Tensores en Pytorch

Diego Andrade Canosa

Roberto López Castro

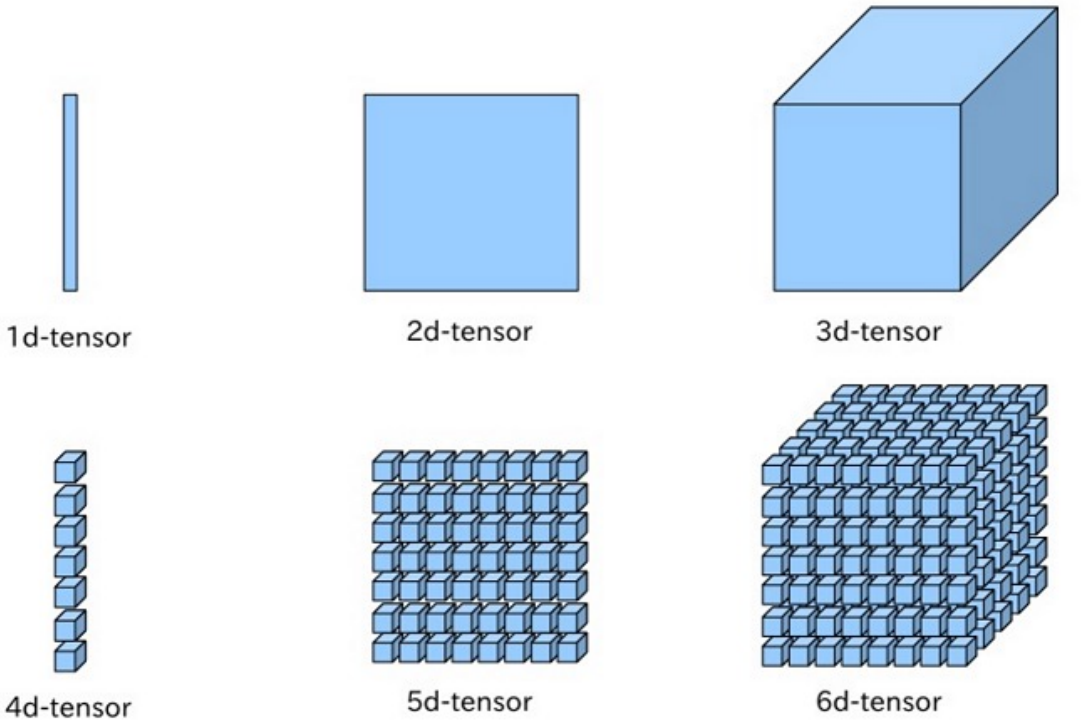
Índice

- Introducción
- Tensores en Pytorch
- Variables y gradients
  - Autograd

Introducción

- En matemáticas, un tensor es un artefacto algebraico multidimensional que permite almacenar información numérica
- Se usan en:
  - Matemáticas
  - Física
  - Química
  - Deep Learning!!
  - ...
- Características básicas de un tensor:
  - Rank: Número de dimensiones
  - Shape: Tamaño de cada dimensión
  - Tipo de datos

Introducción



Fuente: https://leonardoaraujosantos.gitbook.io/opencl/

citic

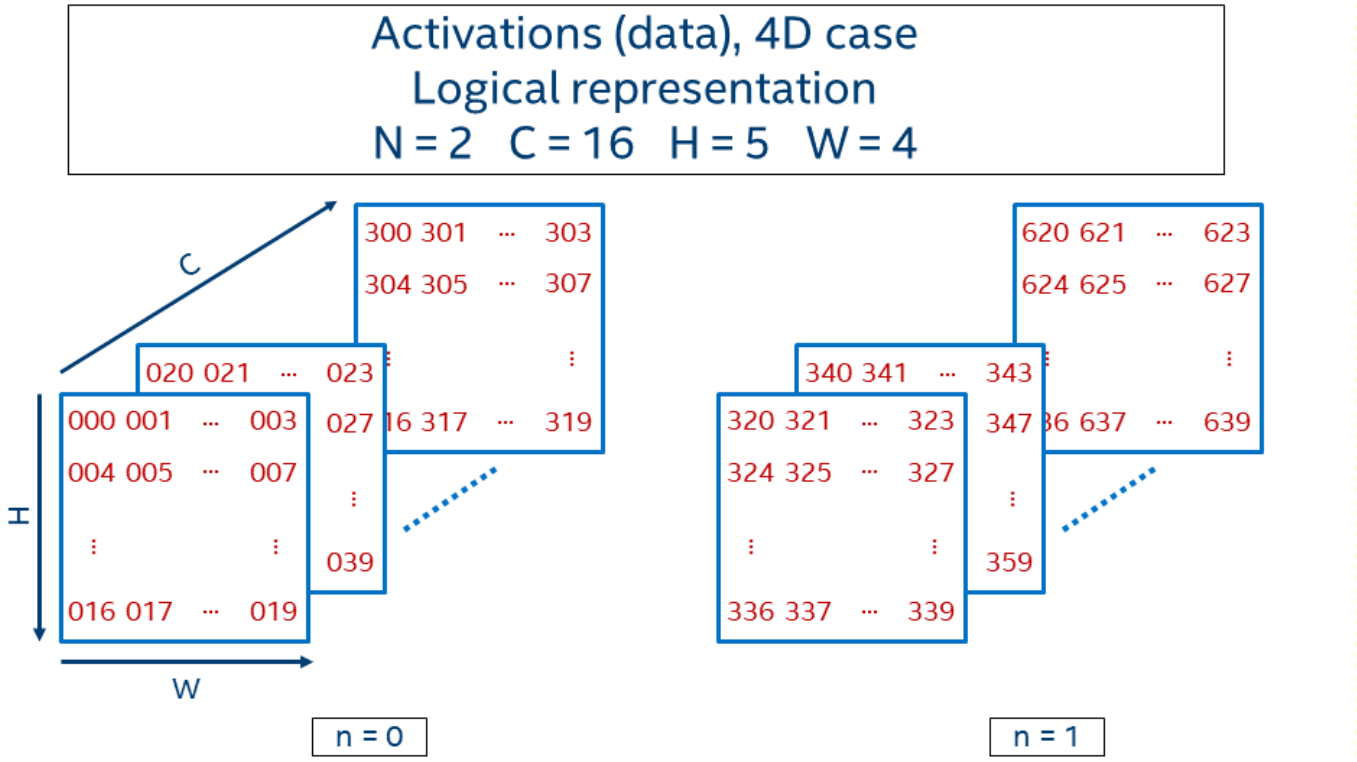
Introducción

- En Deep Learning se usan los tensores para almacenar
  - Entradas
  - Parámetros aprendibles:
    - Pesos
    - Bias
  - Datos auxiliares:
    - Gradientes
    - Activaciones
    - Pérdidas (*loss*)
    - ...
  - Salidas

