Trabajo Práctico 4

Unidad 6 - Redes Neuronales
Unidad 7 - Aplicaciones

Problema 1 - Tráfico

Objetivo

Escriba una IA para identificar qué señal de tráfico aparece en una fotografía.

```
$ python traffic.py gtsrb
Epoch 1/10
500/500 [==============================] - 5s 9ms/step - loss: 3.7139 - a
Epoch 2/10
500/500 [==============================] - 6s 11ms/step - loss: 2.0086 -
Epoch 3/10
500/500 [==============================] - 6s 12ms/step - loss: 1.3055 -
Epoch 4/10
500/500 [==============================] - 5s 11ms/step - loss: 0.9181 -
Epoch 5/10
500/500 [==============================] - 7s 13ms/step - loss: 0.6560 -
Epoch 6/10
500/500 [==============================] - 9s 18ms/step - loss: 0.5078 -
Epoch 7/10
500/500 [==============================] - 9s 18ms/step - loss: 0.4216 -
Epoch 8/10
500/500 [==============================] - 10s 20ms/step - loss: 0.3526 -
Epoch 9/10
500/500 [==============================] - 10s 21ms/step - loss: 0.3016 -
Epoch 10/10
500/500 [==============================] - 10s 20ms/step - loss: 0.2497 -
333/333 - 5s - loss: 0.1616 - accuracy: 0.9535
```

Introducción

Mientras la investigación avanza en el desarrollo de coches autónomos, uno de los retos clave es la visión por computadora, que permite a estos coches que desarrollen una comprensión de su entorno a partir de imágenes digitales. En particular, esto implica la capacidad de reconocer y distinguir señales viales: de alto, límite de velocidad, ceder el paso, y más.

En este proyecto, usted usará TensorFlow para construir una red neuronal para clasificar las señales de tráfico basadas en una imagen de las mismas. Para ello, necesitarás un conjunto de datos etiquetado: una colección de imágenes que ya han sido categorizadas de acuerdo a

la señal que tienen las mismas.

Existen varios conjuntos de datos (dataset) de este tipo, pero para este proyecto, utilizaremos el conjunto German Traffic Sign Recognition Benchmark (GTSRB), que contiene miles de imágenes con 43 diferentes tipos de señales de transito.

Herramientas

- Descargue el código `problema1-trafico.zip` y descomprimirlo.
- Descargue el dataset `problema1-dataset.zip` para este proyecto y descomprimirlo. Mueve el directorio resultante *gtsrb* dentro del directorio *traffic*.
- Dentro del directorio *traffic*, ejecute **pip3 install -r requirements.txt** para la instalación de las dependencias de este proyecto: `opencv-python` para el procesamiento de imágenes, `scikit-learn` para funciones relacionadas con el ML, y `tensorflow` para las redes neuronales.

Explicación del problema

Primero, echa un vista a los datos establecidos abriendo el directorio *gtsrb*. Usted notará 43 subdirectorios en este conjunto de datos, numerados 0 a 42. Cada subdirectorio numerado representa una categoría diferente (un tipo de señal vial). Dentro de cada directorio de señal de tráfico hay una colección de imágenes de ese tipo de señal de tráfico.

A continuación, echa un vistazo a `traffic.py`. En el *main*, aceptamos como argumentos de línea de comando un directorio que contiene el dataset y (opcionalmente) un nombre de archivo al que guardar el modelo entrenado. Los datos y las etiquetas correspondientes se cargan luego del directorio de datos (a través de la función *load_data*) y dividido en conjuntos (datasets) de entrenamiento y pruebas. Después de eso, se llama a la función *get_model* para obtener una red neuronal compilada que luego se ajusta en función a los datos de entrenamiento. El modelo se evalúa entonces con los datos de prueba (test dataset). Finalmente, si se proporcionó un nombre de archivo modelo, el modelo entrenado se guarda en el disco para ser usado en el futuro.

Las funciones *load_data* y *get_model* le tocan implementar a usted.

Especificación

Completar la implementación de *load_data* y *get_model* en `traffic.py`.

- La función *load_data* debe aceptar como argumento **data_dir**, representando la ruta a un directorio donde se almacenan los datos, y devuelve un array ndimensional (matriz) de imágenes y etiquetas para cada imagen en el conjunto de datos.

- Usted puede asumir que *data_dir* contendrá un directorio que lleva el nombre de cada categoría, numerada de **0** a **NUM_CATEGORIES - 1**. Dentro de cada directorio de categoría habrá un número de archivos de imágenes variable.
- Utilice el módulo OpenCV-Python (cv2) para leer cada imagen como `numpy.ndarray` (un arreglo multidimensional `numpy`). Para pasar estas imágenes a una red neuronal, las imágenes tendrán que ser del mismo tamaño, así que asegúrese de cambiar el tamaño de cada imagen para tener ancho **IMG_WIDTH** y altura **IMG_HEIGHT**.
- La función debe devolver una tupla **(images, labels)**. **images** debe ser una lista de todas las imágenes en el conjunto de datos, donde cada imagen se representa como un `numpy.ndarray` del tamaño apropiado. **labels** debe ser una lista de enteros (integer), que representen el número de la categoría para cada una de las imágenes correspondientes en la lista **images**.
- Su función debe ser independiente de la plataforma: es decir, debe funcionar independientemente del sistema operativo. Tenga en cuenta que en macOS, el carácter / se utiliza para separar los componentes de la ruta (path), mientras que el el carácter se utiliza en Windows. Usa os.sep y os.path.join según sea necesario en lugar de utilizar su carácter separador específico de la plataforma (hardcodeado).

- La función *get_model* debe devolver un modelo de red neuronal compilado.
  - Usted puede asumir que la entrada a la red neuronal será de la forma **(IMG_WIDTH, IMG_HEIGHT, 3)**(es decir, un array que representa una imagen de ancho IMG_WIDTH, altura IMG_HEIGHT, y 3 valores para cada píxel rojo, verde y azul, RGB).
  - La capa de salida de la red neuronal debe tener unidades **NUM_CATEGORIES**, una para cada una de las categorías de señales de tráfico.
  - El número de capas y los tipos de capas que usted incluya en el medio depende de usted. Es posible que desee experimentar con:
    - diferentes números de capas de convolución y de pooling
    - diferentes números y tamaños de filtros para capas convolucionales
    - diferentes tamaños de pool para las capas de pooling
    - números y tamaños diferentes de capas ocultas
    - dropout
- En un archivo separado llamado README.md, documenta (al menos en un párrafo o dos) su proceso de experimentación. Qué intentaste? Lo que funcionó bien? Qué no funcionó bien? Qué te has dado cuenta?

En última instancia, gran parte de este proyecto se trata de explorar la documentación e investigar diferentes opciones en cv2 y tensorflow, y ver qué resultados obtienes cuando los pruebas.

No debería modificar nada más en `traffic.py` aparte de las funciones que la especificación requiere que implemente, aunque puede escribir funciones adicionales y/o importar otros módulos de biblioteca estándar de Python. También puede importar numpy o o pandas, si está familiarizado con ellos, pero no debe utilizar ningún otro módulo Python de terceros. Puede modificar las variables globales definidas en la parte superior del archivo para probar su programa con otros valores.

Recomendaciones

- Echa un vistazo a la documentación oficial de Tensorflow Keras para algunas recomendaciones para la sintaxis de la construcción de una red neuronal multicapa. Usted puede encontrar de utilidad el código fuente visto en clase.
- La documentación de OpenCV-Python puede resultar útil para leer imágenes como arrays y luego editarlas (cambiar el tamaño, orientación).
- Una vez que hayas re-dimensionado una imagen **img**, puede verificar sus dimensiones imprimiendo el valor de img.shape. Si re-dimensionas la imagen correctamente, su forma debe ser (30, 30, 3)(asumiendo IMG_WIDTHy IMG_HEIGHTambos 30).
- Si desea practicar con un conjunto de datos más pequeño, puede descargar el dataset modificado `problema1-dataset-reducido.zip` que contiene sólo 3 tipos diferentes de señales de carretera en lugar de 43.

Problema 2 - Parser

Objetivo

Escriba una IA para parsear oraciones en ingles y extraer frases de sustantivo.

```
$ python parser.py
Sentence: Holmes sat.
      S
 _____|_____
NP          VP
|           |
N           V
|           |
holmes      sat

Noun Phrase Chunks
holmes
```

Introducción

Una tarea común en el procesamiento del lenguaje natural es el análisis, el proceso de determinación de la estructura de una oración. Esto es útil por una serie de razones: conocer la estructura de una oración puede ayudar a una computadora a entender mejor el significado de la oración, y también puede ayudar a la computadora a extraer información de una oración. En particular, a menudo es útil extraer frases sustantivas de una oración para obtener un entendimiento de lo que se trata la oración.

En este problema, usaremos gramática libre de contexto para analizar las oraciones en ingles para determinar su estructura. En una gramática libre de contexto, aplicamos repetidamente reglas de reescritura para transformar símbolos en otros símbolos. El objetivo es comenzar con un símbolo no terminal **S** (representando una oración, **S**entence) y aplicar repetidamente reglas de gramática libre de contexto hasta que generemos una frase completa de símbolos terminales (es decir, palabras). La regla $S \rightarrow NV$, por ejemplo, significa que el símbolo **S** puede ser reescrito como **N V**(un sustantivo seguido de un verbo). Si también tenemos la regla $N \rightarrow' Holmes'$ y la regla **V -¿'sat'**, podemos generar la oración completa 'Holmes sat.'.

