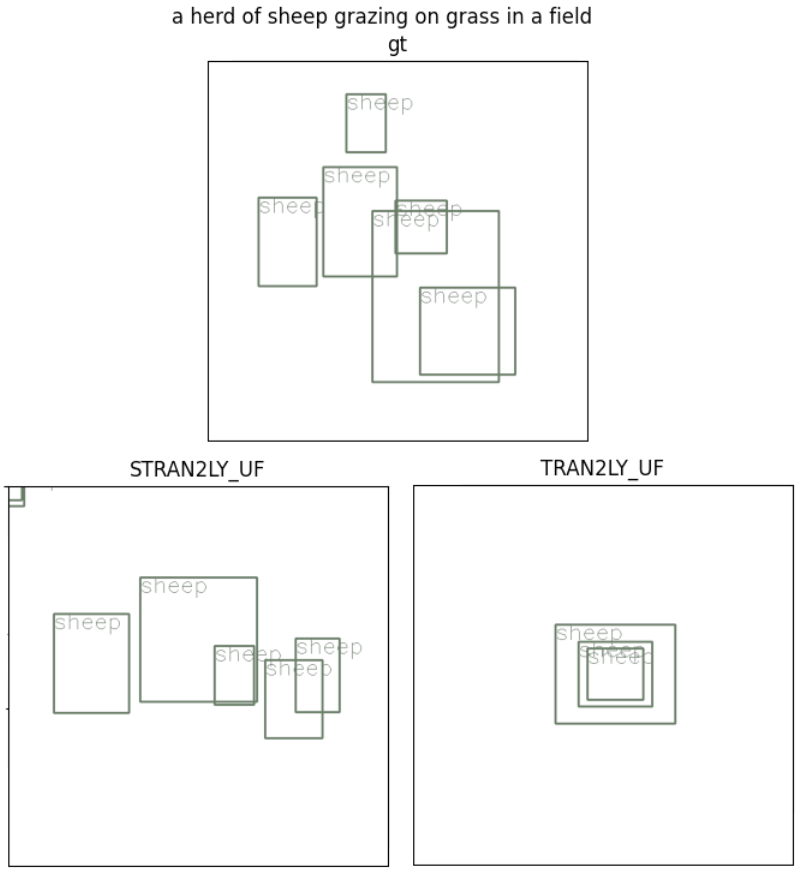

Figura 5.16: Un rebaño de ovejas pastando en el campo.

5.6. Resultados en Spatial CommonSense

Hay que matizar, que los resultados de nuestras arquitecturas y las de [Liu et al., 2022] en 5.3 no son directamente comparables. Las arquitecturas de [Liu et al., 2022] fueron evaluadas en el conjunto de datos completo mientras que las nuestras solo en el subconjunto resultante del pre-procesado mencionado en 4.2.5. Es por ello que los siguientes comentarios sobre el rendimiento de las diferentes arquitecturas se hacen bajo la suposición de que el subconjunto resultante del pre-procesado es representativo del conjunto total.

Como se puede apreciar en 5.3, las arquitecturas entrenadas rivalizan con las evaluadas en

Arquitectura	Tamaño	Altura	Pos. Relativa
Best PLM	54.1	50.8	31.0
VinVL	61.8	64.5	56.1
ISM	72.7	**78.9**	**73.4**
$RNN2LY_F$	0	0	0
$RNN2LY_{UF}$	18.87	4.81	39.42
$TRAN2LY_F$	33.02	27.88	32.69
$TRAN2LY_{UF}$	**78.3**	74.04	58.65
$STRAN2LY_F$	40.57	32.69	28.85
$STRAN2LY_{UF}$	68.87	60.58	41.35
$TRAN2TRAN_F$	0	0	0
$TRAN2TRAN_{UF}$	0	0	0

Tabla 5.3: Precisión de las diferentes arquitecturas en cada subconjunto de *Spatial Commonsense Test*. Los resultados de los modelos Best PLM, VinVL e ISM son los resultados tomados de [Liu et al., 2022]. Son modelos de generación de imágenes así que en su caso, hay que conseguir el resultado y detectar después los objetos en la imagen. Los resultados mostrados son con humanos detectando los objetos y precisión sobre el conjunto de datos completo.

[Liu et al., 2022]. $TRAN2LY_{UF}$ y $STRAN2LY_{UF}$ consiguen resultado similares a los mejores conseguidos por las arquitecturas en [Liu et al., 2022], exceptuando la precisión en el subconjunto de posiciones relativas. En general, la tarea de posiciones relativas es más complicada que el resto. Todas las arquitecturas (excepto *ISM*) tanto en [Liu et al., 2022] como en nuestro proyecto consiguen peores puntuaciones en la tarea de posiciones relativas.

Sin embargo, la dificultad de la tarea no explica completamente la caída en puntuación de nuestras arquitecturas, ya que es algo mayor que la del resto de arquitecturas. Esta diferencia se puede deber al conjunto de entrenamiento utilizado. La mayor parte de descripciones de la tarea de posiciones relativas son acciones que se están tomando: ”Un hombre conduciendo”, ” Una chica montando en bici”, etc. Además, exceptuando los ejemplos de gente andando en bicicleta o jugando al fútbol no hay ejemplos deportivos. No hay ejemplos como ”una persona esperando a batear”. Las descripciones de imágenes de MSCOCO se refieren en su mayoría al escenario y los objetos que contiene. Cuando se menciona la acciones que se está tomando, es en su mayoría en escenas deportivas (y apenas contiene ejemplos de fútbol). Es esta deficiencia la que creemos que hace que nuestras arquitecturas rindan peor en la tarea de posiciones relativas.

Otro patrón que se puede notar en la tabla 5.3 es que la diferencia de resultados entre los mejores modelos del proyecto y los siguientes es mucho mayor que en las métricas de

evaluación. $TRAN2LY_F$, $STRAN2LY_F$ y $RNN2LY_{UF}$, aunque están cerca de $TRAN2LY_{UF}$ y $STRAN2LY_{UF}$ en métricas, aquí hay como mínimo alrededor de 30 % de diferencia entre ellos.

Esta diferencia aumentada se puede deber al *overfitting* de las arquitecturas $TRAN2LY_F$, $STRAN2LY_F$ y $RNN2LY_{UF}$ al conjunto MSCOCO. Es por ello que las métricas conseguidas en el subconjunto desarrollo, con descripciones parecidas al entrenamiento, son tan cercanas a las mejores arquitecturas; mientras que los resultados con otros tipos de descripciones no son tan buenos.

Esta hipótesis se ve reforzada por el hecho de que, al analizar los resultados con mayor detención, observamos que la mayor parte de los fallos de estas tres arquitecturas no se debe al tamaño erróneo de los objetos, sino a la falta de los objetos en sí. Es decir, la arquitectura puede que sepa cuales deberían de ser los tamaños de los objetos referenciados, pero al no saber construir la composición de forma correcta, faltan objetos y baja enormemente la puntuación conseguida.

5.6.1. Ejemplos

En esta sección se hará un análisis cualitativo de diferentes resultados en el conjunto de *Spatial Commonsense Dataset* [Liu et al., 2022]. Se analizarán resultados de las mejores arquitecturas entrenadas: $TRAN2LY_{UF}$ y $STRAN2LY_{UF}$. Se mostrarán resultados de las tres partes del conjunto: tamaño, altura y posición relativa

Tamaño

En este subconjunto se valora que la relación de tamaño entre los objetos de la descripción sea correcta. Es decir, que el área que ocupa el objeto más pequeño sea menor que el objeto mayor. Por ejemplo, entre un pájaro y un camión, el pájaro debería de ocupar menos espacio.

En la figura 5.17 se ve como ambas arquitecturas consiguen generar tanto el vaso como el microondas (aunque $TRAN2LY_{UF}$ coloca el microondas parcialmente fuera de la imagen). La relación de tamaño entre los dos objetos es mantenida. $STRAN2LY_{UF}$ incluso coloca el vaso dentro del microondas. Un resultado excepcional.

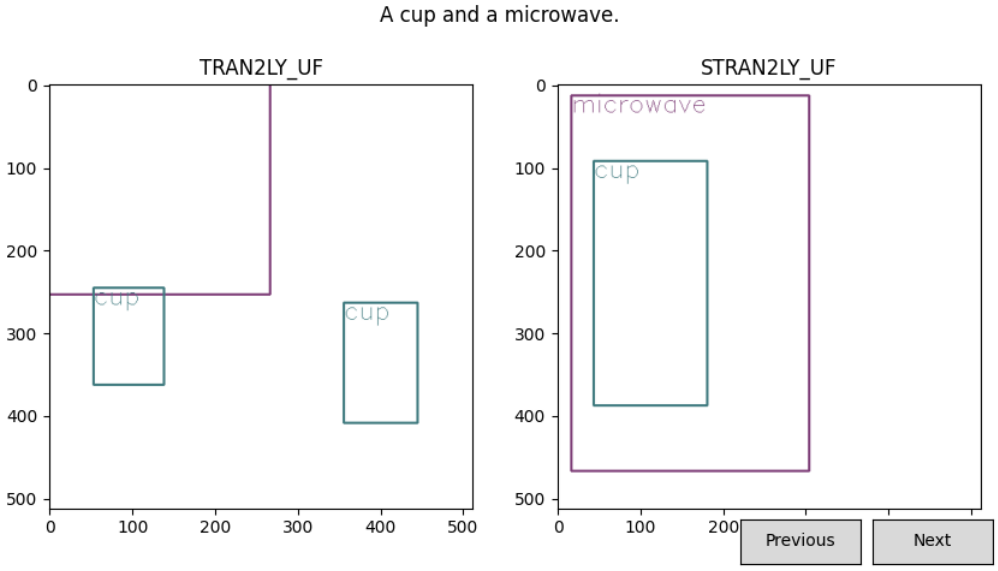

Figura 5.17: Un vaso y un microondas.

En las figuras 5.18 y 5.19 ambas arquitecturas vuelven a generar ambos objetos y mantienen su relación de tamaño. Sin embargo, los tamaños no son tan diferentes como se podría esperar. En la figura 5.19 el pájaro, que claramente tendría que ser muchísimo más pequeño que el avión, tiene la misma altura que el avión. En el caso de la figura 5.18 el tamaño de las botellas se acerca mucho a la del perro.