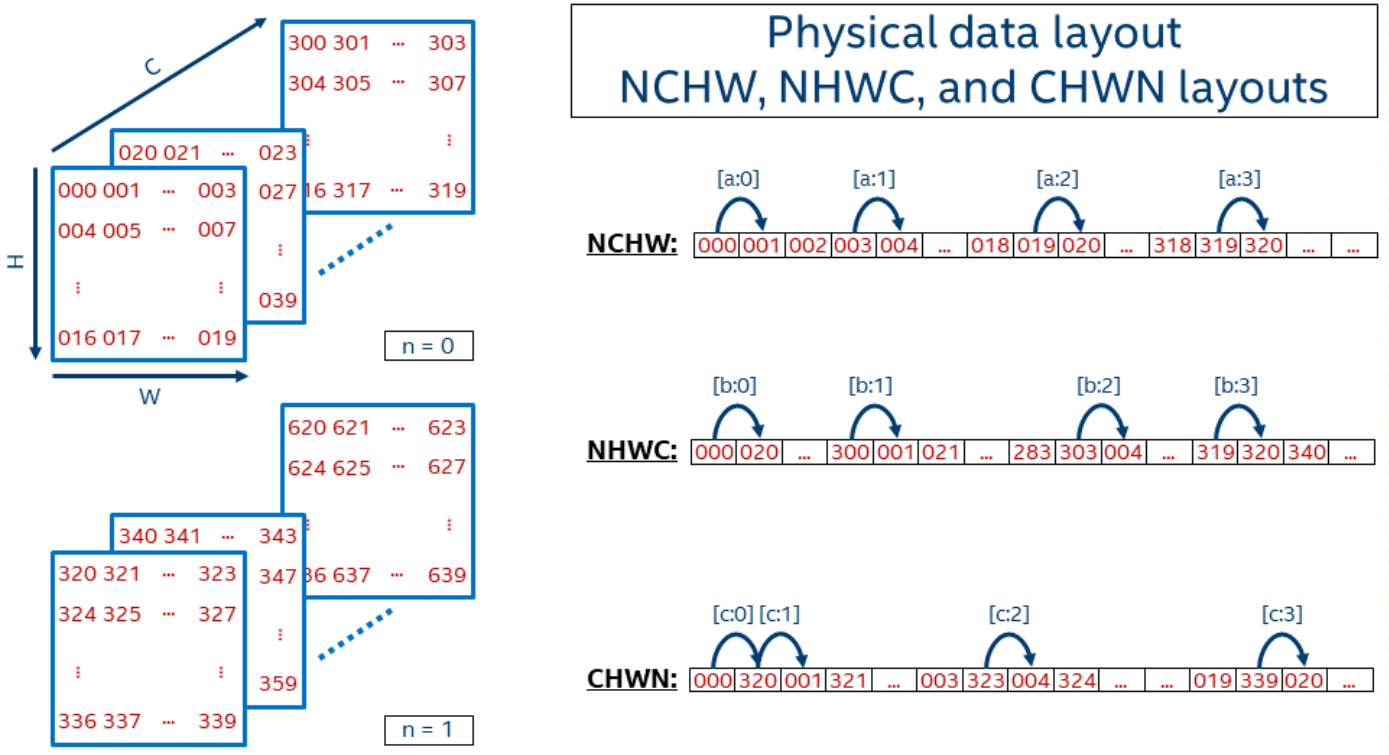
Codificación de entradas usando tensores

- Reconocimiento de imágenes
  - La imagen se proporciona como un tensor tridimensional (C (canales) x H (altura) x W (anchura)
  - Se suele incluir una cuarta dimensión N que tiene el número de *samples* en un batch
    - El procesamiento batch hace que los datos de entrada se procesen en conjuntos (de imágenes en este caso)
    - **RECUERDA:** El cálculo de la función de pérdida, los consiguientes gradientes y su aplicación se realiza al haber procesado todas las entradas de un batch
  - Los formatos disponibles dependen del orden de estas 4 dimensiones (NCHW es el más común en Pytorch, channel first y el preferido por GPUs de Nvidia)

Ejemplo: NCHW



Ejemplo: CHWN



citic

Formatos en Pytorch

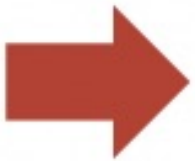
- NHWC: channel last (memory_format=torch.channel_last)
- NCHW: channel first (memory_format=torch.contiguous_format)

Codificación de entradas usando tensores

- Lenguaje:
  - El texto de entrada se somete a un proceso de tokenización por los que se divide el texto en palabras, caracteres, o grupos de caracteres o palabras
    - Cada token se representa mediante un valor numérico
    - Se codifica la secuencia de tokens. Dos opciones principales:
      - One-hot encoding
        - Se usa un vector disperso del mismo tamaño que el número de tokens distintos
        - Se pone un **1** sólo en la posición del vector que pertenece al dato del token actual; **0** en las demás posiciones
      - Word embedding
        - Se le asigna un vector de valores a cada posible token
        - Se codifica el texto como un tensor que contiene estos vectores para una secuencia de tokens
        - Palabras similares serán representadas por valores similares