Por supuesto, las frases de sustantivo no siempre pueden ser tan simples como una sola palabra como 'Holmes'. Podríamos tener frases sustantivas como *'my companion'* o *'a country walk'* o *'the day before Thursday'*, que requieren normas más complejas para tener en cuenta. Para tener en cuenta la frase 'my companion', por ejemplo, podríamos imaginar una regla como:

$$NP- > N|DetN$$

En esta regla, decimos que un **NP** (una frase sustantiva) podría ser sólo un sustantivo (N) o un determinante (Det) seguido de un sustantivo, donde los determinantes incluyen palabras como *'a'*, *'the'*, y *'my'*. La barra vertical (—) sólo indica que hay múltiples maneras posibles de reescribir un NP, con cada posible reescribo separado por una barra (or programático).

Incorporar esta regla en la forma en que analizamos una oración (S), también tenemos que modificar la regla anterior $S- > NV$ para permitir frases de sustantivo (NPs) como objeto de nuestra sentencia. Ves cómo? Y para dar cuenta de tipos más complejos de frases de sustantivos, es posible que necesitemos modificar nuestra gramática aún más.

Explicación del problema

En primer lugar, mire los archivos de texto en el directorio *sentences*. Cada archivo contiene una oración en inglés. Su objetivo en este problema es escribir un analizador que sea capaz de analizar todas estas frases.

Echa un vistazo ahora a `parser.py`, y tome nota de las reglas gramaticales libres de contexto definidas en la parte superior del archivo. Ya hemos definido para usted un conjunto

de reglas para la generación de símbolos terminales (en la variable global **TERMINALS**). Note que **Adj** es un símbolo no terminal que genera adjetivos, **Adv** genera adverbios, **Conj** genera conjunciones, **Det** genera determinantes, **N** genera sustantivos (repartidos en múltiples líneas para mejor legibilidad), **P** genera preposiciones, y **V** genera verbos.

A continuación está la definición de **NONTERMINALS**, que contendrá todas las reglas gramaticales libres de contexto para generar símbolos no terminales. Ahora mismo, sólo hay una sola regla: $S- > NV$. Con solo esa regla, podemos generar frases como 'Holmes arrived.' o 'He chuckled.', pero no frases más complejas que eso. Editar las reglas reglas **NONTERMINALS** para que todas las frases puedan ser analizadas depende de ti.

A continuación, echa un vistazo a la función main. Primero acepta una oración como entrada, ya sea desde un archivo o a través de la entrada del usuario. La oración es preprocesada (a través de la función preprocess) y luego analizada de acuerdo a la gramática libre de contexto definida en el archivo. Los árboles resultantes se imprimen, y todos los comoponentees de frases sustantivas (definidas en la especificación) también se imprimen (a través de la función *np_chunk*).

Además de escribir reglas gramaticales libres de contexto para analizar estas oraciones, también las funciones preprocess y np_chunk deben ser implementadas.

Especificación

Completar la aplicación de *preprocess* y *np_chunk*, y completar las reglas gramaticales sin contexto definidas en **NONTERMINALS**.

- La función *preprocess* debe aceptar una **sentence** como entrada y devuelve una lista de sus palabras.
  - Usted puede asumir que **sentence** será un **string**.
  - Deberías usar la función de la librería nltk *word_tokenize* para realizar tokenización.
  - Su función debe devolver una lista de palabras, donde cada palabra es un string en minúsculas.
  - Cualquier palabra que no contenga al menos un carácter alfabético (por ejemplo **.** o **28**) debe excluirse de la lista devuelta.
- La variable global **NONTERMINALS** debe reemplazarse por un set de normas gramaticales libres de contexto que, cuando se combinen con las reglas en **TERMINALS**, permitan parsear todas las oraciones en el Directorio *sentences*.
  - Cada regla debe estar en su propia línea. Cada regla debe incluir los caracteres **-¿** para denotar qué símbolo se está sustituyendo, y pueden incluir opcionalmente símbolos — si hay múltiples maneras de reescribir un símbolo.

- No es necesario mantener la regla existente $S- > NV$ en su solución, pero su primera regla debe comenzar con **S -¿** ya que **S**(representando una oración) es el símbolo de partida.
- Usted puede agregar tantos símbolos no terminales como desee.
- Utilice el símbolo no terminal **NP** para representar una frase sustantiva, como el objeto de una oración.

- La función *np_chunk* debe aceptar un **tree** representando la sintaxis de una oración, y devuelve una lista de todas las letras de la frase sustantiva en esa oración.
  - Para este problema, un *'chunk' de frase sustantiva* se define como una frase sustantiva que no contiene otras frases sustantivas dentro de ella. Dicho más formalmente, un trozo de frase sustantiva es un sub-árbol del árbol original cuya etiqueta es **NP** y eso no contiene otras frases de sustantivo como subárboles.
    - Por ejemplo, si 'the home' es un trozo de frase sustantiva, entonces 'the armchair in the home' no es una frase sustantiva, porque el segundo contiene el primero como subárbol.
  - Usted puede asumir que la entrada será un objeto *nltk.tree* cuya etiqueta es **S** (es decir, la entrada será un árbol que represente una oración).
  - Su función debe devolver una lista de objetos *nltk.tree*, donde cada elemento tiene la etiqueta **NP**.
  - Revise la documentación para nltk.tree, le será útil para identificar cómo manipular un objeto nltk.tree.

No debería modificar nada más en `parser.py` aparte de las funciones, la especificación requiere que implemente, aunque puede escribir funciones adicionales y/o importar otros módulos de biblioteca estándar de Python. Usted tendrá que modificar la definición de **NONTERMINALS**, pero no debe modificar la definición de **TERMINALS**.

Recomendaciones

- Es de esperar que su analizador pueda generar algunas frases que usted crea que no son sintácticas o semánticamente correctas. No tienes que preocuparte, por lo tanto, si tu analizador permite analizar oraciones sin sentido como 'His Thursday chuckled in a paint.'
  - Dicho esto, deberías evitar la generación excesiva de sentencias cuando sea posible. Por ejemplo, su analizador definitivamente no debería aceptar oraciones como 'Armchair on the sat Holmes.'
  - También debe evitar la sub-generación de sentencias. Una regla como $S- > NVDetAdjAdjAdjNPDetNPDetN$ técnicamente generaría la oración 10, pero no de manera particularmente útil o generalizable.

- Las reglas del código fuente vistas en clase son (intencionadamente) un conjunto de reglas muy simplificado, y como resultado puede sufrir de una sobregeneración. Usted puede (y debe) hacer modificaciones a esas reglas para tratar de ser lo más general posible sin sobregenerar. En particular, considere cómo podría hacer que su analizador acepte la frase 'Holmes sat in the armchair' (y 'Holmes sat in the red armchair'. y 'Holmes sat in the little red armchair'), pero tienes que hacer que **no** acepte la frase 'Holmes sat in the the armchair."

- Es de esperar que su analizador pueda generar múltiples maneras de analizar una oración. La gramática inglesa es inherentemente ambigua.
- Dentro de la nltk.tree documentación, puede encontrar útiles las funciones *label* y *subtrees*.
- Para centrarse en probar su analizador antes de trabajar en la extracción de frases sustantivas, puede ser útil tener temporalmente hacer que *np_chunk* simplemente devuelve una lista vacía [], para que su programa pueda operar sin extraer las frases sustantivas mientras pruebas las otras partes de tu programa.

Problema 3 - Atención

Objetivo

Escriba una IA para predecir una palabra escondida en una secuencia de texto.

```
$ python mask.py
Text: We turned down a narrow lane and passed through a small [MASK].
We turned down a narrow lane and passed through a small field.
We turned down a narrow lane and passed through a small clearing.
We turned down a narrow lane and passed through a small park.

$ python mask.py
Text: Then I picked up a [MASK] from the table.
Then I picked up a book from the table.
Then I picked up a bottle from the table.
Then I picked up a plate from the table.
```

Introducción

Una forma de crear modelos de lenguaje es construir un modelo de lenguaje enmascarado, donde se entrena a un modelo de lenguaje para predecir una palabra escondida que falta en una secuencia de texto. BERT es un modelo de lenguaje basado en transformadores desarrollado por Google, y fue entrenado con este enfoque: el modelo de lenguaje fue entrenado para predecir una palabra enmascarada basada en las palabras de contexto circundante.

BERT utiliza una arquitectura con transformadores y por lo tanto utiliza un mecanismo de atención para entender el lenguaje. En el modelo base de BERT, el transformador utiliza 12 capas, donde cada capa tiene 12 cabezas (heads) de auto-atención, para un total de 144 cabezas (heads) de auto-atención.

Este proyecto incluirá dos partes:

Primero, usaremos transformers, la biblioteca de Python, desarrollada por la compañía de software de IA Hugging Face, para escribir un programa que utiliza BERT para predecir palabras enmascaradas. El programa también generará diagramas que visualizan las puntuaciones (scores) de atención, con un diagrama generado para cada una de las 144 cabezas de atención.

En segundo lugar, analizaremos los diagramas generados por nuestro programa para tratar de entender que *cabezas de atención* de BERT podrían estar prestando atención mientras intenta entender nuestro lenguaje natural.

- Descargue el código `problema3-atencion.zip`.
- Dentro del directorio *attention*, ejecute $pip3 install -r requirements.txt$ para instalar las dependencias del proyecto.