Por último en la figura 5.20 se les pide generar un pájaro y una silla. Aunque los dos consiguen generarlos y la relación de tamaños se mantiene, ambos generan bastantes objetos adicionales. Especialmente $STRAN2LY_{UF}$, que genera una mesa además de 3 sillas dife-

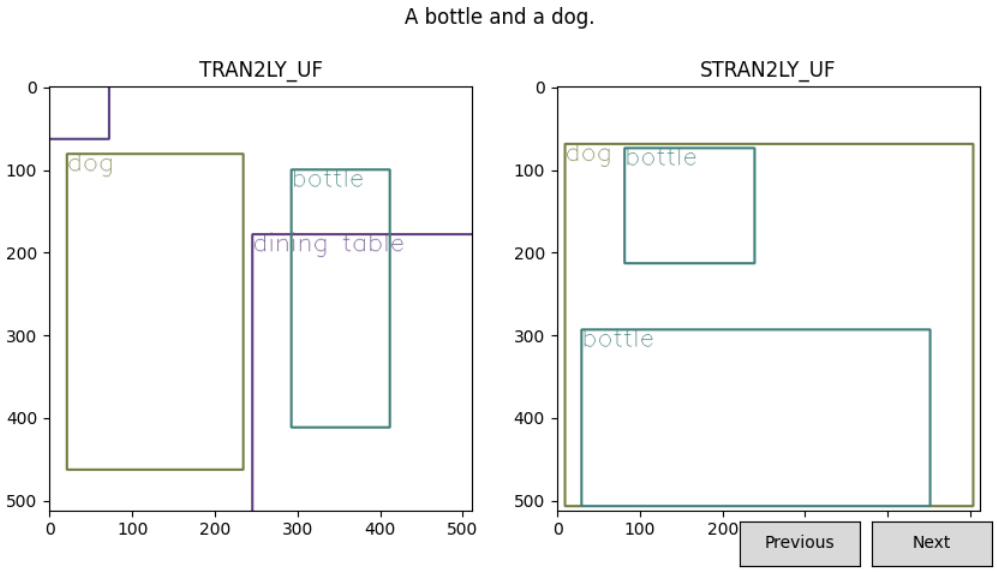

Figura 5.18: Un perro y una botella.

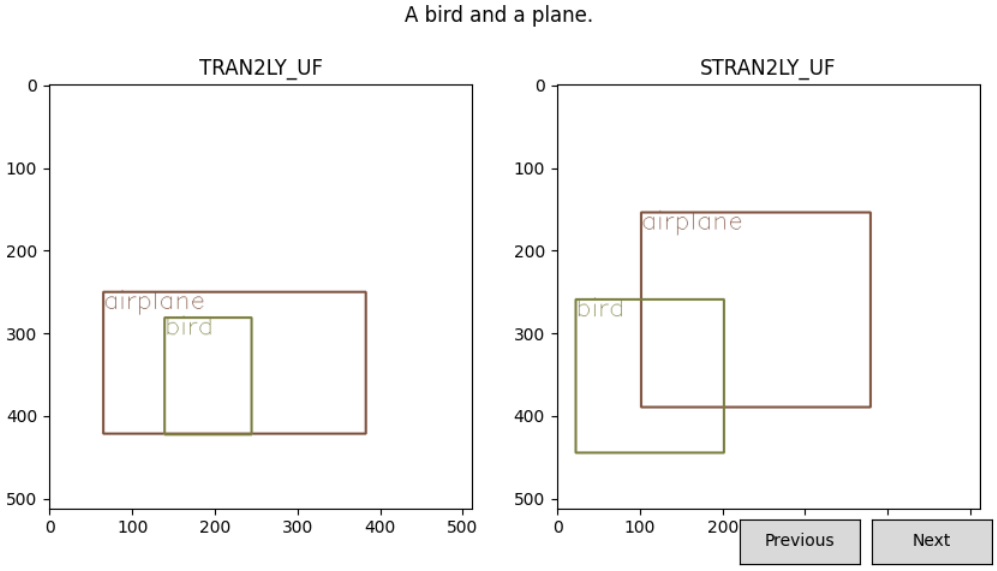

Figura 5.19: Un pájaro y un avión.

rentes. Aunque no es estrictamente erróneo, hace que estas generación den una sensación de menor estabilidad.

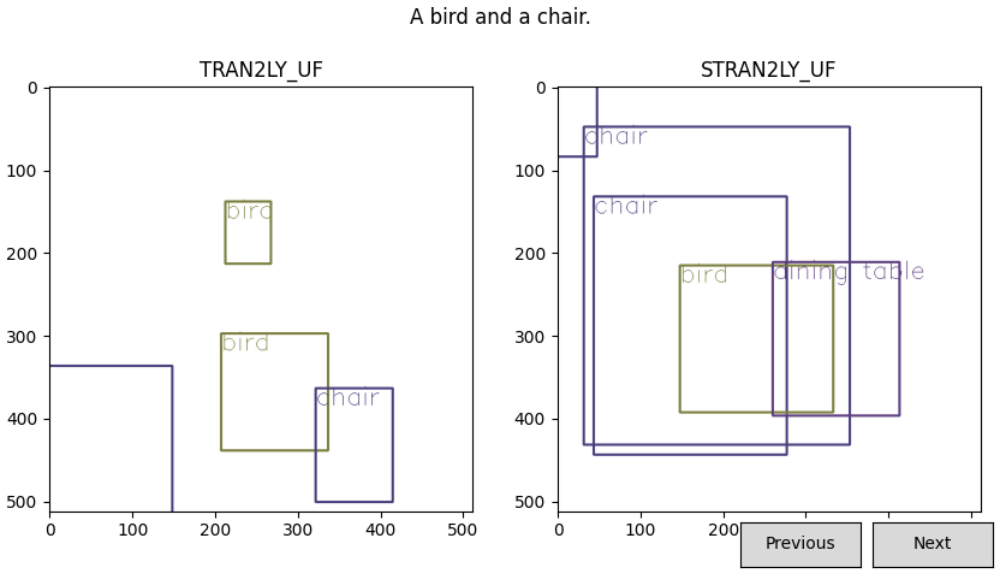

Figura 5.20: Un pájaro y una silla.

En general, el posicionamiento de los objetos es bastante aleatorio. En muchos casos es comprensible que sea así ya que no se le da ninguna dirección respecto a su posición relativa y no se describe la escena en la que se encuentran los objetos. En algunos casos cuando se les pide generar objetos como sillas, mesas u otros objetos de interior las arquitecturas tienden a generar un exceso de objetos y el caos dentro de la imagen es mayor.

Dicho esto, ambas arquitecturas generan ambos objetos de la descripción la gran mayoría de las veces. También mantienen las relaciones de tamaño. Sin embargo cuando la diferencia de tamaños es más grande tienden a igualar un poco las escalas, haciendo que la diferencia no sea tan grande como uno podría esperar.

Altura

En este subconjunto se valora que la relación de altura entre los objetos de la descripción sea correcta. Es decir, que la altura del objeto más bajo sea menor que el objeto más alto. Por ejemplo, entre un humano y una jirafa, la jirafa tendría que ser más alta.

La figura 5.21 es un ejemplo de un buen resultado de parte de las arquitecturas. Las

relaciones tanto de altura como de tamaño de los objetos son respetadas. Además genera cada uno una sola copia de cada objeto. Un resultado muy limpio

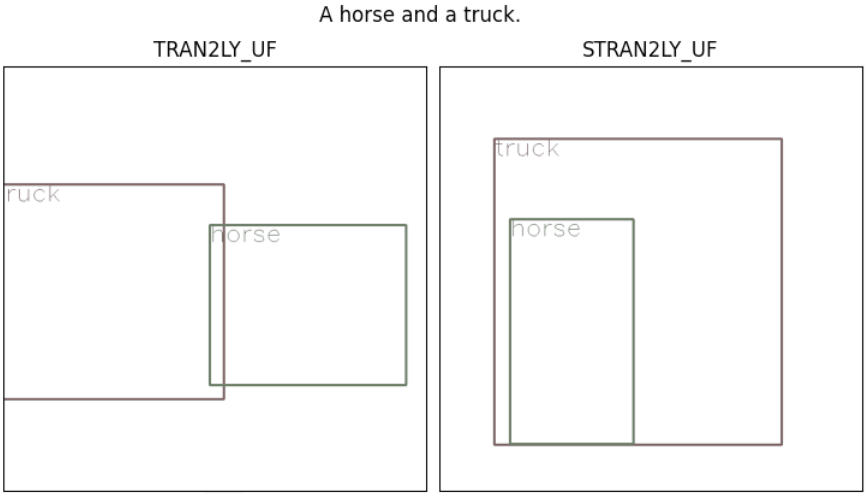

Figura 5.21: Un caballo y un camión.

En las figuras 5.22 es otro ejemplo de las dos arquitecturas generando ambos objetos y manteniendo la relación de altura. Sin embargo, con las diferencias que hay entre las alturas de los objetos (pájaro y jirafa; botella y camión) sería de esperar que la diferencia en las composiciones fuese mayor. Ocurre lo mismo que con el tamaño: las arquitecturas parecen tender a generar ambos objetos principales de tamaños similares (pero mantienen el más pequeño algo menor).

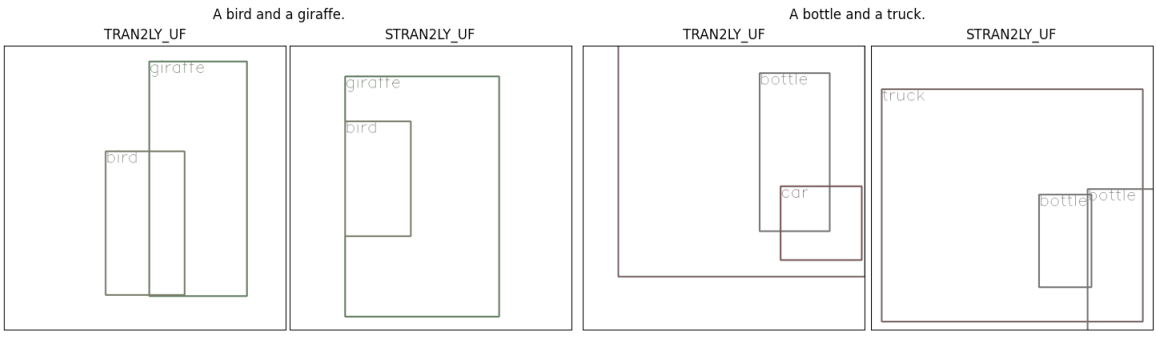

Figura 5.22: ”Un pájaro y una jirafa” (izquierda) y ”Una botella y un camión” (derecha).

En la figura 5.23 se representan los peores tipos de resultado de las arquitecturas. Al pedirles generar objetos de interior (sofás, sillas, mesas) las composiciones generadas son muy caóticas y las alturas peor definidas.