Ejemplo: one-hot embedding

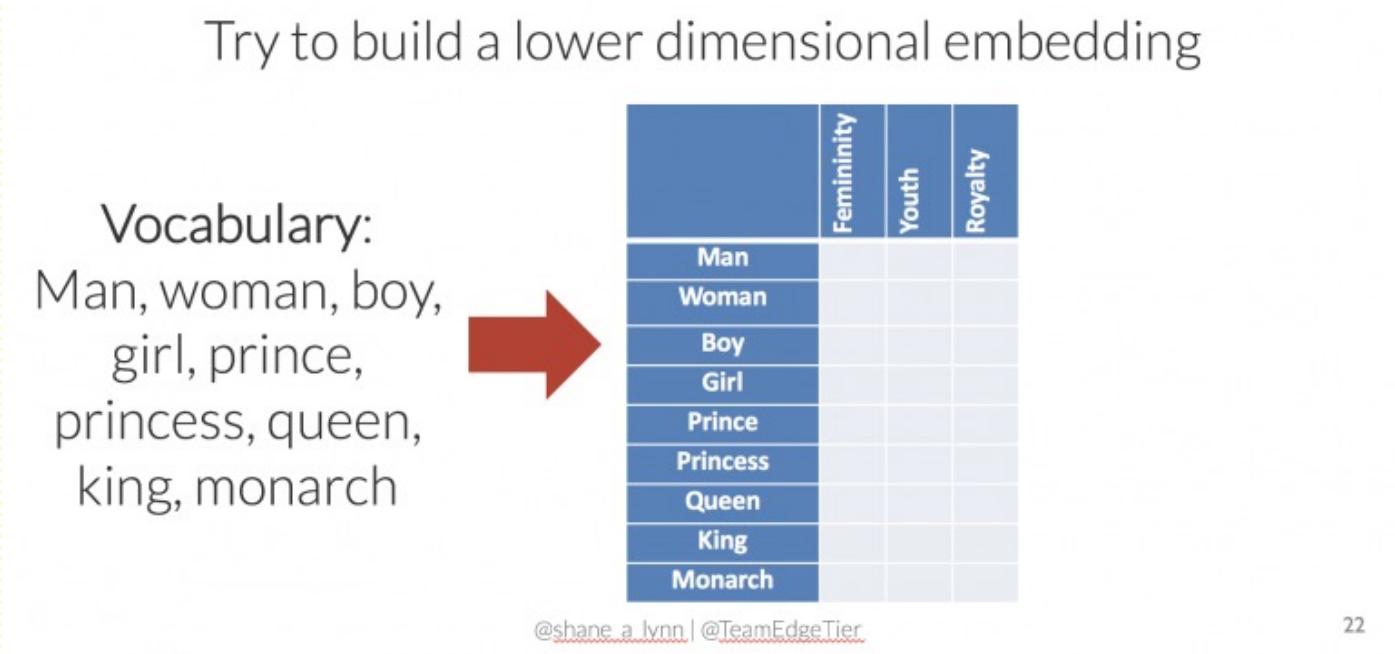
Vocabulary:
Man, woman, boy, girl, prince, princess, queen, king, monarch



	1	2	3	4	5	6	7	8	9
man	1	0	0	0	0	0	0	0	0
woman	0	1	0	0	0	0	0	0	0
boy	0	0	1	0	0	0	0	0	0
girl	0	0	0	1	0	0	0	0	0
prince	0	0	0	0	1	0	0	0	0
princess	0	0	0	0	0	1	0	0	0
queen	0	0	0	0	0	0	1	0	0
king	0	0	0	0	0	0	0	1	0
monarch	0	0	0	0	0	0	0	0	1

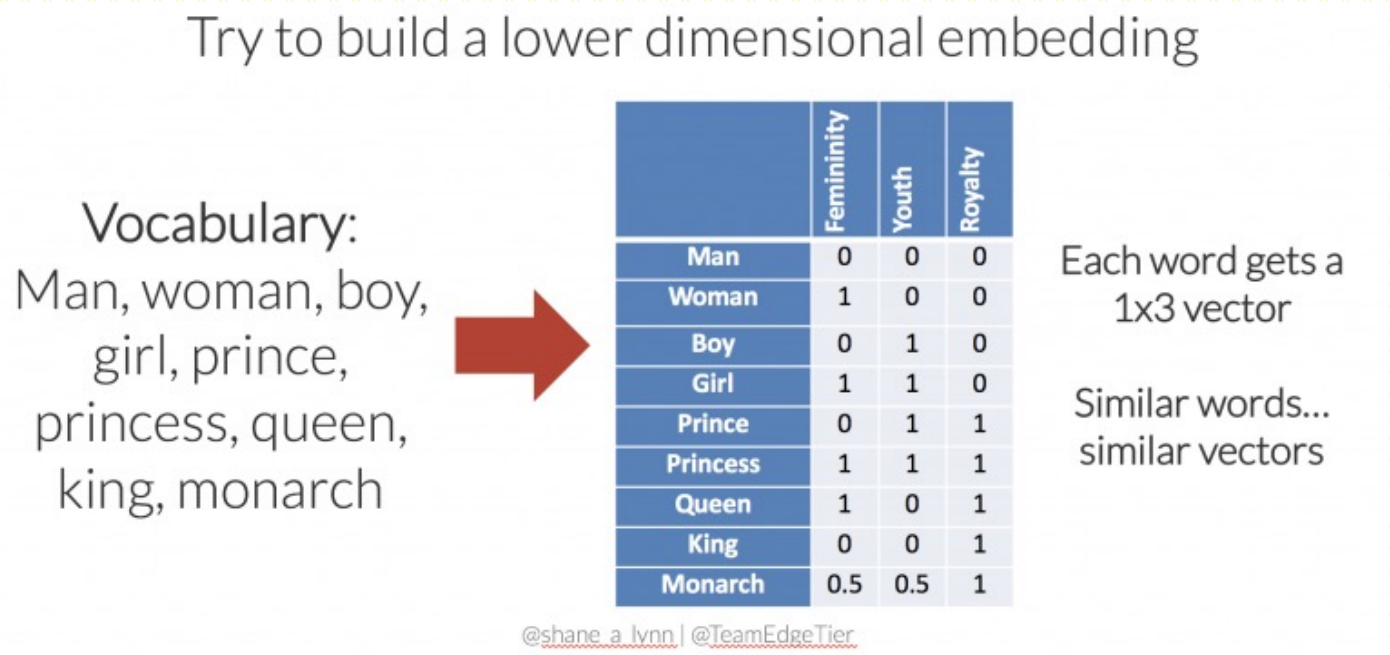
Each word gets a 1x9 vector representation

Ejemplo: word embedding



citic

Ejemplo: word embedding



citic

Codificación de parámetros aprendibles

- La representación interna de pesos y bias depende del tipo de modelo
  - Perceptrón multicapa:
    - Pesos y Bias se representan en tensores diferentes
  - Redes convolucionales:
    - Pesos y bias se representan en tensores diferentes
    - Suelen tener 2 dimensiones (altura y anchura)
    - Existe un tercera dimensión para indexar la información respecto a cada kernel (o filtro) que se suele asociar con las características identificadas internamente por el modelo
  - Redes Recurrentes
  - Transformers