Explicación del problema

Primero, echa un vistazo al programa `mask.py`. En el main, se le solicita primero para algún texto como entrada al usuario. La entrada de texto debe contener un token de máscara [**MASK**] representando la palabra que nuestro modelo de lenguaje debería tratar de predecir. La función entonces utiliza un AutoTokenizer para tomar la entrada y dividirla en tokens.

En el modelo BERT, cada token distinto tiene su propio número de identificación. Una identificación, dada por *tokenizer.mask_token_id*, corresponde al token [**MASK**]. La mayoría de los otros tokens representan palabras, con algunas excepciones. El token [**CLS**] siempre aparece al principio de una secuencia de texto. El token [**SEP**] aparece al final de una secuencia de texto y se utiliza para separar secuencias entre sí. A veces una sola palabra se divide en múltiples tokens: por ejemplo, BERT trata la palabra *'inteligently'*, como dos tokens: **intelligent** y **##ly**.

A continuación, usamos una instancia de TFBertForMaskedLM para predecir un token enmascarado usando el modelo de lenguaje BERT. Los token de entrada **(inputs)** se pasan en el modelo, y luego buscamos los **K** mejores tokens de salida. La secuencia original se imprime con el token de máscara reemplazado por cada una de los tokens de salida predichos.

Finalmente, el programa llama a la función *visualize_attentions* , que debe generar diagramas de los valores de atención para la secuencia de entrada para cada cabezal de atención de BERT.

La mayor parte del código ha sido escrito para usted, pero las implementaciones de *get_mask_token_index*, *get_color_for_attention_score*, y *visualize_attentions* debes implementarlas tú.

Una vez que haya completado esas tres funciones, el programa `mask.py` generará diagramas de atención. Estos diagramas pueden darnos una idea de lo que BERT ha aprendido a prestar atención cuando trata de darle sentido al lenguaje. Por ejemplo, en la Fig. 1 se encuentra el diagrama de atención de la capa 3, cabeza 10 cuando procesaba la oración 'Entonces recogí un [MASK]' de la tabla.

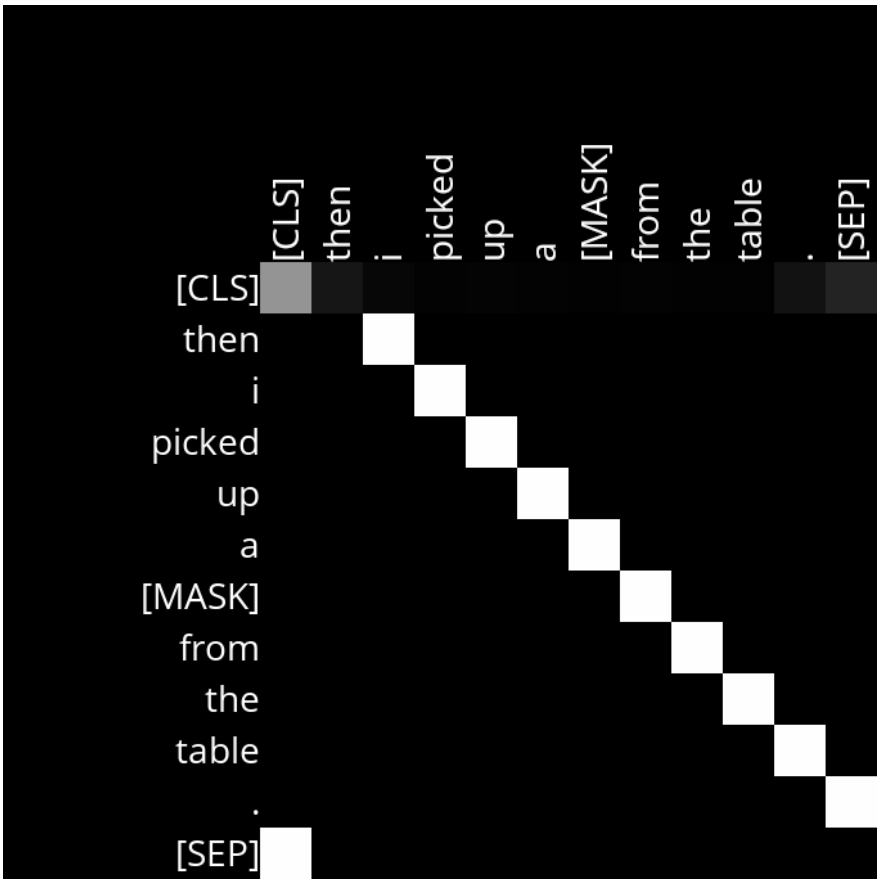

Figura 1: Diagrama de atención capa 3, cabeza 10

Recuerde que los colores más claros representan un mayor peso de atención y los colores más oscuros representan un menor peso de la atención. En este caso, esta cabeza de atención parece haber aprendido un patrón muy claro: cada palabra está prestando atención a la palabra que inmediatamente la sigue. La palabra 'then', por ejemplo, está representada por la segunda fila del diagrama, y en esa fila la celda más brillante es la celda correspondiente a la columna I, sugiriendo que la palabra 'then', está asistiendo fuertemente a la palabra I. Lo mismo ocurre con los otros tokens de la oración.

Puedes intentar correr `mask.py` en otras oraciones para verificar que la capa 3, la cabeza 10 continúa siguiendo este patrón. Y tiene sentido intuitivamente que BERT pueda aprender

a identificar este patrón: entender una palabra en una secuencia de texto a menudo depende de saber qué palabra viene después, así que tener una cabeza de atención (o múltiples) dedicada a prestar atención a lo que la palabra viene después podría ser útil.

Esta cabeza de atención es particularmente clara, pero a menudo las cabezas de atención serán más ruidosas y podrían requerir algo más de interpretación para adivinar a qué BERT puede estar prestando atención.

Digamos, por ejemplo, teníamos curiosidad por saber si BERT presta atención al papel de los adverbios. Podemos darle al modelo una oración como “The turtle moved slowly across the [MASK].”, en español, ’La tortuga se movió lentamente a través del [MASK]’, y luego mirar las cabezas de atención resultantes para ver si el modelo de lenguaje parece notar que ’slowly’, es un adverbio que modifica la palabra ’moved’. Mirando los diagramas de atención resultantes, uno que podría llamar tu atención es la Layer 4, Head 11. Esto se puede observar en la figura 2.

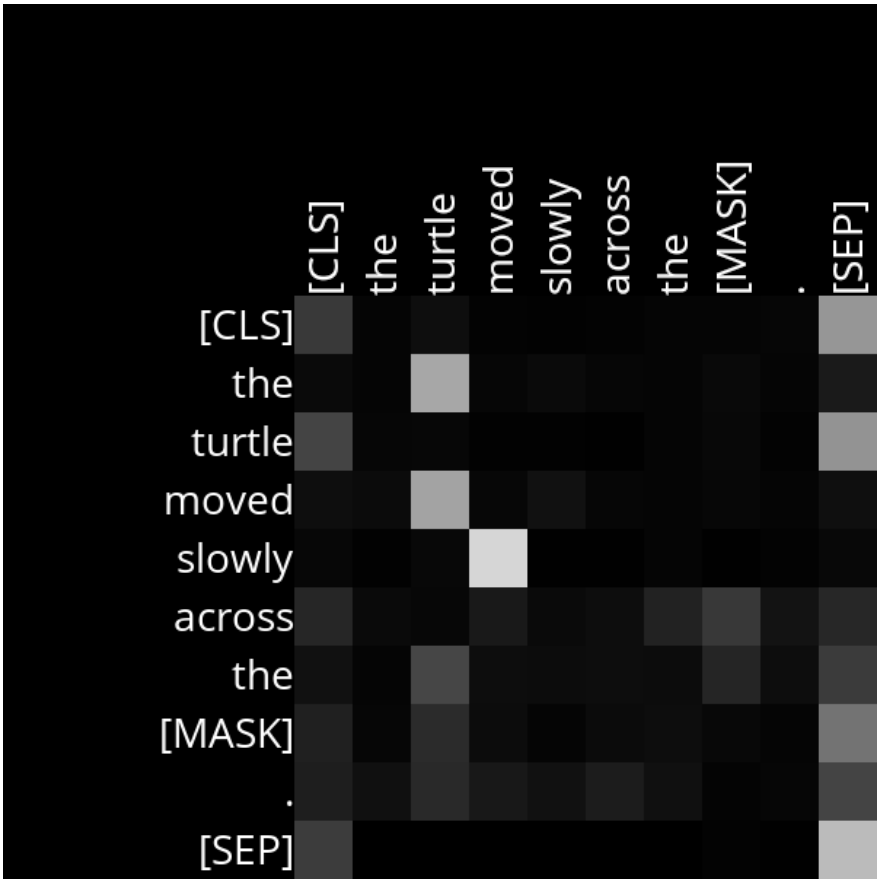

Figura 2: Diagrama de atención cabeza 11 de la capa 4.

Esta cabeza de atención es definitivamente más ruidosa: no es inmediatamente obvio exactamente lo que esta cabeza de atención está haciendo. Pero observe que, para el adverbio ’slow’, atiende más al verbo que modifica: ’movido’. Lo mismo es cierto si intercambiamos

el orden del verbo y adverbio, resultado que se puede observar en la figura 3.