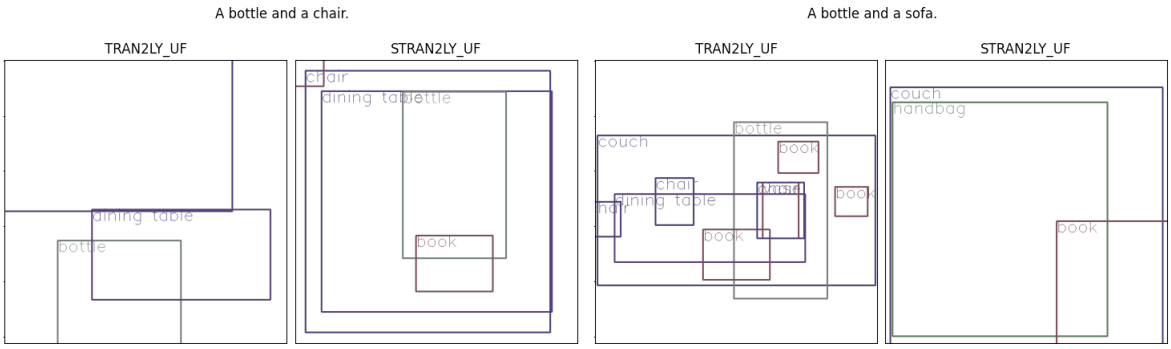

Figura 5.23: ”Una botella y una silla” (izquierda) y ”Una botella y un sofá” (derecha).

Los resultados son muy similares a los de tamaño. Los resultados son generalmente buenos, generando ambos objetos y manteniendo las relaciones de altura. Sin embargo las diferencias de altura hay veces que no es muy significativa y al generar interiores las arquitecturas añaden muchos objetos adicionales.

Posición relativa

En este subconjunto se valora que la relación posicional entre los objetos de la descripción sea correcta. Es decir, que un objeto esté encima de otro, debajo, a su lado o dentro de él. Por ejemplo, si se les da un chico andando en bicicleta, el chico debería de estar encima de la bicicleta.

Cabe también decir que aquí se va a mostrar un ejemplo por cada acción. Por cada acción se han generado diferentes ejemplos. Cada acción tiene una variante con un hombre, una mujer, un niño o una niña haciéndolos. Aunque solo enseñamos uno de ellos, en general los resultados de la misma acción son muy similares independientemente del sexo o edad de la persona que las realiza.

La figura 5.24 es un ejemplo excepcional de ambas arquitecturas. No solo generan ambos objetos. Han entendido la diferencia entre montar un caballo y darle de comer y han generado las posiciones acorde.

La figura 5.25 demuestra como, aunque los objetos en la descripción sean los mismos, el cambio de la acción puede cambiar por completo el rendimiento de las arquitecturas. Aunque ambas arquitecturas generan de forma correcta a personas lavando un coche, $STRAN2LY_{UF}$ no consigue generar correctamente a una persona conduciendo un coche. Siempre genera a la persona más grande que el coche y el coche dentro de la persona

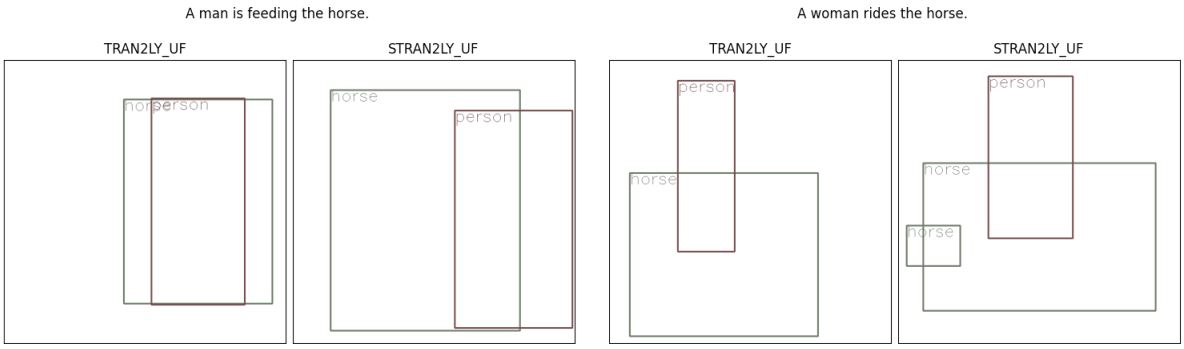

Figura 5.24: "Un hombre alimentando el caballo" (izquierda) y "Una mujer monta el caballo" (derecha).

(cuando debería de ser al contrario). La figura 5.26 es otro ejemplo de este fenómeno pero más extremo. Ninguna de las dos arquitecturas está cerca de conseguir generar personas andando en bicicleta pero si que generan personas reparando bicicletas.

En ambos ejemplos (fig. 5.25 y fig. 5.26) las arquitecturas generan más objetos que en las composiciones de la figura 5.24. Las composiciones son más caóticas, añadiendo frigoríficos en la limpieza de coches (fig. 5.25) y muchas más personas y un autobús junto a las bicicletas (fig. 5.26)

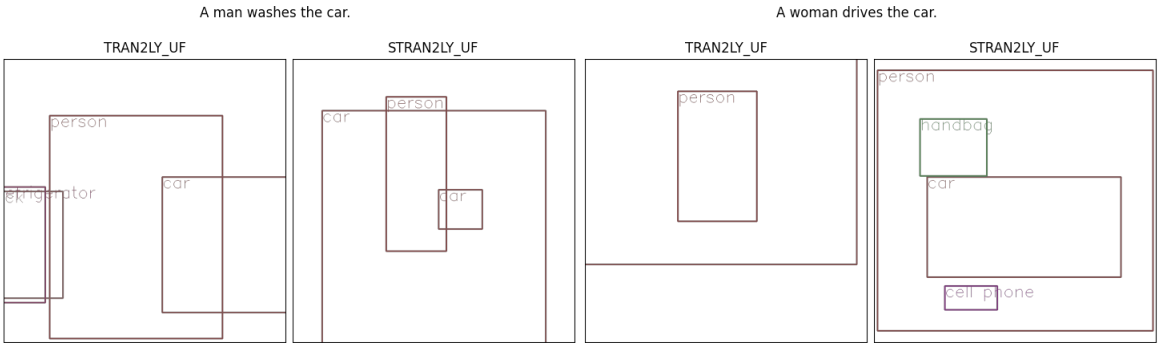

Figura 5.25: "Un hombre limpia el coche" (izquierda) y "Una mujer conduce el coche" (derecha).

Por último en la figura 5.27 se ven los peores resultados de las arquitecturas. Se nota que en el conjunto de entrenamiento no hay muchos ejemplos de fútbol y las arquitecturas no tienen mucha experiencia con el deporte. Cuando se trata de chutar el balón, mantienen la relación de tamaño entre la pelota y las personas. Además a muchas de las composiciones se le puede encontrar sentido y a veces aciertan poniendo la pelota al lado y en la parte baja de la persona principal. También se aumenta significativamente el número de objetos

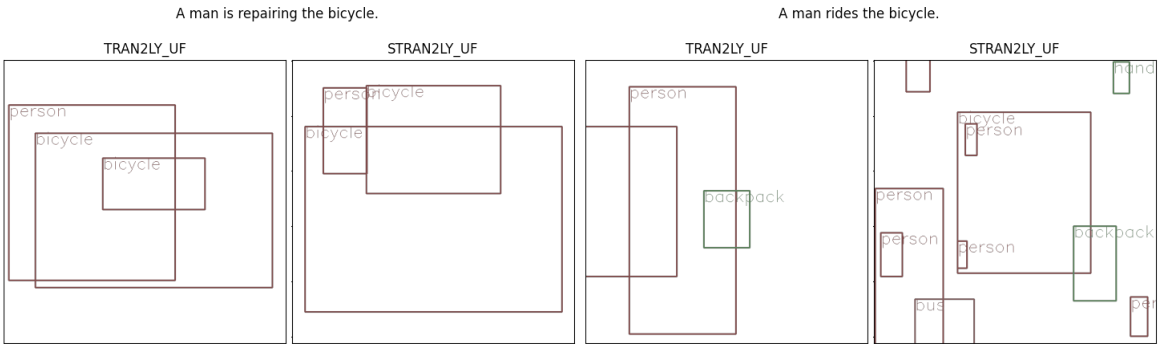

Figura 5.26: "Un hombre reparando una bicicleta" (izquierda) y "Un hombre montando una bicicleta" (derecha).

generados.

Las composiciones sobre darle cabezazos son simplemente erróneas. El número de objetos se dispara y sus posiciones no tienen mucho sentido. Además introducen objetos sin sentido como guantes de béisbol $TRAN2LY_{UF}$ o coches $STRAN2LY_{UF}$. De hecho, es posible que las arquitecturas se hayan encontrado más con el otro significado de la palabra *heading*. En fútbol, *heading* es dar un cabezazo, pero en otros contextos puede significar dirigirse. Por ejemplo, "El hombre se estaba dirigiendo hacia la estación de tren." sería "*The man was heading towards the train station.*".

Los resultados de esta sección del conjunto de datos son mucho más interesantes y variados. El posicionamiento de los objetos tiene un sentido y se puede opinar sobre los resultados más haya de si la arquitectura sabe que un objeto es más pequeño que otro.

La tarea es claramente más difícil y contiene en algunos casos palabras y situaciones a las que no están acostumbradas las arquitecturas. En las anteriores tareas del conjunto las arquitecturas simplemente tenían que conocer la forma de los objetos con los que han trabajado pero en este caso, aunque los objetos sean familiares, las situaciones pueden no serlo.