Codificación de parámetros aprendibles

- La representación interna de pesos y bias depende del tipo de modelo
  - Perceptrón multicapa:
  - Redes convolucionales:
  - Redes Recurrentes
    - Los tensores representan las conexiones recurrentes y los estados ocultos
    - Las conexiones recurrentes son los pesos que conectan los estados ocultos a través de diferentes instantes de tiempo
      - (Batch,Nº de neuronas,Nº de capas ocultas)
  - Transformers
    - Attention weights: (num_heads, sequence_length, sequence_length)
    - Hidden state tensors: (num_layers, batch_size, sequence_length, hidden_size)
    - Positional encoding: (sequence_length, embedding_dimension)
    - Feed-Forward Network Tensors: (num_layers, batch_size, sequence_length, hidden_size)

Codificación de datos de salida

- En problemas de predicción del siguiente valor de una serie numérica: la salida podría ser un único valor numérico
- En problemas de clasificación los logits son las puntuaciones no normalizadas asignadas a cada categoría por nuestro modelo
  - Clasificadores multiclase: la salida podría ser un vector con tantos elementos como categorías, y cada elemento contiene la probabilidad asignada a esa categoría
  - Clasificadores binarios: la salida es la probabilidad de que la entrada pertenezca a la clase positiva

Tensores en Pytorch

- Similares, e interoperables, con los ndarrays de numpy
- Sus métodos cuelgan directamente del paquete principal de pytorch (torch)
- Métodos para distintos tipos de operaciones
  - Inicialización
  - Manipulación
  - Operación

Tensores en Pytorch: Creación e inicialización

- Directamente desde los datos

```
data = [[1, 2],[3, 4]]
x_data = torch.tensor(data)
```

- Desde un array de numpy

```
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
```

```
tensor([[1, 2],
        [3, 4]])
```

- A partir de la forma de otro tensor

```
x_ones = torch.ones_like(x_data)
x_rand = torch.rand_like(x_data,dtype=torch.float)
```

Tensores en Pytorch: Creación e inicialización

- Con unos valores determinados (1^s o 0^s) o aleatorios

```
shape = (2,3,)
rand_tensor = torch.rand(shape)
ones_tensor = torch.ones(shape)
zeros_tensor = torch.zeros(shape)
```

```
tensor([[0.3501, 0.6315, 0.5309],
        [0.1929, 0.4399, 0.3236]])
tensor([[1., 1., 1.],
        [1., 1., 1.]])
tensor([[0., 0., 0.],
        [0., 0., 0.]])
```

Tensores en Pytorch: Atributos

- Un tensor tiene 3 atributos básicos en Pytorch: shape, dtype y device

```
tensor = torch.rand(3,4)

print(f"Shape of tensor: {tensor.shape}")
print(f"Datatype of tensor: {tensor.dtype}")
print(f"Device tensor is stored on: {tensor.device}")
```

```
Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu
```

Tensores en Pytorch: Manipulación

- Dado un tensor **a** de tamaño (2,4,4), podemos
  - Hacer slicing siguiendo una notación estándar

a

```
tensor([[[0.22, 0.96, 0.68, 0.88],
         [0.09, 0.08, 0.57, 0.79],
         [0.09, 0.24, 0.64, 0.26],
         [0.82, 0.07, 0.79, 0.70]],

        [[0.23, 0.46, 0.71, 0.59],
         [0.36, 0.26, 0.43, 0.08],
         [0.67, 0.16, 0.29, 0.28],
         [0.76, 0.86, 0.05, 0.19]]])
```

a[1]

```
tensor([[0.23, 0.46, 0.71, 0.59],
        [0.36, 0.26, 0.43, 0.08],
        [0.67, 0.16, 0.29, 0.28],
        [0.76, 0.86, 0.05, 0.19]])
```

a[1,2:3]

```
tensor([[0.67, 0.16, 0.29, 0.28],
        [0.76, 0.86, 0.05, 0.19]])
```

Tensores en Pytorch: Manipulación

- Cambiar la forma de un tensor con el comando reshape

a

```
tensor([[[0.43, 0.56, 0.42, 0.77],
         [0.55, 0.60, 0.97, 0.73],
         [0.68, 0.77, 0.25, 0.67],
         [0.41, 0.25, 0.57, 0.81]],

        [[0.57, 0.19, 0.58, 0.48],
         [0.12, 0.35, 0.02, 0.94],
         [0.09, 0.28, 0.69, 0.35],
         [0.04, 0.83, 0.40, 0.46]]])
```

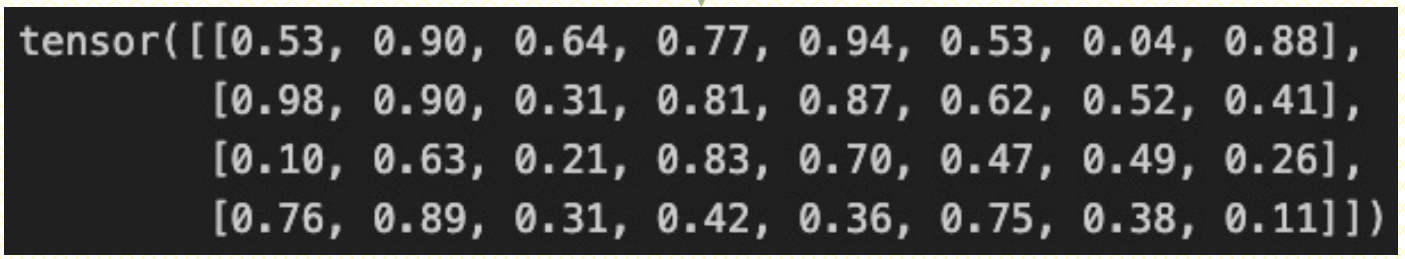
a.reshape(8,4)

```
tensor([[0.43, 0.56, 0.42, 0.77],
        [0.55, 0.60, 0.97, 0.73],
        [0.68, 0.77, 0.25, 0.67],
        [0.41, 0.25, 0.57, 0.81],
        [0.57, 0.19, 0.58, 0.48],
        [0.12, 0.35, 0.02, 0.94],
        [0.09, 0.28, 0.69, 0.35],
        [0.04, 0.83, 0.40, 0.46]])
```