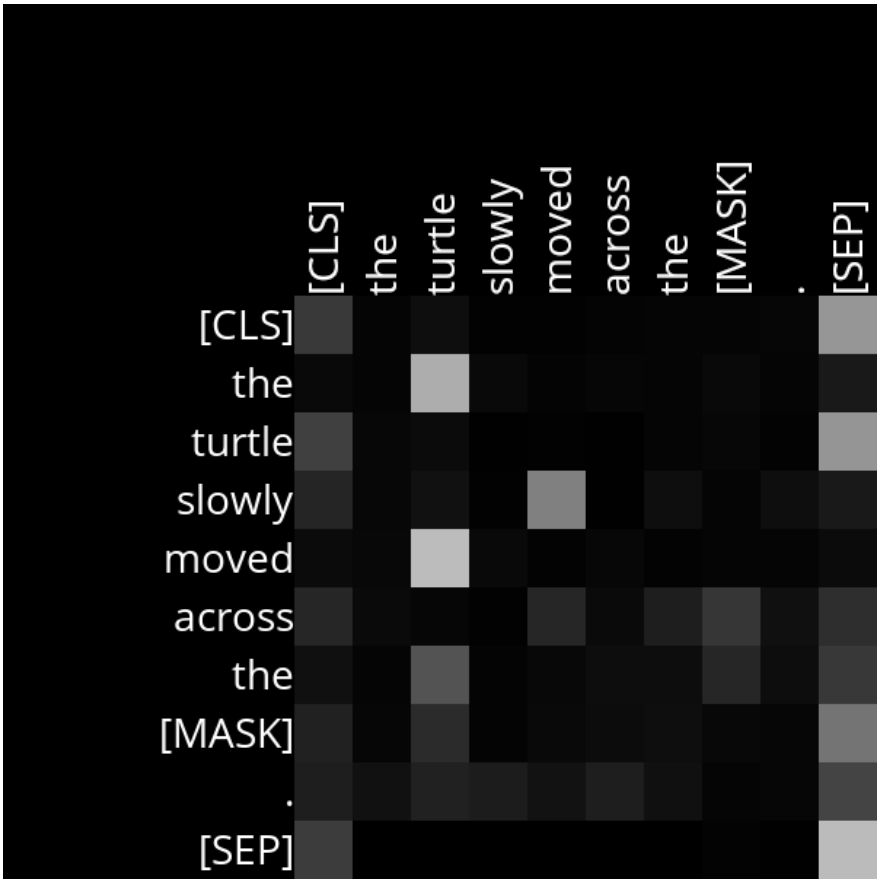

Figura 3

E incluso parece ser cierto para una frase donde el adverbio y el verbo que modifica no están directamente uno al lado del otro, como podemos ver en la Fig. 4.

Así que podríamos razonablemente adivinar que esta cabeza de atención ha aprendido a prestar atención a la relación entre los adverbios y las palabras que modifican. Las capas de atención no siempre se alinearán consistentemente con nuestras expectativas para una relación particular entre palabras, y no siempre corresponderán a una relación humanamente interpretable en absoluto, pero podemos adivinar o inferir basados en lo que parezcan corresponder.. Y eso haremos en este proyecto!.

Especificación

En primer lugar, completar la implrementación de *get_mask_token_index*, *get_color_for_attention_sc* y *visualize_attentions*.

- La función *get_mask_token_index* acepta el **ID** del token de la máscara (representado como una int) y el **inputs** generado por el *tokenizer*, que será de tipo **transformers.BatchEncoding**. Debería devolver el índice del token de la máscara en la secuencia de tokens de entrada .

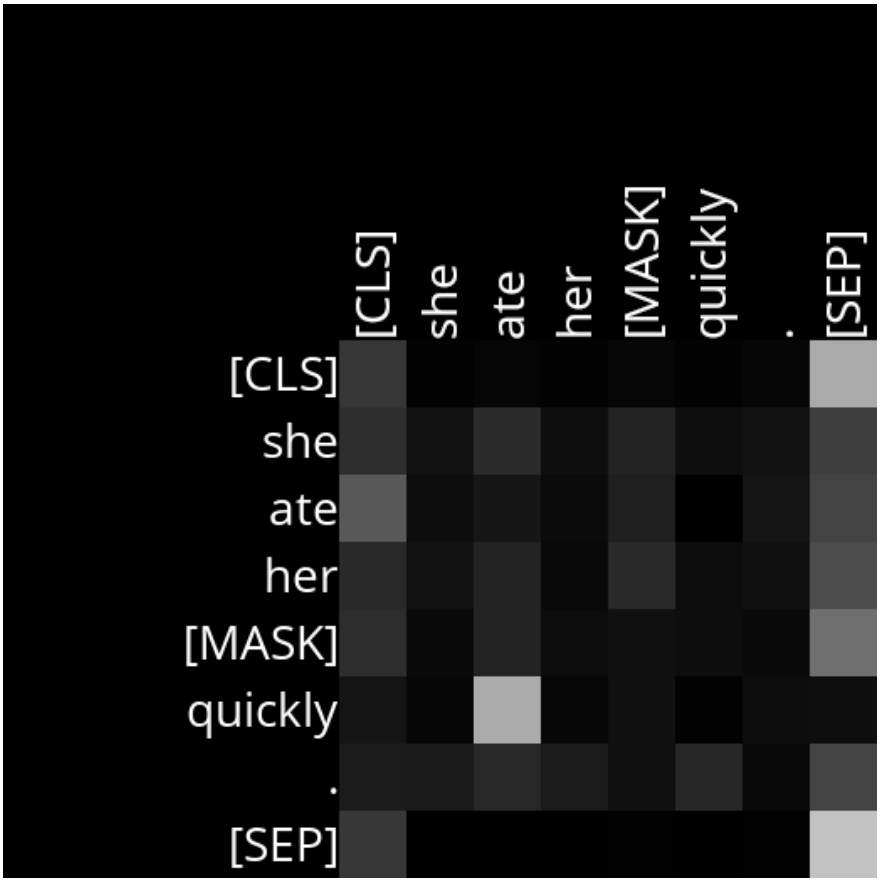

Figura 4

- El índice debería estar indexado a 0. Por ejemplo, si el tercer ID de entrada es el ID de máscara, entonces su función debe devolver 2.
- Si el token de máscara no está presente en la secuencia de entrada en absoluto, su función debe regresar **None**.
- Usted puede asumir que no habrá más de un símbolo de más de una máscara en la secuencia de entrada.
- Usted puede encontrar útil para mirar la documentación de *transformers*, en particular en el valor de retorno de llamar a un tokenizador, para ver qué campos tendrá el BatchEncoding que puedas querer acceder.

- La función *get_color_for_attention_score* debe aceptar una puntuación de atención (un valor entre 0 y 1, inclusive) y sacer una tupla de tres números enteros que representan un trío RGB (un valor rojo, un valor verde, un valor azul) para el color a utilizar para esa celda de atención en el diagrama de atención.
  - Si la puntuación de atención es 0, el color debe ser completamente negro (el valor (0, 0, 0)). Si la puntuación de atención es 1, el color debe ser totalmente blanco (el valor (255, 255, 255)). Para los puntajes de atención en el medio, el color debe ser una sombra de gris que escala linealmente con la puntuación de atención.

- Para que un color sea una sombra de gris, los valores rojos, azules y verdes deben ser iguales.
- Los valores rojos, verdes y azules deben ser enteros, pero se puede elegir si truncar o redondear los valores. Por ejemplo, para la puntuación de atención 0.25, su función puede devolver cualquiera de las dos (63, 63, 63)o o (64, 64, 64), ya que el 25 % de 255 es de 63,75.

- La función *visualize_attentions* acepta una secuencia de tokens (una lista de *strings*) así como **attentions**, que contiene todas las puntuaciones de atención generadas por el modelo. Para cada cabezal de atención, la función debe generar un diagrama de visualización de la atención, como se genera llamando *generate_diagram*.
  - El valor **attentions** es una tupla de tensores (un tensor puede ser considerado como una matriz multidimensional en este contexto).
  - A indexar en el valor attentions para obtener una atención específica de los valores de la cabeza, usted puede hacerlo como **attentions**[**i**][**j**][**k**], dónde **i** es el índice de la capa de atención, **j** es el índice del número del haz (siempre 0 en nuestro caso), y **k** es el índice de la cabeza de atención en la capa.
  - Esta función contiene una implementación existente que genera un solo diagrama de atención, para la primera cabeza de atención en la primera capa de atención. Su tarea es extender esta implementación para generar diagramas para todas las cabezas de atención y capas.
  - La función *generate_diagram* espera que las dos primeras entradas sean el número de capa y el número de cabeza. Estos números deberían estar indexados desde 1 . En otras palabras, para la primera capa de atención y cabeza de atención (cada una de las cuales tiene índice 0), **layer_number** debería ser 1 y **head_number** debería ser 1 también.

Una vez que hayas terminado la implementación de las tres funciones de arriba, deberías ser capaz de ejecutar `mask.py` para predecir palabras enmascaradas y generar diagramas de atención. La segunda parte de este proyecto es analizar esos diagramas de atención para las oraciones de su elección para hacer inferencias sobre el papel que juegan las cabezas de atención específicas en el proceso de comprensión del lenguaje. Rellenarás tu análisis en analisis.md.