Ambas arquitecturas parecen tener problemas en las mismas situaciones. Es por ello que nos parece que esta sección depende mucho de la familiaridad de la arquitectura con la situación. Es decir, depende mucho de conjunto de entrenamiento. Dicho esto, sigue habiendo casos en los que las arquitecturas difieren. Por ejemplo en la figura 5.25 $TRAN2LY_{UF}$ consigue generar a personas conduciendo coches pero $STRAN2LY_{UF}$ no. En general $TRAN2LY_{UF}$ suele generar menos objetos que $STRAN2LY_{UF}$ que en este caso hace que las composiciones sean más claras y ordenadas. En general, $TRAN2LY_{UF}$ parece

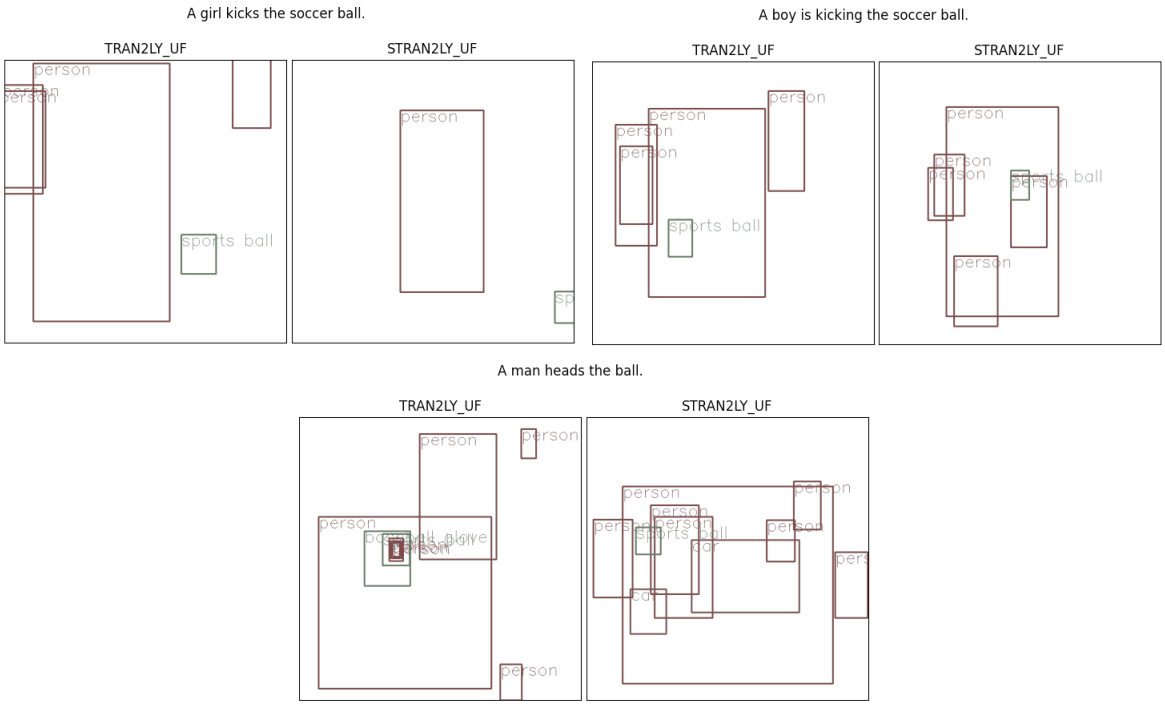

Figura 5.27: ”Una chica chuta el balón de fútbol.” (izquierda) y ”Un chico está chutando el balón de fútbol.” (derecha). ”Un hombre la da un cabezazo al balón.” (debajo)

haber sido algo superior a $STRAN2LY_{UF}$ en esta tarea.

6. CAPÍTULO

Conclusiones y Trabajo Futuro

En este apartado se hablará de las conclusiones del proyecto y se ofrecerán posibles direcciones para próximos proyectos que utilicen este como base.

6.1. Conclusiones

Durante el proyecto se ha conseguido desarrollar tres arquitecturas diferentes para la realización de la tarea *text-to-layout*: *TRAN2LY*, *STRAN2LY* y *TRAN2TRAN*. Estas arquitecturas parten de la base *RNN2LY* tomada de [Dominguez, 2021]. *RNN2LY* es una arquitectura seq2seq formada por dos LSTMs. En este proyecto hemos explorado la posibilidad de sustituir diferentes partes de su arquitectura por modelos transformer.

Las dos primeras arquitecturas desarrolladas *TRAN2LY* y *STRAN2LY* han conseguido resultados satisfactorios. Las dos arquitecturas son el resultado de la sustitución del codificador de *RNN2LY* por un transformer.

En el caso de *TRAN2LY* el codificador es el codificador transformer pre-entrenado llamdo BERT [Devlin et al., 2018]. Este codificador ha sido pre-entrenado para generar representaciones contextualizadas de los tokens de entrada. Como el decodificador requiere de un solo vector por frase en vez de uno por token, la salida del codificador transformer primero ha sido pasada por una capa *max-pooling* antes de introducirse en el decodificador.

En el caso de *STRAN2LY* el codificador tomado es un *sentence-transformer*, que son codificadores transformer normales adaptados para generar un solo vector por frase. La

ventaja respecto al codificador utilizado en *TRAN2LY* es que no solo ha sido adaptado el modelo sino que además ha sido pre-entrenado para conseguir representaciones de frases enteras en vez de de tokens. De forma que tiene un pre-entrenamiento mucho más cercano a la tarea que tiene que realizar.

Por último la arquitectura *TRAN2TRAN* utiliza BERT [Devlin et al., 2018] como codificador y un decodificador transformer como decodificador. Para esta arquitectura se ha tenido que adecuar el decodificador transformer al problema y se ha tenido que experimentar con sus diferentes parámetros. Por desgracia incluso con estos esfuerzos no se han conseguido resultados satisfactorios.

Todas las arquitecturas desarrolladas han sido evaluadas haciendo uso de las métricas propuestas por [Dominguez, 2021]. Se han comparado nuestras arquitecturas tanto con las de [Dominguez, 2021] como con las de ObjGAN [Li et al., 2019]. Los resultados han sido similares a los de [Dominguez, 2021] y superan a los de ObjGAN [Li et al., 2019].

Además de evaluar las arquitecturas con las métricas, se ha utilizado un test diferente a los utilizados en el trabajo de [Dominguez, 2021]: *Spatial Commonsense Test*. Este test pone a prueba el ”sentido común” que han conseguido las arquitecturas durante el entrenamiento. Poniendo a prueba si conocen el tamaño aproximado de diferentes objetos o si saben colocar objetos acorde con la acción que están desempeñando. El test consigue ser sencillo de entender y muy intuitivo. A su vez, es un test que les ha costado realizar a las arquitecturas. En este caso se han vuelto a comparar las arquitecturas desarrolladas con los resultados de diferentes arquitecturas presentados en el artículo de *Spatial Commonsense* [Liu et al., 2022].

6.2. Trabajo Futuro

Por restricciones principalmente de tiempo hay diferentes aspectos que no se han explorado en este proyecto que podría ser interesante explorar en el futuro.

- Revisión de pérdidas. Aunque [Dominguez, 2021] propuso diferentes métricas para comprobar la calidad de los resultados de las arquitecturas, las pérdidas utilizadas no hacen uso de ninguna de estas ideas. Aunque el éxito de ambos proyectos es prueba del funcionamiento de las pérdidas usadas, podría ser interesante hacer versiones derivables de las métricas de [Dominguez, 2021] y entrenar a las arquitecturas con ellas.

Por ejemplo, la métrica de Posición Espacial Relativa Categórica. Esta métrica es, como el nombre indica, categórica. Solo testea si una *bounding-box* está en el mismo lado de la imagen respecto a otro objeto que en el *ground-truth*. En vez de calcular la dirección general en la que están ambos (*ground-truth* y resultado) y mirar a ver si esa dirección general, se podrían comparar los vectores de las direcciones de forma directa. Por ejemplo se podría calcular el producto escalar de los vectores como muestra la figura 6.1.

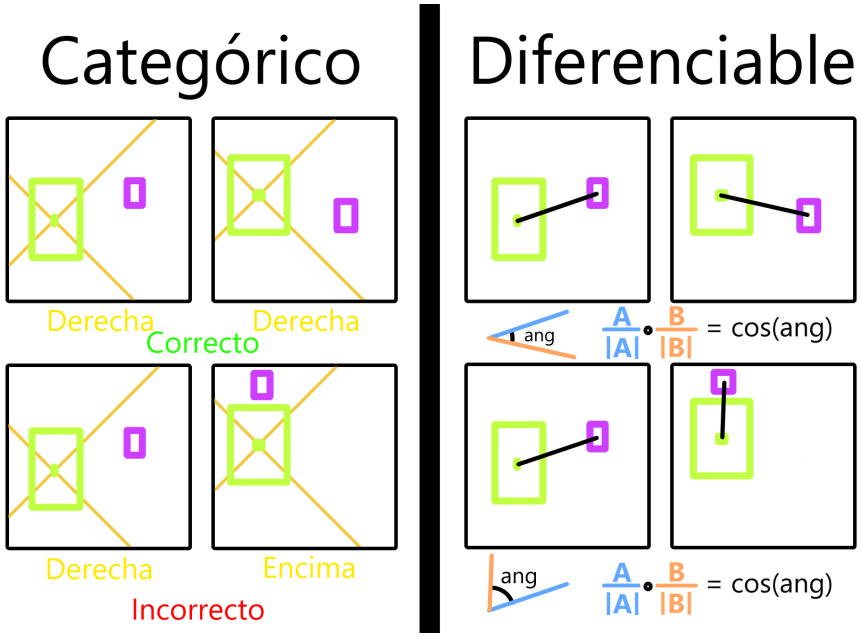

Figura 6.1: Dos ejemplos de *ground-truth* y resultado. Ambos resultados se evaluan con dos métodos: uno categórico (izq.) y uno diferenciable (der.) que se podría utilizar como pérdida.

- Uso del decodificador de DETR [Carion et al., 2020]. DETR es un detector de objetos desarrollado por *Facebook AI*. El decodificador, por lo tanto, es capaz de generar grupos de *bounding-boxes*. El decodificador que utilizan es distinto a un *transformer-decoder* normal. En vez de generar los datos de salida de uno en uno, los genera en paralelo. De esta manera los diferentes objetos se pueden influenciar entre si mientras se generan. De la manera utilizada en este proyecto, una vez generado un objeto influye en los siguientes, pero no puede ser cambiado. Es decir una decisión futura no puede cambiar los objetos pasados.

- Exploración de diferentes conjuntos de datos. MSCOCO es un conjunto de datos abierto utilizado en el entrenamiento de una infinidad de modelos. Sin embargo, en

estos últimos años el número de ejemplos empieza a ser claramente inferior al de otros conjuntos de datos utilizados.

Un conjunto de datos abierto que se ha encontrado durante el proyecto es *Conceptual Captions*[Sharma et al., 2018]. Este contiene 3.3M de imágenes con sus respectivas descripciones en el subconjunto de entrenamiento. Claramente superior a los 73,000 utilizados en este proyecto. El problema de este conjunto es que las descripciones utilizadas son diferentes en naturaleza a las de MSCOCO. Las descripciones de MSCOCO son descripciones generales de la imagen. Sin embargo, las de *Conceptual Captions* son descripciones densas. Estas descripciones mencionan explícitamente todos los detalles que puede sobre la imagen. Por poner un ejemplo, si en MSCOCO tenemos la descripción "Un pueblo pequeño por el que pasa un río", la descripción densa para la misma imagen podría ser "En el centro de la imagen se ve un pueblo rodeado por montañas. A la derecha de la imagen se ve un río que atraviesa el pueblo. En el cielo hay unas pocas nubes blancas.".