Tensores en Pytorch: Manipulación

Transponer el tensor con el comando transpose

```
tensor([[0.43, 0.56, 0.42, 0.77],
        [0.55, 0.60, 0.97, 0.73],
        [0.68, 0.77, 0.25, 0.67],
        [0.41, 0.25, 0.57, 0.81],
        [0.57, 0.19, 0.58, 0.48],
        [0.12, 0.35, 0.02, 0.94],
        [0.09, 0.28, 0.69, 0.35],
        [0.04, 0.83, 0.40, 0.46]])
```



Tensores en Pytorch: Manipulación

Combinar varios tensores

Tensores a y b

```
tensor([[0.60, 0.61],
        [0.95, 1.00]])
tensor([[0.98, 0.25],
        [0.78, 0.80]])
```

torch.vstack((a,b))
torch.concatenate((a,b),dim=0)

```
tensor([[0.60, 0.61],
        [0.95, 1.00],
        [0.98, 0.25],
        [0.78, 0.80]])
```

torch.hstack((a,b))
torch.concatenate((a,b),dim=1)

```
tensor([[0.93, 0.79, 0.12, 0.30],
        [0.79, 0.55, 0.59, 0.98]])
```

Tensores en Pytorch: Manipulación

- *torch.unsqueeze(input,dim)* Devuelve un nuevo tensor al que se le ha insertado una dimensión de tamaño 1 en *dim*

```
>>> x = torch.tensor([1, 2, 3, 4])
>>> torch.unsqueeze(x, 0)
tensor([[ 1,  2,  3,  4]])
>>> torch.unsqueeze(x, 1)
tensor([[ 1],
        [ 2],
        [ 3],
        [ 4]])
```

citic

Tensores en Pytorch: Manipulación

- *torch.argwhere(input)*
  Devuelve un tensor que contiene los índices de todos los elementos no nulos

```
>>> t = torch.tensor([1, 0, 1])
>>> torch.argwhere(t)
tensor([[0],
        [2]])
>>> t = torch.tensor([[1, 0, 1], [0, 1, 1]])
>>> torch.argwhere(t)
tensor([[0, 0],
        [0, 2],
        [1, 1],
        [1, 2]])
```

Tensores en Pytorch: Manipulación

- *torch.chunk(input,chunks,dim=0)* Intenta dividir un tensor en *chunks* trozos a lo largo de su dimensión *dim*

```
>>> torch.arange(11).chunk(6)
(tensor([0, 1]),
 tensor([2, 3]),
 tensor([4, 5]),
 tensor([6, 7]),
 tensor([8, 9]),
 tensor([10]))
>>> torch.arange(12).chunk(6)
(tensor([0, 1]),
 tensor([2, 3]),
 tensor([4, 5]),
 tensor([6, 7]),
 tensor([8, 9]),
 tensor([10, 11]))
>>> torch.arange(13).chunk(6)
(tensor([0, 1, 2]),
 tensor([3, 4, 5]),
 tensor([6, 7, 8]),
 tensor([ 9, 10, 11]),
 tensor([12]))
```

Tensores en Pytorch: Manipulación

Separarlos con el comando split

```
tensor([[0.74, 0.64, 0.85, 0.91, 0.29, 0.20],
        [0.97, 0.46, 0.61, 0.23, 0.94, 0.15],
        [0.56, 0.51, 0.90, 0.65, 0.14, 0.43],
        [0.44, 0.03, 0.31, 0.11, 0.54, 0.04]])
```

```
torch.split(a,2,dim=1)
```

```
(tensor([[0.74, 0.64],
        [0.97, 0.46],
        [0.56, 0.51],
        [0.44, 0.03]]), tensor([[0.85, 0.91],
        [0.61, 0.23],
        [0.90, 0.65],
        [0.31, 0.11]]), tensor([[0.29, 0.20],
        [0.94, 0.15],
        [0.14, 0.43],
        [0.54, 0.04]]))
```

Tensores en Pytorch: Operación

- Podemos realizar operaciones algebraicas con los tensores
  - Producto de matrices torch.matmul: torch.matmul
  - Producto element-wise: torch.mul, u operador *
  - Reducción por suma: tensor.sum()
  - Operaciones in-place
    - El resultado se almacena en el tensor desde el que se llama
    - Se identifican porque su nombre termina con el sufijo _
    - Ejemplos: add_, mul_,copy_, etc…

Usos comunes en ML

- Muchas de las operaciones de manipulación de tensores juegan un rol importante en procesos tales como el preprocesado de los datos
  - Las operaciones de r*eshape* permiten cambiar la forma del tensor con los datos para adaptarla a la entrada requerida por la primera capa del modelo
  - Las operaciones de *slicing* permiten seleccionar una parte de los datos
  - Las operaciones de concatenación permiten combinar múltiples fuentes de datos
  - Las operaciones de concatenación tipo *stack* son útiles para componer los batches de datos
  - Las de transposición son útiles cuando necesitamos intercambiar un par de dimensiones de los datos
  - La operación *unsqueeze* se usa a menudo para añadir una dimensión más a los datos, habitualmente la dimensión batch
  - Las operaciones s*plit* y *chunk* son útiles para dividir los datos en batches más pequeños

Usos comunes en ML: unsqueeze

```python
import torchvision.transforms as transforms
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
image_tensor = transform(image)

# La imagen de entrada debe tener la forma (batch_size, num_channels, height, width)
# Genera una primera dimensión batch_size=1
image_tensor = image_tensor.unsqueeze(0)
```

Usos comunes en ML: pad_sequence

```python
from torch.nn.utils.rnn import pad_sequence

# `word_to_ix` es un diccionario que asocia palabras a ids únicos
# y `embeddings` es un tensor que contiene los word embeddings.

sentence = ["hello", "world"]
indices = [word_to_ix[word] for word in sentence]
word_tensors = [embeddings[i] for i in indices]

# Rellena las secuencias para que tengan todas la misma longitud
padded_sequence = pad_sequence(word_tensors)
```

Usos comunes en ML: split

```python
# Si `image_batch` es un tensor con la forma (100, 3, 64, 64)
image_batch = torch.rand(100, 3, 64, 64)

# Podemos dividirlo en batches más pequeños (por ejemplo de 10)
small_batches = image_batch.split(10)  # Ahora cada batch tiene la forma (10, 3, 64, 64)
```