- Completar los TODOs en el analysis.md.
  - Usted debe describir al menos dos cabezas de atención para las que usted ha identificado alguna relación entre las palabras que la cabeza de atención parece haber aprendido. En cada caso, escriba una oración o dos describiendo lo que la cabeza parece estar prestando atención y da al menos dos frases de ejemplo que usted ha ingresado en el modelo para llegar a su conclusión.
  - La sección 'Explicación del problema' de esta especificación del proyecto incluye dos ejemplos para usted: Capa 3, Cabeza 10 donde los tokens parecen prestar

atención a los tokens que los siguen; y Capa 4, Cabeza 11, donde los adverbios parecen prestar atención a los verbos que modifican. Los aspectos del lenguaje que usted identifica deben ser diferentes de estos dos.

- Las cabezas de atención pueden ser ruidosas, así que no siempre tendrán interpretaciones humanas claras. A veces pueden atender más que la relación que describe, y a veces no identificarán la relación que describe para cada frase. Está bien. El objetivo aquí es hacer inferencias sobre la atención basada en nuestra intuición humana para el lenguaje, no necesariamente para identificar exactamente cuál es el papel de cada cabeza de atención.
- Puedes buscar cualquier relación entre las palabras que te interesen. Si busca ideas, podría considerar cualquiera de las siguientes: la relación entre los verbos y sus objetos directos, preposiciones, pronombres, adjetivos, determinantes o tokens prestando atención a los tokens que les preceden.

Recomendaciones

Cuando analiza los diagramas de atención, usted encontrará a menudo que muchos tokens en muchas cabezas de atención le prestan fuertemente atención al token [**SEP**] o [**CLS**]. Esto puede ocurrir en los casos en que no hay una buena palabra a la que prestar atención en una cabeza de atención dada.

Problema 4 - Dígitos (MNIST)

Objetivo

Escriba una IA para clasificar dígitos del dataset MNIST.

Introducción

Este proyecto será una introducción a machine learning; usted construirá una red neuronal para clasificar dígitos, y más. El código necesario se encuentra en `problema4-mnist.zip`.

Archivos a editar:

- `models.py` Modelos de perceptrón y redes neuralles para una variedad de aplicaciones.

Archivos a revisar:

- `nn.py` Mini librería de redes neuronales.

Archivos de soporte que se pueden ignorar:

- `autograder.py` Corrector.
- `backend.py` Código de backend para varias tareas de aprendizaje automático.
- `data` Conjuntos de datos para la clasificación de dígitos y la identificación de idiomas.

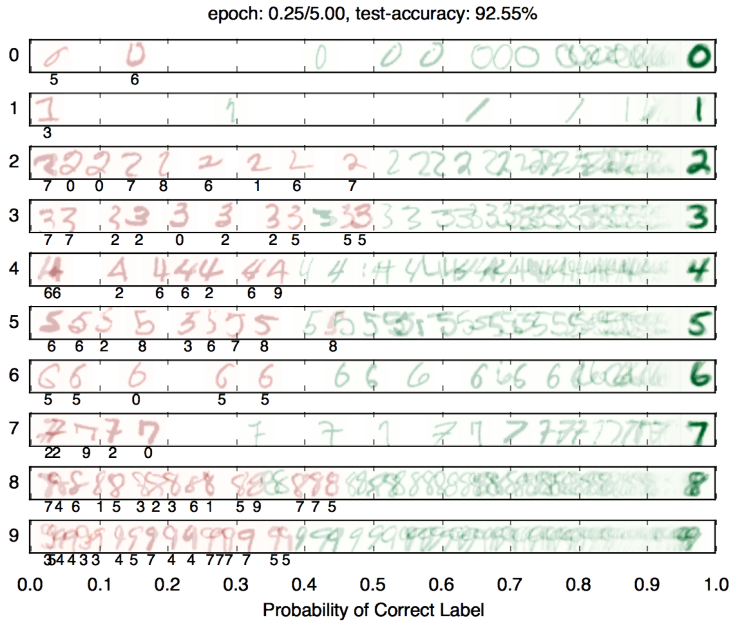

Figura 5

Archivos para Editar: Completaras porciones de `models.py` durante la tarea. Una vez que haya completado la asignación, suba los archivos al repositorio. Por favor no cambie los otros archivos.

Evaluación: Su código será autocorregido. Por favor, no cambie los nombres de ninguna función o clases proporcionadas dentro del código, o causará estragos en el autocorrector. Sin embargo, la corrección de su implementación, no lo que muestre el autograder, es lo que vale.

Instalación

Si ejecutas lo siguiente y se ve la ventana de abajo donde aparece arriba un línea segmentada que gira en un círculo, puede saltar esta sección. Usted debe utilizar el entorno de conda para esto ya que conda viene con las librerías que necesitamos.

`python autograder.py --check-dependencies`
Para este proyecto, tendrá que instalar las siguientes librerías:

- Numpy, que proporciona soporte para grandes matrices multidimensionales rápidas de computar.
- matplotlib, una biblioteca de gráficos en 2D.

```
pip install numpy
pip install matplotlib
```

No usará estas librerías directamente, pero se requieren para ejecutar el código proporcionado y el autocorrector.

Si su configuración es diferente, puede referirse a las instrucciones de instalación de **numpy** y **matplotlib**. Puedes usar cualquiera de los dos pip o conda para la instalación de los paquetes; pip funciona tanto dentro como fuera de los entornos de conda.

Después de instalar, pruebe con un chequeo de dependencias.

Código (Parte 1)

Para este proyecto, se le ha proporcionado una mini-librería de red neuronal (`nn.py`) y una recopilación de conjuntos de datos (`backend.py`).

La librería en `nn.py` define una colección de objetos de nodo. Cada nodo representa un número real o una matriz de números reales. Las operaciones en los objetos del nodo están optimizadas para funcionar más rápido que usando tipos integrados de Python (como las listas).

Estos son algunos de los tipos de nodos proporcionados:

- **nn.Constant** representa una matriz de matriz (2D) de números de punto flotante. Normalmente se utiliza para representar características de entrada o salidas/etiquetas objetivos. Las instancias de este tipo le serán proporcionadas por otras funciones de la API; no necesitará construirlas directamente.
- **nn.Parameter** representa un parámetro entrenable de una red perceptron o neuronal.
- **nn.DotProduct** calcula un producto punto entre sus entradas.

Funciones adicionales:
nn.as_scalar puede extraer un número de punto flotante de Python de un nodo. Cuando entrene a un perceptron o red neuronal, se le pasará un objeto de conjunto de datos. Puede recuperar conjuntos de ejemplos de entrenamiento llamando *dataset.iterate_once(**batch_size**)*:

```
for x, y in dataset.iterate_once(batch_size):
...
```

Por ejemplo, vamos a extraer un lote de talla 1 (es decir, un solo ejemplo de entrenamiento) de los datos de entrenamiento de perceptron:

```
>>> batch_size = 1
>>> for x, y in dataset.iterate_once(batch_size):
...     print(x)
...     print(y)
...     break
```

```
...
<Constant shape=1x3 at 0x11a8856a0>
<Constant shape=1x1 at 0x11a89efd0>
```

Las características de entrada **x** y la etiqueta correcta **y** se proporcionan en forma de nodos **nn.Constant**. La forma de **x** será $[batch\_size \times num\_features]$, y la forma de **y** es $[batch\_size \times num\_outputs]$. Así que, cada fila de x es un/a punto/muestra, y una columna es la misma característica de algunas muestras. Aquí hay un ejemplo de la computación de un producto punto de **x** con él mismo, primero como un nodo y luego como número de Python.

```
>>> nn.DotProduct(x, x)
<DotProduct shape=1x1 at 0x11a89edd8>
>>> nn.as_scalar(nn.DotProduct(x, x))
1.9756581717465536
```

Finalmente, aquí hay algunas formulaciones de multiplicación matricial (puedes hacer algunos ejemplos a mano para verificar esto). Sea **A** una matriz de $m \times n$ y **B** una matriz $n \times p$ la multiplicación funciona como se muestra en la figura 6.

$$\mathbf{AB} = \begin{bmatrix} \vec{A}_0^T \\ \vec{A}_1^T \\ \cdots \\ \vec{A}_{m-1}^T \end{bmatrix} \mathbf{B} = \begin{bmatrix} \vec{A}_0^T\mathbf{B} \\ \vec{A}_1^T\mathbf{B} \\ \cdots \\ \vec{A}_{m-1}^T\mathbf{B} \end{bmatrix} \qquad \mathbf{AB} = \mathbf{A}\begin{bmatrix} \vec{B}_0 & \vec{B}_1 & \cdots & \vec{B}_{p-1} \end{bmatrix} = \begin{bmatrix} \mathbf{A}\vec{B}_0 & \mathbf{A}\vec{B}_1 & \cdots & \mathbf{A}\vec{B}_{p-1} \end{bmatrix}$$

Figura 6: Producto matricial.

- Como una comprobación de sanidad, las dimensiones son lo que esperamos que sean, y la dimensión interior de n se conserva para cualquier multiplicación de matriz restante.
- Esto es útil para ver lo que sucede cuando multiplicamos una matriz de entrada X por una matriz de peso W, solo estamos multiplicando cada muestra una a la vez por toda la matriz de pesos a través de la primera formulación. Dentro de cada producto entre muestra y pesos, sólo estamos consiguiendo diferentes combinaciones lineales de la muestra para ir a cada columna de resultados a través de la segunda formulación. Tenga en cuenta que mientras el las dimensiones concuerden, A puede ser un vector fila y B un vector columna.