Esto hace que entrenar a modelos en este nuevo conjunto de datos cambie el objetivo del proyecto. La tarea ya no sería la misma y no serían comparables modelos que utilizan descripciones normales con los que utilizan descripciones densas. Dicho esto, la tarea utilizando descripciones densas sigue siendo interesante.

- Generación de imágenes. El paso natural después de la tarea *text-to-layout* es la generación de imágenes a partir de las composiciones generadas. La generación de imágenes es una tarea que requiere de muchos más recursos computacionales que la generación de composiciones. Es por ello que su efectuación en un Trabajo de Fin de Grado como este podría ser difícil. Sin embargo, sobre todo con el avance de los modelos de difusión [Dhariwal and Nichol, 2021] en el área, este sigue siendo un proyecto interesante.

Anexos

A. ANEXO

Seguimiento

A.1. Descripción del Proyecto

Este es un proyecto propuesto por el grupo de investigación del lenguaje natural IXA. Los integrantes de IXA Gorka Azkune Galparsoro y Oier López de Lacalle Lecuona son los instructores encargados de orientar y ayudar al alumno durante el proyecto para el correcto desempeño de este. Además, el grupo IXA también ofrece servicios de potencia de cómputo en sus servidores para este proyecto.

La descripción inicial del proyecto es la exploración de arquitecturas para *text-to-layout*. Más específicamente, explorar la viabilidad del uso de modelos *transformers* en arquitecturas de [Dominguez, 2021].Se crearán dos versiones de su arquitectura: una con el codificador cambiado a un codificador *transformer* y otra con tanto el codificador como el decodificador cambiados a un *transformer*. También se permiten revisiones del proyecto de [Dominguez, 2021] según vea necesario el alumno y, una vez desarrollados estas dos arquitecturas, se permite la investigación de opciones diferentes a las del *transformer* común.

Los instructores ayudarán al alumno con el aprendizaje de los diferentes conceptos necesarios para el desarrollo del proyecto. Principalmente, el entendimiento del funcionamiento de *transformers*. También darán ayudarán durante el proceso, ofreciendo consejo y soluciones cuando el alumno se vea atascado y ayudando en el proceso de investigación.

A.2. Objetivos

En este apartado se especifican los objetivos propuestos al principio del proyecto. Al final del apartado se evaluará su cumplimiento.

1. **Comprensión del problema y su resolución.** Se espera conseguir entender el objetivo de la tarea a realizar, la forma y el significado de los datos utilizados para ella y el funcionamiento de arquitecturas propuestas anteriormente para su resolución.

2. **Evaluación de la metodología de [Dominguez, 2021].** Se espera conseguir comprender los diferentes aspectos tanto de su arquitectura, como de su metodología de entrenamiento como de su metodología de evaluación a un nivel suficiente como para ser capaz de evaluar su validez y eficacia. En caso de considerarlo necesario, se espera mejorar estas metodologías.

3. **Comprensión de arquitecturas modernas para *seq2seq*.** El objetivo principal del proyecto es la edición de la arquitectura de [Dominguez, 2021]. Desde el principio la descripción del proyecto se basaba en el reemplazamiento de sus modelos RNN por modelos *transformer*. Más específicamente, implementar una solución donde se reemplace el codificador por un *transformer* dejando el decodificador RNN (TRAN2LY) y otro reemplazando ambos (TRAN2TRAN).También se espera conseguir mejorar el rendimiento de la arquitectura con este cambio.

4. **Definición de nuevas arquitecturas.** Como objetivo adicional, se propone la investigación e implementación de arquitecturas que varíen de los modelos comunes de *transformer*.

5. **Comparación de arquitecturas.** Se debe llegar a comparar las arquitecturas desarrolladas con otras arquitecturas del estado del arte. Es al menos obligatorio la comparación con modelos de [Dominguez, 2021]. También se espera conseguir proponer arquitecturas que rivalicen con el rendimiento del resto de arquitecturas a comparar.

Todos los objetivos propuestos se han cumplido satisfactoriamente con una sola excepción: el objetivo nº4. Era un objetivo ambicioso tener tiempo de investigar y proponer una arquitectura no estándar después de haber implementado otras dos arquitecturas. Se han investigado posibilidades y se ha visto reflejado en la propuesta de trabajo futuro 6.2 DETR pero no ha habido tiempo para su implementación.

Los objetivos de comprensión se han cumplido gracias a la ayuda y orientación de los instructores.

El objetivo de la evaluación de la metodología de [Dominguez, 2021] se ha cumplido de forma ideal. Se ha trabajado con el código de entrenamiento y evaluación completo. Se han discutido diferentes aspectos del entrenamiento como las pérdidas utilizadas tanto en las reuniones como en la memoria en el apartado 6.2. La metodología de evaluación se ha discutido en el apartado 5.3. También se ha ofrecido un nuevo método de evaluación.

El objetivo de la implementación de versiones de las arquitecturas de [Dominguez, 2021] utilizando *transformers* se ha cumplido. Se han implementado las dos arquitecturas propuestas. Además, se ha implementado también una versión alternativa de $TRAN2LY$: $STRAN2LY$. Sin embargo, la solución $TRAN2TRAN$, aunque ha sido completamente implementada, no ha conseguido los resultados deseados.

A.3. Restricciones

- **Potencia de Cómputo**: Solo se podrá utilizar la potencia de cómputo y memoria que posea el alumno o que pueda proporcionar el grupo IXA.

- **Conjuntos de Datos**: Solo se podrán utilizar conjuntos de datos públicamente disponibles, posicionándonos en desventaja respecto a algunos modelos del estado del arte. Aunque técnicamente se podría generar un conjunto de datos propio, el etiquetado correcto de miles de imágenes con sus objetos no es una tarea realista para este proyecto.

- **Tiempo Disponible**: Las 300 horas disponibles en este proyecto pueden no ser suficientes para el cumplimiento de todos los objetivos. En caso de que fuera necesario eliminar alguno de los objetivos la modernización de los modelos se mantendrá como objetivo principal. El revisado del proyecto de Carlos se mantendrá como objetivo secundario ya que su proyecto ya es prueba de su funcionamiento, de forma que cualquier nuevo cambio sería una mejora pero no una necesidad.

A.4. Plan de trabajo

En este apartado se mostrarán y explicarán todos los paquetes de trabajo. Tanto los previstos inicialmente como los realmente realizados. También se marcarán de forma diferente los previstos y cumplidos y los previstos no cumplidos así como los no previstos realizados.

La figura A.1 muestra los diferentes paquetes de trabajo identificados durante el proyecto. Muestra todos los paquetes planeados (tanto si se han realizado como si no) y todos los realizados (aunque no estuviesen inicialmente planeados).

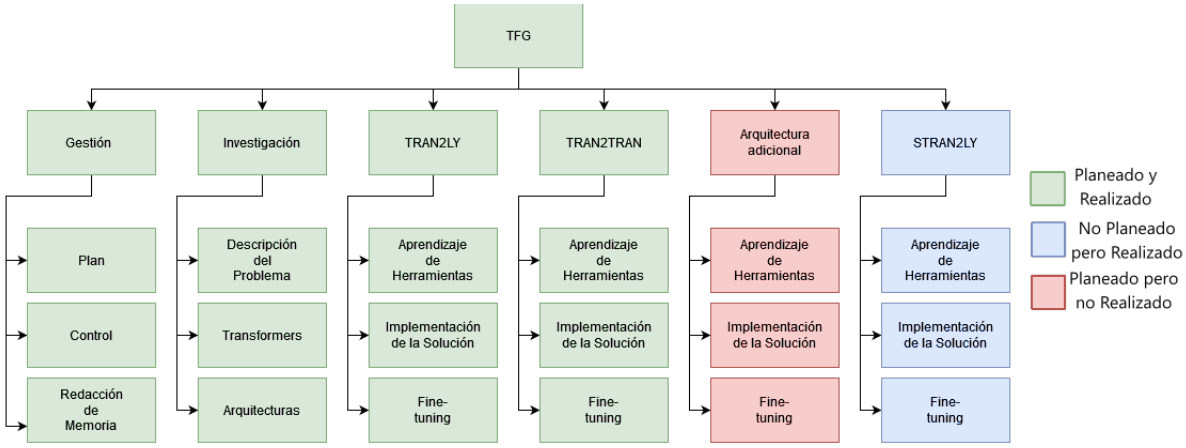

Figura A.1: Paquetes de trabajo identificados durante el trabajo. En verde los paquetes que estaban planeados desde un inicio y se han realizados. En rojo los paquetes planeados inicialmente que no se han realizado. En azul los paquetes que no estaban planificados pero se han realizado.

A continuación se explicará más a fondo cada paquete:

- *Gestión*.
  - *Plan*. Aproximar el tiempo necesario para realizar las diferentes tareas, decidir en qué momento se van a hacer y en qué orden.
  - *Control*. Llevar la cuenta del tiempo utilizado en cada tarea y tomar las decisiones indicadas dependiendo de ello.
  - *Redacción de memoria*. Incluye la recolección y visualización de resultados finales.
- *Investigación*.

- *Descripción del problema*. Comprender la descripción del problema y sus posibles representaciones: los datos de entrada, de salida, y el tipo de dato intermedio necesario en las arquitecturas más comunes.
- *Transformers*. Comprender el funcionamiento de los modelos *Transformer*, tanto de los codificadores como los decodificadores.
- *Arquitecturas*. Investigar las diferentes arquitecturas utilizadas en el estado del arte para la resolución del problema. Incluye el aprendizaje del funcionamiento de las arquitecturas de [Dominguez, 2021] y el descubrimiento de modelos menos comunes para la arquitectura *Arquitectura adicional*.

- *TRAN2LY*
  - *Aprendizaje de herramientas*. Aprendizaje del funcionamiento de las librerías que implementan las diferentes partes de la arquitectura o son necesarios para implementar las diferentes partes de la arquitectura. En este caso, encontrar librerías que implementen un *transformer-encoder* y aprender a usarlas. Sería beneficioso si también puede cargar diferentes modelos ya pre-entrenados.
  - *Implementación de la solución*. Incluye probar el correcto funcionamiento de la implementación.
  - *Fine-tuning*. Óptimización de los parámetros de la arquitectura basándose en resultados en *development*.
- *TRAN2TRAN*
  - *Aprendizaje de herramientas*. Aprendizaje del funcionamiento de las librerías que implementan las diferentes partes de la arquitectura o son necesarios para implementar las diferentes partes de la arquitectura. En este caso, encontrar librerías que implementen un *transformer-decoder* y aprender a usarlas.
  - *Implementación de la solución*. Incluye probar el correcto funcionamiento de la implementación.
  - *Fine-tuning*. Óptimización de los parámetros de la arquitectura basándose en resultados en *development*.
- *Arquitectura adicional*
  - *Aprendizaje de herramientas*. Aprendizaje del funcionamiento de las librerías que implementan o son necesarios para implementar las diferentes partes de la arquitectura.