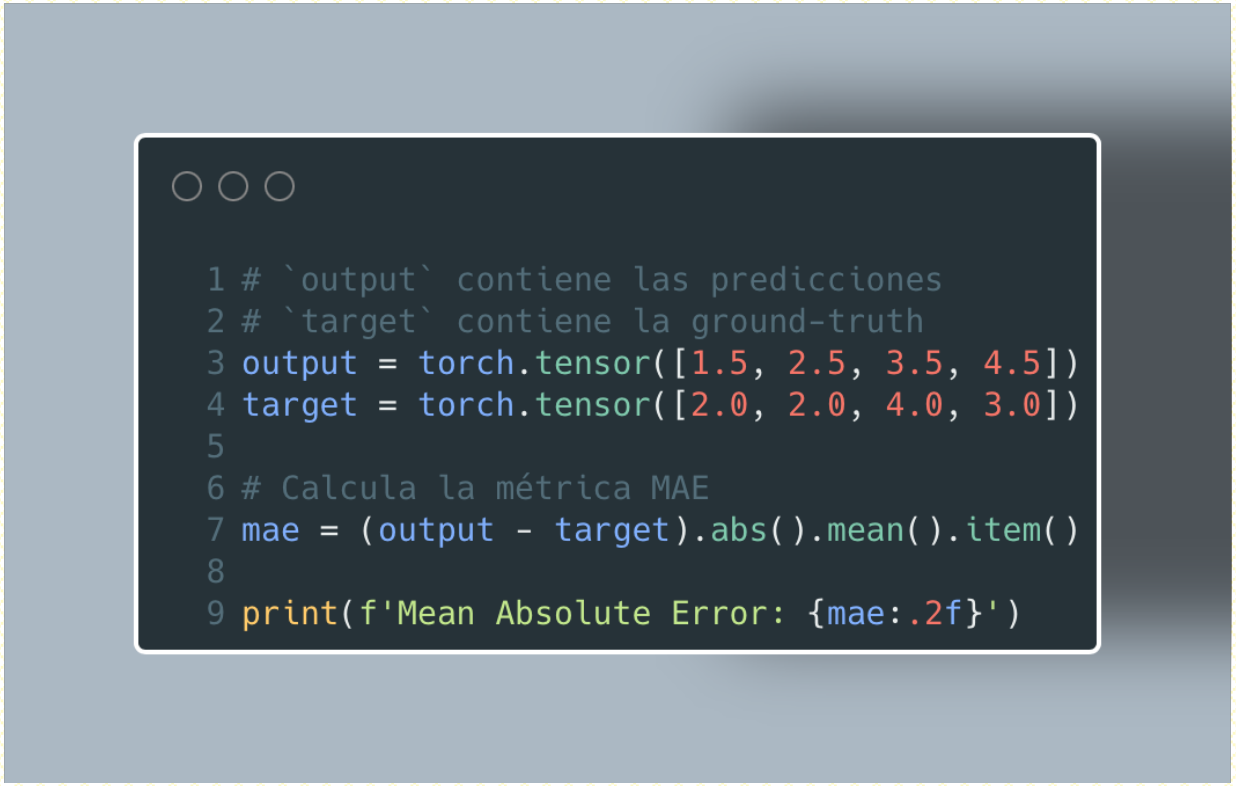
Usos comunes en ML

- Otro uso común de los tensores es al final del proceso de entrenamiento para calcular métricas de precisión por ejemplo
  - Funciones de reducción son útiles como parte del cálculo de estadísticas
  - Funciones que permiten calcular máximos (o mínimos) y su posición en un array, son útiles para post-procesar las salidas de un modelo de clasificación
  - Las operaciones aritméticas sobre tensores son útiles como parte del cálculo de métricas de precisión

Usos comunes en ML: max

```
# Asume que el `model` ya está entrenado
# `data` son los datos de prueba
# y `labels` son las etiquetas correctas

# Haz predicciones
outputs = model(data)
# Calcula la probabilidad máxima y su posición
_, predicted = torch.max(outputs, 1)

# Calcula la predicción
correct = (predicted == labels).sum().item()
total = labels.size(0)
accuracy = correct / total

print('Precisión del modelo sobre las imágenes de prueba: %d %%' % (100 * accuracy))

```

Uso en ML: operaciones aritméticas



```python
# `output` contiene las predicciones
# `target` contiene la ground-truth
output = torch.tensor([1.5, 2.5, 3.5, 4.5])
target = torch.tensor([2.0, 2.0, 4.0, 3.0])

# Calcula la métrica MAE
mae = (output - target).abs().mean().item()

print(f'Mean Absolute Error: {mae:.2f}')
```

Uso en ML: Definición de capas de RNP

```python
def conv2d(input, filter):
    # Extrae las dimensiones de la entrada
    h, w = input.shape[-2:] # Asume shape (batch, channels, height, width)
    f_h, f_w = filter.shape[-2:] # Asume filter shape (out_channels, in_channels, filter_height, filter_width)

    # Computa las dimensiones de la salida
    out_h = h - f_h + 1
    out_w = w - f_w + 1

    # Crea el tensor de salida
    output = torch.zeros((input.shape[0], filter.shape[0], out_h, out_w))

    # Operación de convolución
    for i in range(out_h):
        for j in range(out_w):
            # Selecciona la región de la entrada a la que se aplicará el filtro en esta iteración
            input_region = input[:, :, i:(i + f_h), j:(j + f_w)]

            # La función einsum hace el dot product entre el filtro
            # y la región correspondiente de la entrada
            output[:, :, i, j] = torch.einsum('bchw, ochw -> bo', input_region, filter)

    return output

```

Laboratorio

- Revisa el notebook tensores.ipynb del repositorio github del curso
- Realiza los **Ejercicios** al final del notebook

Variables y gradientes

Torch.autograd

- Torch.autograd: Sistema de diferenciación automática (*automatic differentiation system)*
  - Componente central de Pytorch
- Registra un grafo que representa todas las operaciones realizadas sobre un conjunto de tensores
  - Las hojas son los tensores de entrada
  - La raíz es la salida
- Recorriendo el grafo (raíz->hojas) podemos calcular automáticamente los gradientes usando la regla de la cadena