Pregunta 1: Perceptrón

Antes de empezar esta parte, asegúrese de tener `numpy` y `matplotlib` instalados.

En esta parte, implementarás un perceptrón binario. Su tarea será completar la implementación de la clase **PerceptronModel** en `models.py`.

Para el perceptrón, las etiquetas de salida serán **1** o **-1**, lo que significa que los puntos de datos **(x, y)** del dataset tendrán guardados en **y** nodos de **nn.Constant** que contengan

1 o **-1** como valores posibles.

Ya hemos inicializado los pesos del perceptrón **self.w** para ser un nodo de parámetro de $1 \times dimensions$. El código proporcionado incluirá un valor de sesgo(bias) adentro de **x** cuando sea necesario, por lo que no necesitará un parámetro separado para el sesgo(bias).

Sus tareas son:

- Implementar el método *run(**self, x**)*. Esto debe calcular el producto de punto del vector de peso almacenado y la entrada dada, devolviendo un objeto **nn.DotProduct**.
- Implementar *get_prediction(**self, x**)*, la cual debe devolver **1** si el producto punto no negativo (es positivo o 0) y sino **-1**.
- Escribe el método *train(**self**)*. Este debe iterar sobre el conjunto de datos (dataset) y hacer actualizaciones en los pesos con los ejemplos que se clasifican erróneamente. Utilice el método *update* de la clase **nn.Parameter** para actualizar los pesos. Cuando se completa un pase entero por encima del conjunto de datos sin cometer ningún error, se ha logrado una precisión de entrenamiento del 100 %, y el entrenamiento puede terminar.
- En este proyecto, la única manera de cambiar el valor de un parámetro es llamando *parameter.update(**direction, multiplier**)*, que realizará la actualización de los pesos:

$$weights \leftarrow weights + direction \times multiplier$$

El argumento **dirección** es un nodo con el mismo tamaño que el parametro, u el argumento **multiplier** es un escalar de Python. Adicionalmente, usa *iterate_once* para iterar por el dataset, ver la sección anterior para ver como se usa.

Para probar tu implementación, usa el autocorrector. Ten en cuenta que deberia tomara como mucho 20 segundos en su ejecución aproximadamente

```
python autograder.py -q q1
```

Recomendaciones para redes neuronales

En lo que queda del proyecto vamos a trabajar con los siguientes modelos:

- Pregunta 2: Regresión no lineal
- Pregunta 3: Clasificación de dígitos escritos.
- Pregunta 4: Identificación de lenguaje

Construcción de reder neuronales

A lo largo de la aplicación del proyecto, usted usará el marco proporcionado en nn.py para crear redes neuronales para resolver una variedad de problemas de aprendizaje automático. Una simple red neuronal tiene capas lineales, donde cada capa lineal realiza una operación lineal (al igual que el perceptron). Las capas lineales están separadas por una no linealidad, lo que permite a la red aproximar las funciones generales. Utilice la operación ReLU para nuestra no linealidad, definida como $relu(x) = max(x, 0)$. Por ejemplo, una red con una sola capa oculta/ dos capas lineales para mapear un vector **x** de fila de entrada a un vector salida $F(x)$ sería dado por la función:

$$F(x) = relu(x \cdot W_1 + b_1) \cdot W_2 + b_2$$

donde tenemos matrices de parámetros W_1 y W_2 y vectores de parámetros b_1 y b_2 para aprender durante la ejecución del algoritmo descenso de gradiente (gradient descent). W_1 será una matriz de $i \times h$, donde i sera la dimension de nuestros vectores de entrada x y h es el tamaño de la capa oculta. b_1 será un vector de tamaño h. Somos libres de elegir cualquier valor que queramos para el tamaño de la capa oculta (sólo tendremos que asegurarnos de que las dimensiones de las otras matrices y vectores están de acuerdo para que podamos realizar las operaciones). El uso de un tamaño oculto más grande suele hacer que la red sea más potente (capaz de encajar más datos de entrenamiento), pero puede hacer la red más difícil de entrenar (ya que añade más parámetros a todas las matrices y vectores que necesitamos aprender), lo puede llevar a un exceso de ajuste en los datos de entrenamiento particulares (overfitting).

Podedmos también crear redes neuronales mas profundas añadiendo mas capas, por ejemplo red de tres capas lineales:

$$\hat{y} = f(x) = relu(relu(x \cdot W_1 + b_1) \cdot W_2 + b_2) \cdot W_3 + b_3$$

O, podemos descomponer lo anterior y explícitamente tener en cuenta las 2 capas ocultas:

$$h_1 = f_1(x) = relu(x \cdot W_1 + b_1)$$
$$h_2 = f_2(h_1) = relu(h_1 \cdot W_2 + b_2)$$
$$\hat{y} = f_3(h_2) = h_2 \cdot W_3 + b_3$$

Tenga en cuenta que no tenemos un *relu* al final porque queremos ser capaces de producir números negativos, y porque el punto de tener una función *relu* en primer lugar es tener transformaciones no lineales, y hacer que la salida sea una transformación lineal afina de algún intermedio no lineal puede ser lo sensato.

Conjuntos/lotes

Para la eficiencia, se le exigirá procesar todo por lotes de datos en lugar de un solo ejemplo a la vez. Esto significa que en lugar de un solo vector **x** de entrada de tamaño i, se le presentará un lote de entradas b representado como una matriz X de dimensiones $b \times i$. Se le proporciona un ejemplo de regresión lineal para demostrar cómo se puede implementar una capa lineal configurada por lotes.

Aleatoriedad

Los parámetros de su red neuronal se inicializarán al azar, y los datos en algunas tareas se presentarán en orden mezclado. Debido a esta aleatoriedad, es posible que usted todavía va a fallar ocasionalmente algunas tareas incluso con una arquitectura fuerte - este es el problema de la optimización local. Esto debería suceder muy raramente, aunque al probar su código falla el autocorrector dos veces seguidas para una pregunta, usted debe explorar otras arquitecturas.

Diseño de arquitecturas de redes neuronales

Diseñar redes neuronales puede tomar bastante ensayo y error. Estos son algunos consejos para ayudarle en el camino:

- Sé sistemático. Mantén un registro de cada arquitectura que haya probado, cuáles eran los hiperparámetros (tamaños de capas, velocidad de aprendizaje, etc.) y cuál era el rendimiento resultante. A medida que intentas más cosas, puedes empezar a ver patrones sobre qué parámetros importan. Si encuentras un error en tu código, asegúrese de cruzar los resultados pasados que no son válidos debido al error.
- Comience con una red poco profunda (sólo una capa oculta, es decir, una no-linealidad). Las redes más profundas tienen combinaciones de hiperparámetros exponencialmente más complejas, y si incluso un solo esta mal se puede arruinar su rendimiento. Utiliza la pequeña red para encontrar una buena tasa de aprendizaje y tamaño de capa; después puedes considerar agregar más capas de tamaño similar.
- Si su tasa de aprendizaje está errada, ninguna de sus otras opciones de hiperparámetro importa. Puedes tomar un modelo de red neuronal de última generación de un trabajo de investigación, y cambiar la tasa de aprendizaje de tal manera que no funciona mejor que elegir al azar. Una tasa de aprendizaje demasiado baja resultará en el aprendizaje del modelo demasiado lentamente, y una tasa de aprendizaje demasiado alta puede causar pérdidas y divergencia al infinito. Comienza probando diferentes tasas de aprendizaje mientras observas cómo disminuye la pérdida con el tiempo.
- Los lotes más pequeños requieren tasas de aprendizaje más bajas. Cuando experimente con diferentes tamaños de lotes, tenga en cuenta que la mejor tasa de aprendizaje puede ser diferente dependiendo del tamaño del lote.

- Abstente de hacer la red demasiado ancha (tamaños de capa ocultas demasiado grandes) Si sigues haciendo que la red más ancha la presición disminuirá gradualmente, y el tiempo de cálculo aumentará cuadráticamente en el tamaño de la capa. Es probable que te rindas debido a la lentitud excesiva mucho antes de que la precisión caiga demasiado. El autocorrector completo para todas las partes del proyecto tarda 2-12 minutos en ejecutarse con las soluciones optimas; si su código está tardando mucho más tiempo, debe comprobarlo para obtener eficiencia.
- Si tu modelo está devolviendo `Infinity` o `NaN`, su tasa de aprendizaje es probablemente demasiado alta para su arquitectura actual.
- Valores recomendados para sus hiperparametros:
  - Tamaños de capas ocultas: entre 10 y 400.
  - Tamaño de lotes(batch): entre 1 y el tamaño del conjunto de datos. Para las preguntas 2 y 3 se requiere que el tamaño total del dataset sea divisible por el tamaño de lote.
  - Tasa de aprendizaje: entre 0,001 y 1,0.
  - Número de capas ocultas: entre 1 y 3.