- *Implementación de la solución*. Incluye probar el correcto funcionamiento de la implementación.
- *Fine-tuning*. Óptimización de los parámetros de la arquitectura basándose en resultados en *development*.

- *STRAN2LY*
  - *Aprendizaje de herramientas*. Aprendizaje del funcionamiento de las librerías que implementan las diferentes partes de la arquitectura o son necesarios para implementar las diferentes partes de la arquitectura. En este caso, encontrar librerías que implementen un *sentence-transformer* y aprender a usarlas. Sería beneficioso si también puede cargar diferentes modelos ya pre-entrenados.
  - *Implementación de la solución*. Incluye probar el correcto funcionamiento de la implementación.
  - *Fine-tuning*. Óptimización de los parámetros de la arquitectura basándose en resultados en *development*.

Después de planear los diferentes paquetes de trabajo, se realizó una aproximación del tiempo que se utilizaría en cada uno. También se marcó las fechas aproximadas en las que se realizaría cada tarea.

En la figura A.2 se muestran las fechas en las que se planearon realizar cada paquete. La figura A.3 muestra las fechas en las que se han realizado. Se puede ver que el paquete inesperado de STRAN2LY ha retrasado el resto de paquetes. Además de atrasado, el paquete TRAN2TRAN ha tomado más tiempo del esperado. Estos dos hechos han hecho que no se pueda implementar ninguna arquitectura con modelos más exóticos y ha atrasado la entrega del proyecto de Junio hasta Septiembre.

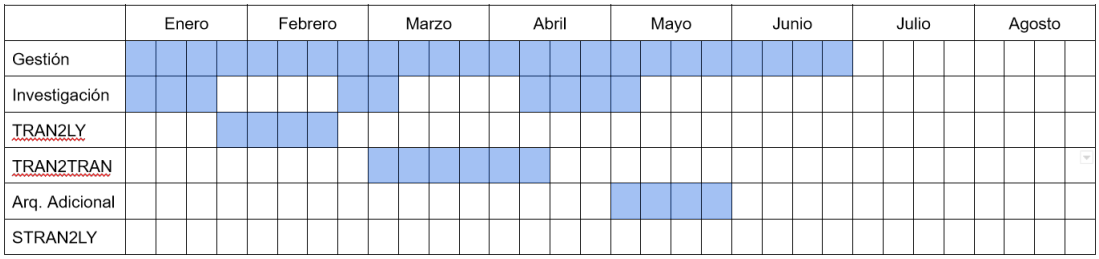

Figura A.2: Fechas en las que se planeó realizar cada paquete de trabajo.

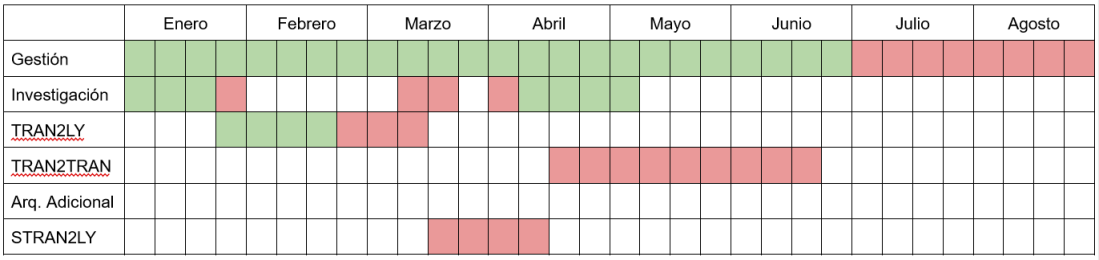

Figura A.3: Fechas en las que se han realizado los diferentes paquetes de trabajo. En verde las fechas que coinciden con el plan inicial y en rojo las que no.

Las horas invertidas en cada paquete se muestran en la tabla A.1. Como se puede ver, el mayor desvío está en TRAN2TRAN qeu ha presentado más problemas de los esperados.

	Plan	Realidad
Gestión	80	100
Investigación	50	43
TRAN2LY	40	48
TRAN2TRAN	40	100
STRAN2LY	0	23
Arq. Adicional	90	0
	300	314

Tabla A.1: Horas planeadas y horas utilizadas en cada paquete de trabajo.

A.5. Metodología

La metodología de trabajo a utilizar ha sido discutida al inicio del proyecto por todas las personas afectadas (Gorka, Oier y el alumno, Eneko).

A.5.1. Entorno de trabajo

El trabajo se realizará en remoto tanto por parte de los instructores como del alumno. En el caso del alumno, se hará desde su casa.

Ni las reuniones ni ninguna otra tarea requiere de la presencia de ninguno relacionado en ningún sitio en específico. Tan solo requiere de un ordenador para el desarrollo y de

conexión a internet en el caso del uso de los servidores de IXA y de las reuniones en remoto.

A.5.2. Comunicación

La mayor parte de la comunicación se espera realizarla en las reuniones semanales. Para estas reuniones en remoto se utilizará *Google meets* ya que permite compartir la llamada de forma sencilla y también permite mantener el mismo enlace de entrada a la llamada durante tiempo indefinido. De esta manera se mantiene el mismo enlace de entrada a las reuniones durante el proyecto completo. En estas reuniones se realizará el control del progreso del proyecto así como la planificación para la próxima semana y las reflexiones sobre la investigación y resultados hasta el momento.

La comunicación asíncrona se realizará a través de correo electrónico. Se utilizará para la organización de las reuniones y para comunicaciones puntuales.

No se ha establecido ningún canal de comunicación síncrona permanente ya que no es necesario este tipo de comunicación. El *feedback* semanal es suficiente para la mayoría de problemas y el tiempo de respuesta normal de los integrantes a correos electrónicos es suficiente para el resto de casos puntuales.

A.5.3. Horarios de trabajo y reuniones

El horario de desarrollo es flexible según las necesidades del alumno. En la planificación semanal que se lleva a cabo en las reuniones, se tendrá en cuenta la situación del alumno esa semana para la correcta asignación de objetivos acorde con el tiempo que le puede dedicar al proyecto.

Las reuniones se realizarán cada lunes a las once de la mañana. En caso de que algún integrante no pueda acudir a ella, se avisará en la reunión anterior o por correo electrónico con una antelación mayor a un día. En casos que se considere más apropiado o sea necesario, la fecha y hora de reuniones se puede cambiar tras discusión y aprobación del equipo entero.

A.5.4. Provisión de potencia de cómputo

El alumno tendrá que seguir el método interno de reserva de GPUs. Dependiendo del servidor, estos tienen o un sistema de cola automático o un documento visible para los usuarios donde se pueden reservar GPUs específicas manualmente en caso de que nadie más la esté usando en ese momento. Además, en ambas opciones hay que respetar el nivel de prioridad dado al proyecto por el grupo. Siendo un proyecto de fin de grado, la prioridad es mínima: inferior a proyectos de investigación del propio grupo y de trabajos de doctorado.

A.6. Riesgos

En este apartado se presentarán los diferentes factores de riesgo identificados al inicio del proyecto junto a las diferentes acciones de mitigación posibles por cada uno.

- **Capacidades de cómputo**. Los modelos *transformer* han dominado el campo del *deep-learning* durante los últimos años. Este nivel de rendimiento se debe en parte a la capacidad para escalar estos modelos. Es por ello que los modelos *transformer* en general consumen más memoria que modelos como los RNN o LSTM. Es por ello que el riesgo de no poseer suficientes recursos como para entrenarlo es mayor.

  En caso de que el alumno no tenga suficiente potencia de cómputo en su ordenador personal, pedirá potencia prestada de los servidores de IXA. Los servidores de IXA tienen potencia más que suficiente para entrenar los modelos relativamente sencillos que se utilizarán en este proyecto. Sin embargo, al utilizar estos servicios el riesgo pasa a ser no tener suficiente acceso a los servidores como para realizar todos los entrenamientos y pruebas que se desearían. No poner realizar entrenamientos por estar a la espera de poder usar los servidores de IXA podría suponer retrasos en la investigación y desarrollo. En caso de que este sea un problema grave, habrá que investigar otras soluciones como por ejemplo intentar utilizar más de un ordenador personal (con ayuda de conocidos) para realizar los entrenamientos.

- **Naturaleza de la investigación**. Cuando cualquier proyecto incluye un apartado de investigación, siempre existe el riesgo de no encontrar el tipo de contenido que se buscaba.

En el caso de las arquitecturas más sencillas (reemplazar el codificador por un *transformer* y después el codificador y el decodificador por un *transformer*) el riesgo es inexistente. Existen muchos recursos donde explican el funcionamiento de este tipo de modelos y cómo ajustarlos a diferentes tareas. Es por ello que el riesgo de no encontrar ejemplos de arquitecturas *text-to-layout* que utilicen *transformers* o recursos que expliquen cómo adecuar el *transformer* a la tarea es mínimo o inexistente.

El problema será cuando llegue la investigación de arquitecturas más exóticas. Cuando se acaben de desarrollar las arquitecturas sencillas, se buscarán alternativas al *transformer* tradicional. Puede que simplemente no se encuentren arquitecturas exóticas que sirvan para el proyecto.

- **Comprensión del proyecto anterior**. Trabajar con código ajeno siempre es más difícil que con el propio.

  En caso de que el alumno se vea atascado intentando comprender o adecuar el código de [Dominguez, 2021], se intentará contactar con el autor de [Dominguez, 2021] para pedir ayuda. En casos extremos se tendrá que aceptar el hecho de que habrá que hacer una inversión de tiempo en reimplementar la parte de código que de problemas. Hay una enorme cantidad de tutoriales en linea explicando diferentes partes de la implementación del entrenamiento de este tipo de modelos. Es por ello que aunque se llegue a este extremo no se espera que este contratiempo hunda el proyecto entero.

- **Inexperiencia en proyectos del mismo tipo**. El alumno no tiene experiencia extensa implementando modelos de aprendizaje profundo. Por ello, puede que se haya subestimado en el plan el tiempo necesario para el aprendizaje de las herramientas de implementación.

  Conociendo este punto débil, el alumno estará desde un principio atento a cualquier contratiempo que pueda llevar a un atasco. Como respuesta a esta situación, el alumno intentará pedir ayuda tanto a los instructores, a compañeros o en linea lo antes posible. En el peor caso, si las anteriores medidas no fuesen suficientes, esto podría requerir volver a planear el proyecto y rebajar o recortar objetivos no principales.