Torch.autograd

- Autograd mantiene en el grafo un conjunto de objetos de tipo función (*graph_fn*)
  - Se van creando durante la operación de los tensores
  - Llamando a la pasada *backward* vamos ejecutando la función *apply()* sobre todos ellos
- El grafo se recrea cada vez que se opera de nuevo con los tensores
  - En la práctica, esto sucede en cada pasada *forward*
  - Esta generación dinámica del grafo permite enriquecer la pasada forward con elementos dinámicamente configurables o con sentencias condicionales

Torch.autograd

- Existen nodos del grafo cuyas funciones no son diferenciables
  - Ejemplos: relu, o sqrt sobre 0
- Para reducir el impacto de estas funciones en el cálculo, se siguen una serie de reglas bien definidas
  - Si la función es diferenciable, se calcula el gradiente
  - Si la función es cóncava, usa el sub-gradiente de la norma mínima (es la dirección descendente más pronunciada)
  - Si la función es convexa, usa el super-gradiente de la norma mínima (se calcula lo mismo que en el punto anterior, pero para –f(x)
  - …

Torch.autograd

- Las Redes Neuronales (RNs) son una colección de funciones anidadas ejecutadas sobre algunos datos de entrada
  - Funciones parametrizadas a través de parámetros
    - Pesos (weights)
    - Y Bias
  - Función $\vec{y} = \vec{M}(\vec{x})$
    - $\vec{y}$ salida
    - $\vec{M}$ function
    - $\vec{x}$ entrada

Torch.autograd

- Función de pérdida (*loss*)
  - $L(\vec{y}) = L\left(\vec{M}(\vec{x})\right)$
    - L función de pérdida
      - La salida es un valor único
      - Indica la distancia entre la salida actual de nuestra función y la deseada
- El objetivo del proceso de entrenamiento es minimizar la función de pérdida
  - Ajuste progresivo de los parámetros de la función M (pesos+bias) hasta minimizar la función de pérdida
    - Implica hacer que su primera derivada respecto a la entrada sea 0
      - $\frac{\partial L}{\partial x} = 0$
- En una RN la función de pérdida no depende directamente de la entrada sino de una función de la salida del modelo (que a su vez es una función de la entrada)
  - $\frac{\partial L}{\partial x} = \frac{\partial(\vec{y})}{\partial x}$
  - Que por la regla de la cadena del cálculo diferencia es $\frac{\partial L}{\partial y}\frac{\partial M(x)}{\partial x}$

Torch.autograd

- El cálculo de la derivada parcial de una función $\frac{\partial M(x)}{\partial x}$ puede ser complejo
  - La función M encierra una gran complejidad matemática
    - Multiplicación de parámetros (pesos)
    - Aplicación de funciones de activación
    - Otras transformaciones matemáticas aplicadas por el modelo
  - La función no tiene un único camino posible
    - Puede ser un grafo con cierta complejidad
- El mecanismo de autograd proporciona una solución a la complejidad de este cálculo
  - Mantiene la historia de computaciones realizadas
  - Es capaz de calcular la derivada parcial de cada uno de los nodos de este grafo

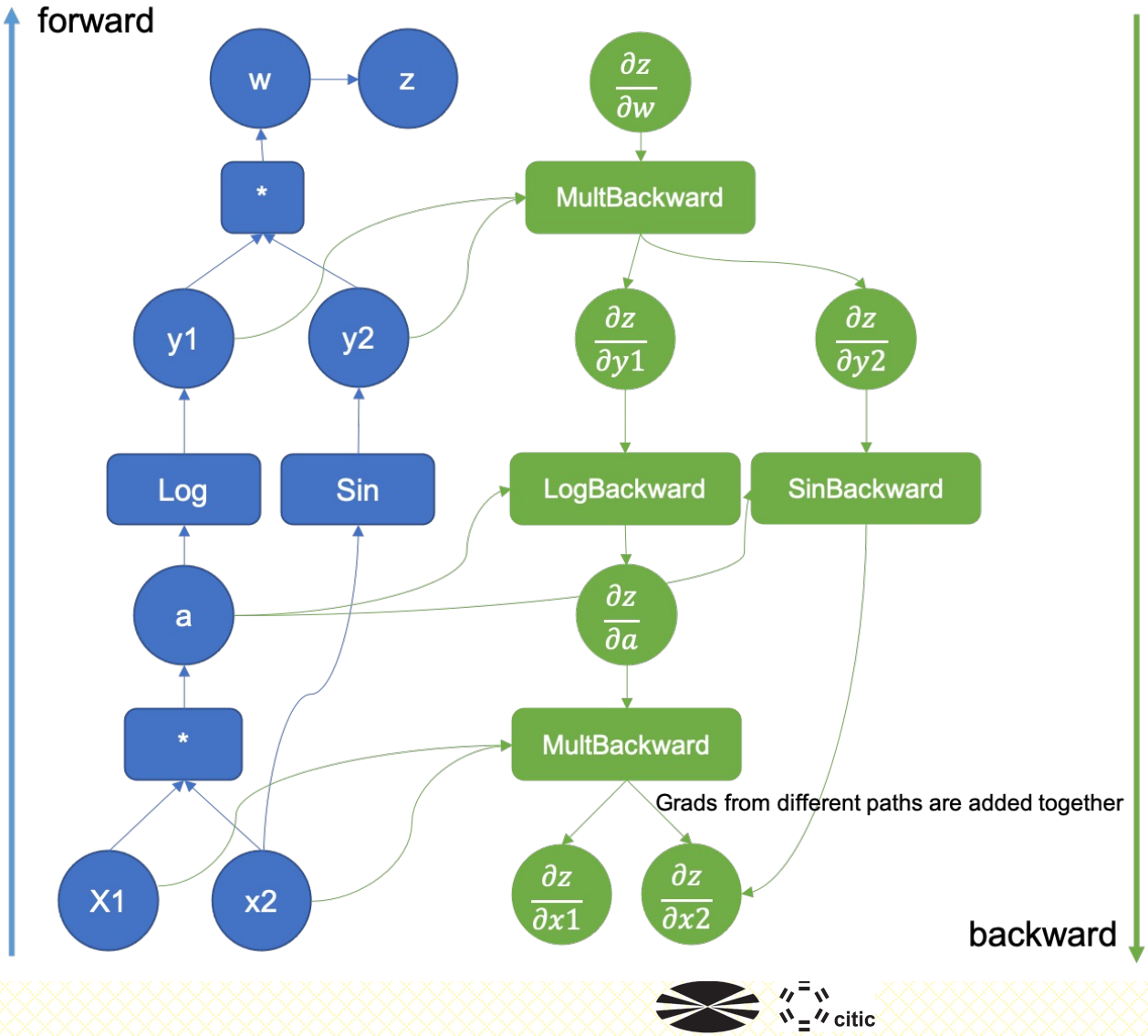
Torch.autograd

- El entrenamiento de una RN implica el ajuste de estos parámetros en dos fases:
  - Forward propagation: Se suministra una entrada a la red para comprobar qué salida se obtiene
  - Backward propagation: Se ajustan los parámetros de la red en una proporción que depende del error de su estimación
    - Ajuste hacia atrás
    - Es necesario computar las derivadas del error respecto a los parámetros de la función (gradients)
    - Con estos valores se ajustan los parámetros utilizando gradiente descents (GD)