Código (Parte 2)

Aquí hay una lista completa de nodos disponibles en `nn.py`. Usted hará uso de estos en las partes restantes de la asignación:

- **nn.Constant** representa una matriz de matriz (2D) de números de punto flotante. Normalmente se utiliza para representar características de entrada o salidas/etiquetas di-objetivos. Las instancias de este tipo le serán proporcionadas por otras funciones de la API; no necesitará construirlas directamente.
- **nn.Parameter** representa un parámetro entrenable de un perteptron o red neuronal. Todos los parámetros deben ser bidimensionales. Uso: **nn.Parameter(n, m)** construye un parametro de forma $n \times m$
- **nn.Add** suma matrices elemento a elemento. Uso: **nn.Add(x, y)** acepta dos nodos de forma $batch\_size \times num\_features$ y construye un nodo que también tiene forma $batch\_size \times num\_features$.
- **nn.AddBias** añade un vector de sesgo (bias) a cada vector de características. Nota: emite distribuye automáticamente el *bias* para añadir el mismo vector a cada fila de *features*. Uso: **nn.AddBias(features, bias)** acepta *features* de forma $batch\_size \times num\_features$ y *bias* de forma $1 \times num\_features$, y construye un nodo que tiene forma $batch_size \times num_features$.
- **nn.Linear** aplica una transformación lineal (multiplicación matricial) a la entrada. Uso: **nn.Linear(features, weights)** acepta *features* de forma $batch\_size \times num_input_features$

y *weights* de forma $num\_input\_features \times num\_output\_features$, y construye un nodo que tiene forma $batch\_size \times num\_output\_features$.

- **nn.ReLU** aplica el elemento a elemento la función de Rectificación Lineal Unitaria (RELU) $relu(x) = max(x, 0)$. Esta nolinealidad reemplaza todas las entradas negativas en su entrada con ceros. Uso **nn.ReLU(features)**, que devuelve un nodo con la misma forma que la entrada.

- **nn.SquareLoss** calcula una función de pérdida cuadrática por lote, utilizada para problemas de regresión. Uso: **nn.SquareLoss(a, b)**, dónde a y b ambos tienen forma $batch\_size \times num\_outputs$.

- **nn.SoftmaxLoss** calcula una función de pérdida tipo softmax por lotes, utilizada para problemas de clasificación. Uso: **nn.SoftmaxLoss(logits, labels)**, dónde *logits* y *labels* ambos tienen forma $batch\_size \times num\_classes$. El término 'logits' se refiere a las puntuaciones producidas por un modelo, donde cada entrada puede ser un número real arbitrario. Las etiquetas, sin embargo, deben ser no negativas y cada fila (todos sus elementos) tiene que sumar 1. Asegúrese de no intercambiar el orden de los argumentos.

- No usar **nn.DotProduct** para cualquier modelo que no sea el perceptron.

Los siguientes métodos están disponibles en `nn.py`:

- **nn.gradients** calcula los gradientes de una pérdida con respecto a los parámetros proporcionados. Uso: **nn.gradients(loss,** $[parameter_1, parameter_2, ..., parameter_n]$**)** devolverá una lista $[gradient_1, gradient_2, ..., gradient_n]$, donde cada elemento es un **nn.Constant** que contiene el gradiente de la pérdida con respecto a un parámetro.

- **nn.as_scalar** puede extraer un número de punto flotante de Python que representa la pérdida de un nodo. Esto puede ser útil para determinar cuándo dejar de entrenar. Uso: **nn.as_scalar(node)**, donde el nodo es un nodo de pérdida o tiene forma (1,1).

Los dataset proporcionados también tienen dos métodos adicionales:

- **dataset.iterate_forever(batch_size)** produce una secuencia infinita de lotes de ejemplos.

- **dataset.get_validation_accuracy()** devuelve la exactitud de su modelo en el conjunto de validación. Esto puede ser útil para determinar cuándo dejar de entrenar.

Ejemplo: Regresión lineal

Como ejemplo de cómo funciona el marco de la red neuronal, vamos a entrenar un conjunto de puntos de datos. Iniciaremos con cuatro puntos de los datos de entrenamiento construidos con la función $y = 7x_0 + 8x_1 + 3$. En forma por lotes, nuestros datos son:

Supongamos que los datos se nos proporcionan en forma de nodos **nn.Constant**:

$$\mathbf{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \qquad \mathbf{Y} = \begin{bmatrix} 3 \\ 11 \\ 10 \\ 18 \end{bmatrix}$$

Figura 7: Representacion matricial de 4 puntos de datos que pertenecen a nuestra función

```
>>> x
<Constant shape=4x2 at 0x10a30fe80>
>>> y
<Constant shape=4x1 at 0x10a30fef0>
```

Vamos a construir y entrenar un modelo de la forma $f(X) = x_0 \cdot m_0 + x_1 \cdot m_1 + b$. Si se hace correctamente, deberiamos poder aprender que $m_0 = 7$, $m_1 = 8$ y $b = 3$.

Primero, creamos nuestros parámetros entrenables. En forma de matriz, estos son:

$$\mathbf{M} = \begin{bmatrix} m \\ m \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} b \end{bmatrix}$$

Figura 8: Parametros a entrenar.

Lo que corresponde al siguiente código:

```
m = nn.Parameter(2, 1)
b = nn.Parameter(1, 1)
```

Imprimirlas da:

```
>>> m
<Parameter shape=2x1 at 0x112b8b208>
>>> b
<Parameter shape=1x1 at 0x112b8beb8>
```

A continuación, calculamos nuestras predicciones de su modelo para y:

```
xm = nn.Linear(x, m)
predicted_y = nn.AddBias(xm, b)
```

Nuestro objetivo es que los y valores predichos coincidan con los datos proporcionados. En regresión lineal hacemos esto minimizando la pérdida cuadrática:

$$\mathcal{L} = \frac{1}{2} \sum_{(\mathbf{x},y)} (y - f(\mathbf{x}))^2$$

Figura 9: Formula de perdida cuadrática.

Creamos entonces un nodo de pérdida:

```
loss = nn.SquareLoss(predicted_y, y)
```

En nuestro marco, proporcionamos un método que devolverá los gradientes de la pérdida con respecto a los parámetros:

```
grad_wrt_m, grad_wrt_b = nn.gradients(loss, [m, b])
```

Impresión de los nodos utilizados da:

```
>>> xm
<Linear shape=4x1 at 0x11a869588>
>>> predicted_y
<AddBias shape=4x1 at 0x11c23aa90>
>>> loss
<SquareLoss shape=() at 0x11c23a240>
>>> grad_wrt_m
<Constant shape=2x1 at 0x11a8cb160>
>>> grad_wrt_b
<Constant shape=1x1 at 0x11a8cb588>
```

Podemos usar el método *update* para actualizar nuestros parámetros. Aquí hay una actualización para m, suponiendo que ya hayamos inicializado un variable *multiplier* basada en una tasa de aprendizaje adecuada de nuestra elección:

```
m.update(grad_wrt_m, multiplier)
```

Si también incluimos una actualización para b y añadimos un bucle para realizar repetidamente actualizaciones de gradiente, tendremos el procedimiento de entrenamiento completo para la regresión lineal.

Pregunta 2: Regresion no lineal

Para esta pregunta, entrenará una red neuronal para aproximar $\sin(x)$ en el intervalo $[-2\pi, 2\pi]$.

Usted tendrá que completar la implementación de la clase **RegressionModel** en `models.py`. Para este problema, una arquitectura relativamente simple debe ser suficiente (ver la sección Diseño de arquitecturas de redes neuronales). Uso **nn.SquareLoss** como tu función de pérdida.

Las tareas son:

- Implementar **RegressionModel.__init__** con cualquier inicialización necesaria.
- Implementar **RegressionModel.run** para devolver un nodo de $batch\_size \times 1$ que represente la predicción de su modelo.
- Implementar **RegressionModel.get_loss** que debe devolver una pérdida para las entradas dadas y las salidas previstas.
- Implementar **RegressionModel.train**, que debe entrenar su modelo actualizando pesos basados en el gradiente.

Sólo hay una división del dataset para esta tarea (es decir, sólo hay datos de entrenamiento y no hay datos de validación o conjunto de pruebas). Su implementación idealmente debe obtener una pérdida de 0.02 o mejor, promediado en todos los ejemplos en el conjunto de datos. Puede utilizar la pérdida de entrenamiento para determinar cuándo detener el entrenamiento (utilizar **nn.as_scalar** para convertir un nodo de pérdida a un número de Python). Tenga en cuenta que el modelo debe tardar unos minutos en entrenar.

Arquitectura de red sugerida: Normalmente, tendrías que usar prueba y error para encontrar hiperparametros de trabajo. A continuación se encuentra un conjunto de hiperparámetros que funcionaron para nosotros, pero no dude en experimentar y usar el suyo propio.

- Tamaño de capa oculta 512
- Tamaño del musco 200
- Tasa de aprendizaje 0.05
- Una capa oculta (2 capas lineales en total)

```
python autograder.py -q q2
```

Pregunta 3: Clasificación de digitos

Para esta pregunta, entrenará una red para clasificar los dígitos manuscritos del dataset MNIST.

Cada dígito es de tamaño 28×28 píxeles, cuyos valores se almacenan en un vector de 1×784 números de punto flotante. Cada salida que proporcionamos es un vector de 1×10, que tiene ceros en todas las posiciones, excepto por un uno en la posición correspondiente a la clase correcta del dígito.

Completar la implementación de la clase **DigitClassificationModel** en `models.py`. El valor de retorno de **DigitClassificationModel.run()** debería ser un Nodo de $batch\_size \times 10$ que contiene valores de puntuación, donde los valores más altas indican una mayor probabilidad de un dígito perteneciente a una clase en particular (0-9). Deberías usar **nn.SoftmaxLoss** como tu función de pérdida. No ponga una activación de ReLU en la última capa lineal de la red.