B. ANEXO

Optimizaciones

Se indican en este apartado las modificaciones cuyo único objetivo es acelerar la ejecución del código. El conjunto de optimizaciones realizados aceleran el proceso de entrenamiento hasta un 40 %. Una cantidad impresionante teniendo en cuenta que las optimizaciones no son al cálculo realizado por los modelos en sí sino a diferentes procesos necesarios para el cálculo de los *loss'* y el pre-procesado de los datos para su uso en los modelos.

B.1. Conversión de Coordenadas a *one-hot encoding*

Ésta parte del código se encarga de convertir pares de coordenadas x,y (ambas entre 0 y 1) a un *one-hot encoding*[1] donde el único valor positivo es el índice [0,$xy\_distrivution\_size^2$) que representa la casilla de las coordenadas en la rejilla de salida.

La nueva solución es 225 veces más rápida que la anterior. 10000 ejecuciones con grupos de 32 pares de coordenadas pasan de tardar 103.46 segundos a tan solo 0.46 (el tiempo exacto varia de ordenador a ordenador pero la diferencia relativa debería mantenerse).

Anterior solución:

La anterior solución es la función *convert_from_coordinates*: primero convierte las coordenadas de [0,1] a [0,*xy_distribution_size*), después inicializa una matriz con las mismas dimensiones que la rejilla con ceros, utilizando las coordenadas [0,*xy_distribution_size*)

[1] Vector poblado por ceros en todas las posiciones menos la posición de interés, que contiene un uno. En este caso la posición de interés es la casilla donde se encuentra la coordenada.

pone la casilla indicada a 1 y por último aplana esta matriz para conseguir el *one-hot encoding*. Esta última parte de la generación de vectores se hace dentro de un bucle for (ya que la función admite más de un par como entrada).

Nueva solución:

La anterior solución realmente no realiza ningún cálculo excesivo. Sin embargo, su rendimiento se puede mejorar utilizando herramientas de la librería numpy y pytorch para hacer los cálculos en C y en grupo en vez de utilizando bucles for de python.

El resultado se calcula además en un vector plano directamente en vez de generar uno de dos dimensiones para después aplanarlo. Esto requiere del cálculo del índice final en el vector plano en vez de el índice de las coordenadas x e y por separado. Esto se consigue sencillamente multiplicando el resultado del índice de *y* por *xy_distribution_size* y sumándole al resultado el índice de *x*.

Este cambio también permite separar la función en dos trozos: uno que devuelve el índice donde debería de estar la coordenada en la rejilla y otra que, llamando a la primera, genera *one-hot encodings*. En algunas partes del código no se precisa de un *one-hot encoding* sino tan solo del índice donde iría el valor 1 en dicho *encoding*. De esta manera, en esos casos, se ahorra la inicialización de vectores en memoria.

B.2. Redondeo de Coordenadas a la Rejilla de Salida

Las coordenadas apuntadas en el conjunto de datos pueden (evidentemente) estar en cualquier punto de la imagen. No se alinean con la rejilla a la que nosotros limitamos nuestra salida. Es por ello que hay que redondearlos a valores que se alineen con la rejilla.

La nueva solución es 752 veces más rápido que la anterior. 1000 ejecuciones con grupos de 32 pares de coordenadas pasan de tardar 60.19 segundos a tan solo 0.08 (el tiempo exacto varia de ordenador a ordenador pero la diferencia relativa debería mantenerse).

Anterior solución:

Esta conversión se tiene que hacer con varios pares a la vez. En la anterior solución, se va haciendo cada conversión de uno en uno dentro de un bucle for a través de los pares de coordenadas. Cada par de coordenadas se pasa después a través de dos funciones: *convert_from_coordinates* y *convert_to_coordinates*. *convert_from_coordinates* genera un *one-hot encoding* y *convert_to_coordinates* necesita un índice (no el vector

entero) así que para encontrar donde está el valor 1 del *one-hot encoding*, se ejecuta un *argmax*.

convert_from_coordinates es la función descrita en el anterior apartado que convierte de un par de coordenadas x,y a un *one-hot encoding* de su posición en la rejilla de salida.

convert_to_coordinates convierte de un índice entre 0 y $xy\_distribution\_size^2$ representando la posición en la rejilla del punto a un par de coordenadas entre 0 y 1. La altura del punto en la imagen se consigue dividiendo el índice entre $xy\_distribution\_size$ y la anchura con el resto de la misma división.

Nueva solución:

Los valores entre 0 y 1 se pasan a valores entre 0 y $xy\_distribution\_size$ simplemente multiplicándolos por $xy\_distribution\_size$. Estos valores se redondean hacia abajo. Así conseguimos las coordenadas en términos de la casilla en la que se encuentra la coordenada. El único problema es que el resultado del cálculo se encuentra en [0,$xy\_distribution\_size$][2] y queremos que esté [0,$xy\_distribution\_size - 1$]. Simplemente se fuerza que el resultado esté en [0,$xy\_distribution\_size - 1$] con un clamp que en la práctica lo único que hace es pasar los resultados de $xy\_distribution\_size$ a $xy\_distribution\_size - 1$. Por último el resultado se pasa de vuelta a [0,1) dividiendo el resultado en [0,$xy\_distribution\_size$-1] por $xy\_distribution\_size$.

Por ejemplo, con $xy\_distribution\_size = 32$, para conseguir el resultado de $0{,}23658 \rightarrow 7/32 = 0{,}21875$ se hace:

$0{,}23658 \cdot 32 = 7{,}57056 \rightarrow$ redondear y clamp $\rightarrow 7 \rightarrow 7/32$

Todos estos cálculos se realizan utilizando diferentes herramientas de pytorch para hacerlos al grupo entero de coordenadas a la vez. Claramente más rápidos que hacerlo de uno en uno dentro de un bucle for.

[2] El resultado de $xy\_distribution\_size$ en este cálculo solo ocurre cuando la coordenada siendo convertida es exactamente 1. Esto en un principio no ocurre porque significaría que el centro del *bounding-box* está en el límite de la imagen

C. ANEXO

Fine-Tuning Process

Comentar el tema de los diferentes tipos de pooling y pruebas que se han hecho con diferentes parámetros.

En este apartado se mencionarán los diferentes experimentos realizados con los parámetros de cada arquitectura.

C.1. TRAN2LY

Los esfuerzos en experimentación con esta arquitectura se han centrado en dos parámetros: el tipo de *pooling* y la temperatura utilizada en el *temperature softmax*.

C.1.1. *Pooling*

Como se ha explicado en el apartado 3.2, la salida del codificador de esta arquitectura es un grupo de n vectores. n siendo el número de tokens dados como entrada. Sin embargo, el decodificador requiere de un solo vector de tamaño fijo como entrada. Es por ello que es necesario fusionar los n vectores de salida a uno solo. A este procceso se le llama *pooling*.

El método elegido ha sido *max-pooling*. Dados n vectores de k elementos, genera un único vector de k elementos. Los k elementos del resultado son calculados tomando el mayor elemento con el mismo índice de los n vectores a fusionar C.1.

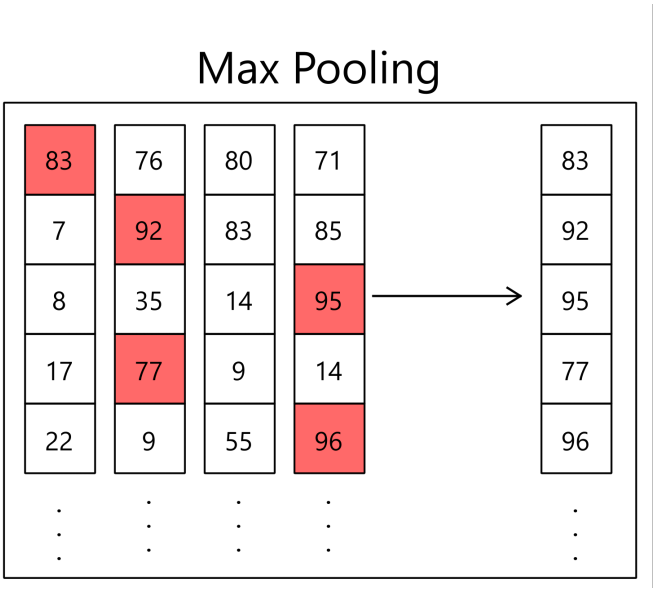

Figura C.1: Ejemplo del proceso de *max-pooling* fusionando cuatro vectores

Pero éste no es el único método. Además de *max-pooling*, uno podría también hacer *average-pooling*. Es decir, en vez de tomar el máximo de entre los vectores a fusionar, hacer una medie entre los valores de los vectores a fusionar C.2.

Por último, el tokenizador que utiliza BERT genera un token llamado CLS al inicio de cada frase. Diferentes artículos [Devlin et al., 2018] han utilizado el vector final de este token como representación para la frase entera. Tras algo de entrenamiento, el vector resultante se podría utilizar como vector fijo representante de la frase completa.

Después de experimentar con las tres opciones hemos encontrado tanto con resultados cuantitativos como cualitativos que el método de *max-pooling* es superior a los otros dos.

C.1.2. Temperatura

El algoritmo de *temperature softmax* es increíblemente similar a *softmax* pero añade un parámetro que permite mayor control de la forma del resultado.

El resultado con mayor valor sigue teniendo el mayor valor después del *temperature softmax*. Sin embargo, nuestro método de selección es aleatorio. En vez de simplemente tomar el resultado con mayor valor, se toman los valores resultantes del *temperature softmax* y se utilizan como la probabilidad de cada resultado de ser escogido de forma aleatoria.