Gradientes

- Cálculo de Mutiple Partial Derivatives sobre cálculos complejos
  - Crea dinámicamente y en tiempo de ejecución un grafo (DAG, Direct Acyclic Graph) que va registrando la computación (durante la forward pass)
  - Ese grafo se utiliza para calcular los gradientes en la back propagation
    - Útil cuando el cómputo para la que hay que calcular el gradiente es complejo
      - Difícil de calcular analíticamente
    - Indispensable si el cómputo tiene ramas que a veces se ejecutan y otras no, o bucles cuyo número de iteraciones no es fijo

Torch.autograd



citic

Torch.autograd

- Existen varios métodos para recopilar los gradientes asociados a tensores
  - Durante la creación del tensor, se usa el parámetro requires_grad=True
    - En la creación del tensor
  - Durante la operación del tensor
    - Usando tensor.requires_grad=True
    - Realizando operaciones dentro del ámbito with torch.no_grad()
- Puede ser problemático con las operaciones in place: ejemplo matmul_
  - Podrían sobrescribir los valores necesarios para calcular los gradientes
  - Solución sencilla: evitar su uso

Autograd en tiempo de entrenamiento

- El modelo entrenado debe estar construido a partir de tensores (o capas) con autograd activado
- Definimos:
  - Un optimizador: por ejemplo, SGD
  - Una función de pérdida: que mida la diferencia entre la salida deseada y la predicción actual del modelo para cada entrada
- Ejecutamos la función backward sobre el cálculo de la función de pérdida para calcular los gradientes
- Actualizamos los parámetros del modelo con los gradientes calculados usando la función step sobre el optimizador
  - Después de cada paso de optimización debemos reinicializar los gradientes a cero usando la función zero_grad()

Autograd en tiempo de inferencia

- Una vez el modelo ha sido entrenado, se debería desactivar el mecanismo de autograd
  - No es necesario, los pesos no se van a modificar
  - Si está activo, va a generar sobrecarga
    - Mayor uso de memoria
    - Mayor uso de recursos computacionales
  - Se aplica *.requires_grad(False)* sobre tensores individuales, o sobre un *Module* completo

Modos de autograd

- Modo de evaluación: nn.Module.eval()
  - Modo Grad: Es el modo por defecto, en el que el mecanismo de autograd funciona correctamente
  - Modo No-Grad: Deshabilita el mecanismo de autograd
    - No registra las operaciones realizadas sobre los tensores
    - Registra las salidas intermedias
      - Podrían ser útiles para un proceso de autograd posterior
  - Modo Inferencia: Es un modo extremo de No-Grad en el que se deshabilitan todos los mecanismos y resultados de autograd
    - Acelera todavía más la ejecución del modelo

Autograd multihilo

```python
# Define a train function to be used in different threads
def train_fn():
    x = torch.ones(5, 5, requires_grad=True)
    # forward
    y = (x + 3) * (x + 4) * 0.5
    # backward
    y.sum().backward()
    # potential optimizer update


# User write their own threading code to drive the train_fn
threads = []
for _ in range(10):
    p = threading.Thread(target=train_fn, args=())
    p.start()
    threads.append(p)

for p in threads:
    p.join()
```

Autograd multihilo

- Introduce concurrencia en la ejecución de la pasada *backward*
- Produce no-determismo
  - Varios hilos pueden acumular en el mismo atributo gradiente
    - Peligro *race-condition*
- Cada hilo mantiene su propio sub-grafo, pero podría haber partes del grafo compartidas
  - Efecto similar a ejecutar dos veces la operación *backward()* -> la segunda ejecución produce error en tiempo de ejecución
  - Evitable con *retain_graph=True*

El procesador de optimización de parámetros se puede monitorizar usando el profiler de autograd

```
>>> x = torch.randn((1, 1), requires_grad=True)
>>> with torch.autograd.profiler.profile() as prof:
>>>     for _ in range(100):  # any normal python code, really!
>>>         y = x ** 2
>>>         y.backward()
>>> # NOTE: some columns were removed for brevity
>>> print(prof.key_averages().table(sort_by="self_cpu_time_total"))
-----------------------------------  --------------  ---------------  ------------
--
Name                                 Self CPU total   CPU time avg     Number of
Calls
-----------------------------------  --------------  ---------------  ------------
--
mul                                  32.048ms         32.048ms         200
pow                                  27.041ms         27.041ms         200
PowBackward0                         9.727ms          55.483ms         100
torch::autograd::AccumulateGrad      9.148ms          9.148ms          100
torch::autograd::GraphRoot           691.816us        691.816us        100
-----------------------------------  --------------  ---------------  ------------
--
```

Profiling de autograd

Laboratorio

- Revisa el notebook autograd.ipynb del repositorio github del curso
- Realiza los **Ejercicios** al final del notebook

Referencias

- The fundamentals of Autograd: https://pytorch.org/docs/stable/autograd.html
- How Computational Graphs Are Constructed in Pytorch: https://pytorch.org/blog/computational-graphs-constructed-in-pytorch/
- Pytorch Internals: http://blog.ezyang.com/2019/05/pytorch-internals/
- https://oneapi-src.github.io/oneDNN/dev_guide_understanding_memory_formats.html

Referencias

- https://github.com/pytorch/tutorials/blob/main/intermediate_source/memory_format_tutorial.py
- https://www.shanelynn.ie/get-busy-with-word-embeddings-introduction/
- https://pytorch.org/docs/stable/notes/autograd.html