Tanto para esta pregunta como para la Pregunta 4, además de los datos de entrenamiento, también hay datos de validación (validation set) y un conjunto de pruebas (test set). Puedes usar **dataset.get_validation_accuracy()** para calcular la precisión de validación para su modelo, que puede ser útil a la hora de decidir si dejar de entrenar. El conjunto de prueba será utilizado por el autocorrector.

Para que el autocorrector considere esta pregunta, su modelo debe lograr una precisión de al menos 97 % sobre el conjunto de pruebas. Como referencia, la implementacion normal es de 98 % con los datos de validación después de la entrenar alrededor de 5 épocas (epochs). Tenga en cuenta que el autocorrector mide la prueba con el set de test, mientras que usted sólo tiene acceso a la precisión con el set de validación - por lo que si su precisión de validación cumple con el umbral del 97 %, todavía puede fallar la prueba si su precisión con el dataset de prueba no cumple el umbral. Por lo tanto, puede ayudar a establecer un umbral de parada ligeramente más alto en la precisión de validación, como el 97,5 % o el 98 %.

Arquitectura de red sugerida: Normalmente, tendrías que usar prueba y error para encontrar hiperparametros de trabajo. A continuación se encuentra un conjunto de hiperparámetros que funcionaron para nosotros (tenía menos de un minuto entrenando en las máquinas colmenas), pero no dude en experimentar y usar el suyo propio.

- Tamaño de capa oculta 200
- Tamaño de la batch 100
- Tasa de aprendizaje 0.5
- Una capa oculta (2 capas lineales en total)

Para probar su implementación, ejecute el autocorrector

```
python autograder.py -q q3
```

Pregunta 4: Identificación de lenguaje

La identificación del idioma es la tarea de averiguar, dado un texto, en qué idioma se escribe el texto. Por ejemplo, su navegador podría ser capaz de detectar si usted ha visitado una página en un idioma extranjero y ofrece traducirlo para usted. Aquí hay un ejemplo de Firefox (que utiliza una red neuronal para implementar esta característica):

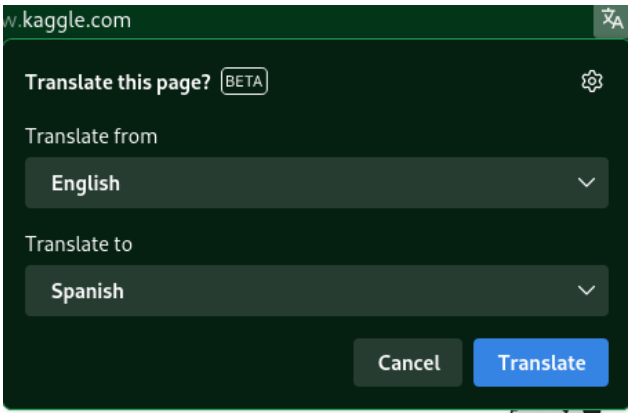

Figura 10: Detección automática de idioma

En este proyecto, vamos a construir un modelo de red neuronal más pequeño que identifique el lenguaje para una palabra a la vez. Nuestro conjunto de datos consta de palabras en cinco idiomas, como la siguiente tabla:

Palabra	Idioma
discussed	English
eternidad	Spanish
itseänne	Finnish
paleis	Dutch
mieszkać	Polish

Diferentes palabras consisten en diferentes números de letras, por lo que nuestro modelo necesita tener una arquitectura que pueda manejar entradas de longitud variable. En lugar de una sola entrada x (como en las preguntas anteriores), tendremos una entrada separada para cada carácter en la palabra: $x_0, x_1, \ldots x_{L-1}$ donde L es la longitud de la palabra. Comenzaremos aplicando una red $f_{initial}$ que es igual que las redes en los problemas anteriores. Acepta su entrada x_0 y calcula un vector de salida h_1 de dimension d:

$$h_1 = f_{initial}(x_0)$$

A continuación, combinaremos la salida del paso anterior con la siguiente letra en la palabra, generando un resumen vectorial de las dos primeras letras de la palabra. Para ello,

aplicaremos una sub-red que acepta una letra y devuelve un estado oculto, pero ahora también depende del estado oculto anterior h_1. Denotamos esta sub-red como f.

$$h_2 = f(h_1, x_1)$$

Este patrón se continúa para todas las letras en la palabra de entrada, donde el estado oculto en cada paso resume todas las letras que la red ha procesado hasta ahora:

$$h_3 = f(h_2, x_2)$$
$$\dots$$

A lo largo de estos cálculos, la función $h = f(\cdot, \cdot)$ es la misma pieza de red neuronal y utiliza los mismos parámetros entrenables. $f_{initial}$ también compartirá algunos de los mismos parámetros que $h = f(\cdot, \cdot)$. De esta manera, los parámetros utilizados al procesar palabras de diferente longitud se comparten. Puede implementar esto iterando sobre las entradas proporcionadas **xs**, donde cada iteración del bucle computa ya sea $f_{initial}$ o f.

La técnica descrita se llama Red Neuronal Recurrente (RNN - Recurrent Neural Network en inglés). A continuación se muestra un diagrama esquemático de la RNN:

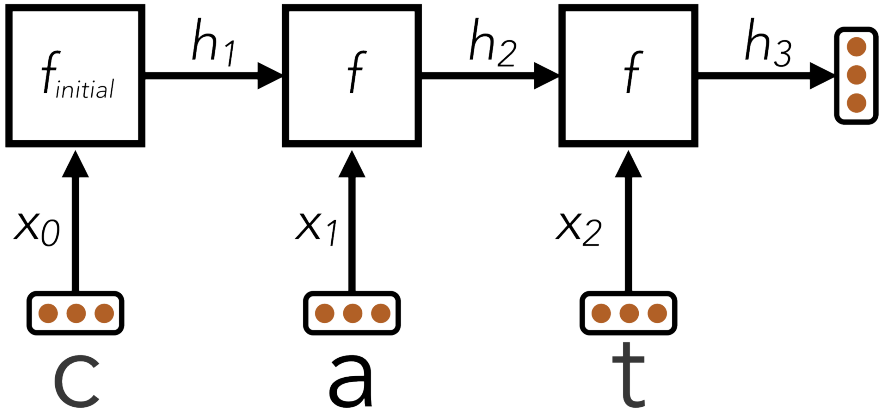

Figura 11: Diagrama de red neuronal recurrente para analisis de palabras.

Aquí, un RNN se utiliza para encodificar la palabra *cat* en un vector de tamaño fijo h_3.

Después de que el RNN haya procesado la longitud completa de la entrada, ha codificado la palabra de entrada de longitud arbitraria en un vector de tamaño fijo h_L , donde L es la longitud de la palabra. Este resumen vectorial de la palabra de entrada ahora se puede alimentar a través de una capa de transformación de salida adicional para generar puntajes de clasificación para la identidad del lenguaje de la palabra.

Conjuntos/lotes

Aunque las ecuaciones anteriores son en términos de una sola palabra, en la práctica debe utilizar lotes de palabras para la eficiencia. Para la simplicidad, nuestro código en el proyecto asegura que todas las palabras dentro de un solo lote tengan la misma longitud. En forma por lotes, un estado oculto h_i se sustituye por la matriz H_i de dimensiones $batch\_size \times d$.

Consejos de diseño

El diseño de la función recurrente $f(\cdot,\cdot)$ es el principal reto para esta tarea. Estos son algunos consejos:

- Comience con una arquitectura $f_{initial}(x)$ de su elección similar a las preguntas anteriores, siempre y cuando tenga al menos una no linealidad.
- Usted debe utilizar el siguiente método de construcción de $f(\cdot,\cdot)$ dado $f_{initial}(x)$. La primera capa de transformación de $f_{initial}$ comenzará multiplicando el vector x_0 por alguna matriz de peso W_x para producir $z_0 = x_0 \cdot W_x$. Para las letras posteriores, usted debe reemplazar este cálculo con $z_i = x_i \cdot W_x + h_i \cdot W_{hidden}$ usando una operación **nn.Add**. En otras palabras, usted debe reemplazar un cómputo de la forma **z0 = nn.Linear(x, W)** con un cómputo de la forma **z = nn.Add(nn.Linear(x, W), nn.Linear(h, W_hidden))**.
- Si se hace correctamente, la función resultante $f(x_i, h_i) = g(z_i) = g(z_{x_i}, h_i)$ no será lineal tanto en x como en h.
- El tamaño oculto d debería ser suficientemente grande.
- Comience con una red superficial para f, y encuentre buenos valores para el tamaño oculto y la tasa de aprendizaje antes de hacer la red más profunda. Si empiezas con una red profunda de inmediato tendrás exponencialmente más combinaciones de hiperparámetros, y conseguir cualquier hiperparámetro incorrecto puede causar que tu rendimiento sufra dramáticamente.

Tarea

Completar la implementación de la clase **LanguageIDModel**.

Para recibir puntos completos sobre este problema por el autocorrector, su arquitectura debe ser capaz de lograr una precisión de al menos 81 % en el conjunto de pruebas.

Para probar su implementación, ejecute el autocorrector:

```
python autograder.py -q q3
```

Nota: ste conjunto de datos se generó mediante el procesamiento automatizado de textos. Puede contener errores. Tampoco se ha filtrado para la profanidad. Sin embargo, nuestra implementación de referencia todavía puede clasificar correctamente más del 89 % del conjunto de validación a pesar de las limitaciones de los datos. La implementación usada de referencia toma 10-20 épocas para entrenar.