Temperaturas menores a 1 hacen que los valores más altos aumenten (en relación al resto)

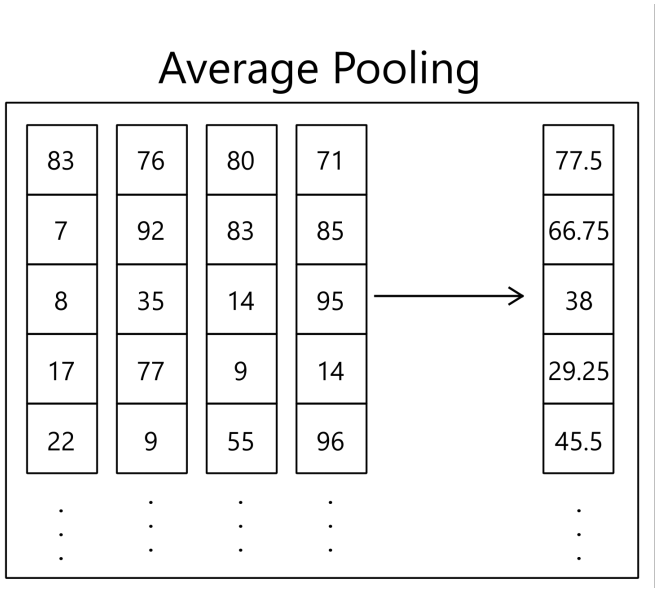

Figura C.2: Ejemplo del proceso de *average-pooling* fusionando cuatro vectores

aún más. La clave está en conseguir que un subconjunto lo suficientemente pequeño sea el que esté claramente separado del resto de opciones pero a su vez que no sea demasiado pequeño. Es decir, queremos que no solo un resultado sea claramente mayor al resto: queremos diversidad en los resultados. Pero también queremos que haya una clara diferencia entre el grupo de resultados mejores y los peores.

Para ello [Dominguez, 2021] eligió una temperatura de 0.4 siguiendo el ejemplo de [Li et al., 2019]. Al empezar el proyecto y cambiar el codificador, nos dimos cuenta de que los resultados que estábamos consiguiendo tenían valores post-*temperature softmax* mucho mayores que los que conseguían las arquitecturas de [Dominguez, 2021]. Es por ello que probamos con diferentes valores de temperatura para ajustar estos valores.

Sin embargo, después de analizar los resultados de la arquitectura utilizando diferentes temperaturas entre 0.2 y 1.2, llegamos a la conclusión de que el valor de 0.4 seguía siendo el mejor.

C.1.3. Tabla de parámetros

En la tabla C.1 se muestran los valores utilizados para el resto de parámetros de la arquitectura.

Parámetro	Valor	Descripción
hidden_size	768	Tamaño del vector numérico de salida del codificador y que utiliza después el decodificador.
use_attention	false	Si el decodificador utiliza atención o no.
xy_distribution_size	32	Tamaño de la cuadrícula en la que se reparte la imagen para la generación de posiciones.
bidirectional	true	Si el decodificador es bidireccional.
temperature	0.4	Temperatura a utilizar en el *temperature softmax* para elegir la posición entre las probilidades dadas por la arquitectura.
pooling_type	max	Tipo de *pooling* a utilizar para fusionar los vectores de salida del codificador en un solo vector. Puede tomar los valores max, avg y cls.

Tabla C.1: Parámetros utilizados en la versión final de TRAN2LY

C.2. STRAN2LY

Para esta arquitectura no se ha realizado mucha experimentación. El codificador está ya preparado para su uso en la tarea y el decodificador ya ha sido parametrizado en la anterior arquitectura.

Parámetro	Valor	Descripción
hidden_size	768	Tamaño del vector numérico de salida del codificador y que utiliza después el decodificador.
use_attention	false	Si el decodificador utiliza atención o no.
xy_distribution_size	32	Tamaño de la cuadrícula en la que se reparte la imagen para la generación de posiciones.
bidirectional	true	Si el decodificador es bidireccional.
temperature	0.4	Temperatura a utilizar en el *temperature softmax* para elegir la posición entre las probilidades dadas por la arquitectura.

Tabla C.2: Parámetros utilizados en la versión final de STRAN2LY

C.3. TRAN2TRAN

El codificador es el mismo que el de TRAN2LY pero no hace falta fusionar los vectores de salida para este decodificador así que no hay parámetros con los que experimentar por

el lado del codificador.

Por otro lado, el decodificador tiene muchísimos parámetros a optimizar. Se ha probado con diferentes parámetros para intentar conseguir que funcionase. Por desgracia ha sido en vano. No se han conseguido resultados aceptables con esta arquitectura.

En la tabla C.3 se muestran los diferentes valores con los que se ha probado cada parámetro.

Parámetro	Valores Probados	Descripción
hidden_size	768	Tamaño del vector numérico de salida del codificador y que utiliza después el decodificador.
xy_distribution_size	32	Tamaño de la cuadrícula en la que se reparte la imagen para la generación de posiciones.
temperature	0.4	Temperatura a utilizar en el *temperature softmax* para elegir la posición entre las probilidades dadas por la arquitectura.
nhead	3, 6, 12	Número de cabezas de atención en cada capa del decodificador.
num_decoder_layers	3, 6, 12	Número de capas del decodificador.
dim_FFN	1024, 2048, 3072	Dimensión de las *feed forward networks* internas del decodificador.
class_hidden_layers	[], [256], [512], [1024]	Lista de tamaños de las capas ocultas de la red neuronal que genera la clase a partir del vector de salida del decodificador.
xy_hidden_layers	[256], [512], [1024], [1024, 256]	Lista de tamaños de las capas ocultas de la red neuronal que genera la clase a partir del vector de salida del decodificador y la clase.
wh_hidden_layers	[256], [512], [1024], [1024, 256]	Lista de tamaños de las capas ocultas de la red neuronal que genera la clase a partir del vector de salida del decodificador, la clase y la posición.

Tabla C.3: Parámetros probados para TRAN2TRAN

Hay que remarcar que cada valor de cada parámetro o se ha probado con todas las posibles combinaciones de los valores del resto de parámetros. Cada parámetro se ha probado por separado manteniendo el resto de valores constantes. Después de cada grupo de pruebas se ha tomado el valor que mejor resultado ha dado de ahí en adelante.

Los valores finales con los que se han hecho las pruebas para los resultados mostrados en

este documento son los de la tabla C.4.

Parámetro	Valor	Descripción
hidden_size	768	Tamaño del vector numérico de salida del codificador y que utiliza después el decodificador.
xy_distribution_size	32	Tamaño de la cuadrícula en la que se reparte la imagen para la generación de posiciones.
temperature	0.4	Temperatura a utilizar en el *temperature softmax* para elegir la posición entre las probilidades dadas por la arquitectura.
nhead	6	Número de cabezas de atención en cada capa del decodificador.
num_decoder_layers	6	Número de capas del decodificador.
dim_FFN	3072	Dimensión de las *feed forward networks* internas del decodificador.
class_hidden_layers	[]	Lista de tamaños de las capas ocultas de la red neuronal que genera la clase a partir del vector de salida del decodificador.
xy_hidden_layers	[512]	Lista de tamaños de las capas ocultas de la red neuronal que genera la clase a partir del vector de salida del decodificador y la clase.
wh_hidden_layers	[512]	Lista de tamaños de las capas ocultas de la red neuronal que genera la clase a partir del vector de salida del decodificador, la clase y la posición.

Tabla C.4: Parámetros utilizados en la versión final de TRAN2TRAN

Bibliografía

[Alammar, 2020] Alammar, J. (2020). The illustrated transformer. https://jalammar.github.io/illustrated-transformer/.

[Brown et al., 2020] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

[Carion et al., 2020] Carion, N., Massa, F., Synnaeve, G., Usunier, N., Kirillov, A., and Zagoruyko, S. (2020). End-to-end object detection with transformers. In *European conference on computer vision*, pages 213–229. Springer.

[Devlin et al., 2018] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

[Dhariwal and Nichol, 2021] Dhariwal, P. and Nichol, A. (2021). Diffusion models beat gans on image synthesis. *Advances in Neural Information Processing Systems*, 34:8780–8794.

[Dominguez, 2021] Dominguez, C. (2021). Text to layout. https://github.com/CarlosDominguezBecerril/text-to-layout.

[Hong et al., 2018] Hong, S., Yang, D., Choi, J., and Lee, H. (2018). Inferring semantic layout for hierarchical text-to-image synthesis. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7986–7994.

[Johnson et al., 2018] Johnson, J., Gupta, A., and Fei-Fei, L. (2018). Image generation from scene graphs. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1219–1228.

[Li et al., 2019] Li, W., Zhang, P., Zhang, L., Huang, Q., He, X., Lyu, S., and Gao, J. (2019). Object-driven text-to-image synthesis via adversarial training. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12174–12182.

[Liang et al., 2022] Liang, J., Pei, W., and Lu, F. (2022). Layout-bridging text-to-image synthesis. *arXiv preprint arXiv:2208.06162*.

[Lin et al., 2014] Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., and Zitnick, C. L. (2014). Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer.

[Liu et al., 2022] Liu, X., Yin, D., Feng, Y., and Zhao, D. (2022). Things not written in text: Exploring spatial commonsense from visual signals. *arXiv preprint arXiv:2203.08075*.

[Reed et al., 2016] Reed, S., Akata, Z., Yan, X., Logeswaran, L., Schiele, B., and Lee, H. (2016). Generative adversarial text to image synthesis. In *International conference on machine learning*, pages 1060–1069. PMLR.

[Rogers et al., 2020] Rogers, A., Kovaleva, O., and Rumshisky, A. (2020). A primer in bertology: What we know about how bert works. *Transactions of the Association for Computational Linguistics*, 8:842–866.

[Sharma et al., 2018] Sharma, P., Ding, N., Goodman, S., and Soricut, R. (2018). Conceptual captions: A cleaned, hypernymed, image alt-text dataset for automatic image captioning. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2556–2565.

[Song et al., 2020] Song, K., Tan, X., Qin, T., Lu, J., and Liu, T.-Y. (2020). Mpnet: Masked and permuted pre-training for language understanding. *Advances in Neural Information Processing Systems*, 33:16857–16867.

[Thoppilan et al., 2022] Thoppilan, R., De Freitas, D., Hall, J., Shazeer, N., Kulshreshtha, A., Cheng, H.-T., Jin, A., Bos, T., Baker, L., Du, Y., et al. (2022). Lamda: Language models for dialog applications. *arXiv preprint arXiv:2201.08239*.

[Vaswani et al., 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 30.

[Vig, 2021] Vig, J. (2021). Bertviz. https://github.com/jessevig/bertviz.

[von Platen, 2020] von Platen, P. (2020). Transformers-based encoder-decoder models. https://huggingface.co/blog/encoder-decoder.

[Zakraoui et al., 2021] Zakraoui, J., Saleh, M., Al-Maadeed, S., and Jaam, J. M. (2021). Improving text-to-image generation with object layout guidance. *Multimedia Tools and Applications*, 80(18):27423–27443.

[Zhang et al., 2017] Zhang, H., Xu, T., Li, H., Zhang, S., Wang, X., Huang, X., and Metaxas, D. N. (2017). Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks. In *Proceedings of the IEEE international conference on computer vision*, pages 5907–5915.

[Zhu et al., 2019] Zhu, M., Pan, P., Chen, W., and Yang, Y. (2019). Dm-gan: Dynamic memory generative adversarial networks for text-to-image synthesis. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 5802–5810.