Creación y composición de arquitecturas de red

Diego Andrade Canosa

Roberto López Castro

Índice

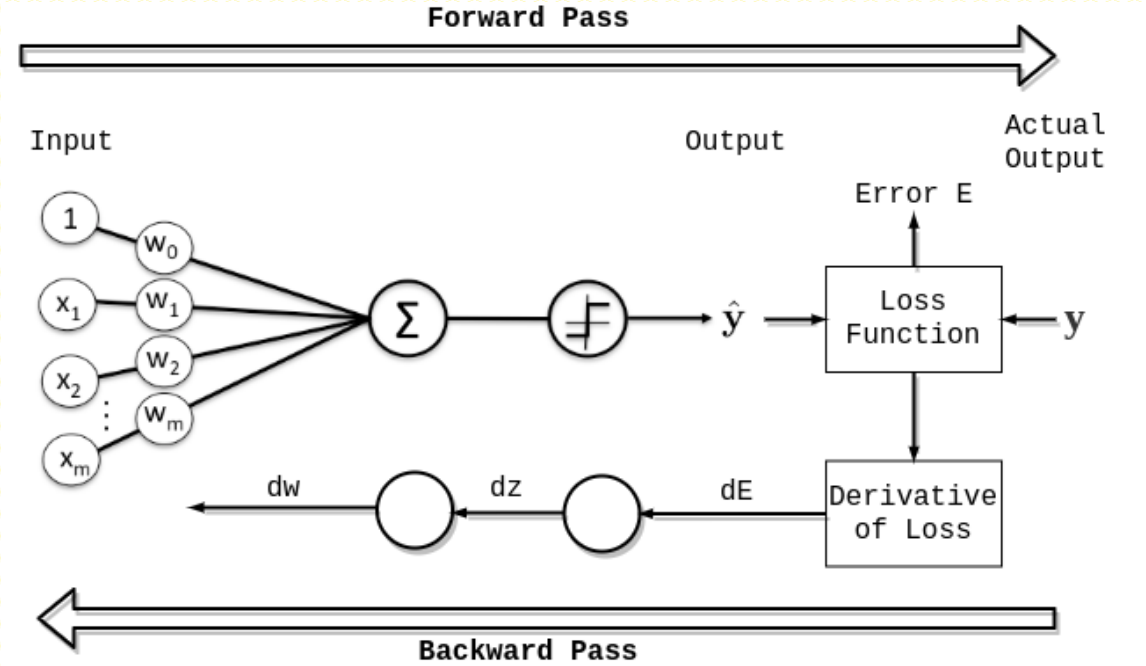
- Introducción
- Module y Sequential
- Bucles de entrenamiento
- Tipos de capas en Pytorch
- Optimizadores y funciones de pérdida
- Scores de modelos
- Tensorboard en Pytorch

Introducción

Objetivos

- Aprender a utilizar Pytorch para entrenar una red neuronal
- Hacer un recorrido por los bloques constructores que Pytorch nos proporciona:
  - Contenedores
  - Capas predefinidas
  - Optimizadores
  - Funciones de pérdida

Vista general de un proceso de entrenamiento



- Fuente: https://www.baeldung.com/cs/epoch-neural-networks

citic

Elementos básicos

- El entrenamiento del modelo requiere:
  - Definir el modelo a entrenar (tipos de capas, contenedores)
    - Definiendo su arquitectura
    - La pasada forward
  - Definir el bucle de entrenamiento
    - Cargar un dataset de entrenamiento, dividirlo en:
      - Conjunto de entrenamiento
      - Conjunto de validación
    - Paso (step) de entrenamiento
      - Optimizador
      - Función de pérdida
    - Comprobación de la precisión del modelo actual

Module y Sequential

Definición de la arquitectura del modelo

- Las RNs se representan en Pytorch utilizando módules (*Modules*)
  - Es el bloque constructor de *Stateful computation*
  - Integrada con el sistema de autograd de Pytorch
    - Facilita la especificación de cuáles son los parámetros aprendibles de la red que deben ser actualizados por el optimizador
  - Facilidad de uso y transformación:
    - Se pueden restaurar, guardar y transferir entre dispositivos
    - Purgar (*prune*)
    - Cuantizar (*quantize*)

Ejemplo simple de definición

```python
import torch
from torch import nn

class MyLinear(nn.Module):
  def __init__(self, in_features, out_features):
    super().__init__()
    self.weight = nn.Parameter(torch.randn(in_features, out_features))
    self.bias = nn.Parameter(torch.randn(out_features))

  def forward(self, input):
    return (input @ self.weight) + self.bias
```

Ejemplo simple de definición

```
m = MyLinear(4, 3)
sample_input = torch.randn(4)
m(sample_input)
```

✓ 0.0s

```
tensor([ 1.2322,  2.6996, -0.7115], grad_fn=<AddBackward0>)
```

Ejemplo simple de definición

- Es necesario proporcionar una implementación para el método forward
- Es posible, pero no necesario, proporcionarla para el método backward
  - Si no se hace, autograd es capaz de gestionar automáticamente esta pasada
- El proceso de entrenamiento se configura a través de sucesivas llamadas a ambos métodos

Parámetros del modelo

- En el “ejemplo simple”, los *Parameters* registrados en el modelo son los parámetros aprendibles del mismo

Modules como bloques constructores

```
net = nn.Sequential(
  MyLinear(4, 3),
  nn.ReLU(),
  MyLinear(3, 1)
)

sample_input = torch.randn(4)
net(sample_input)
```

- Los *modules* se pueden combinar de forma recursiva en Pytorch
- Una forma de hacerlo es a través del módulo *Sequential*
  - Recibe una lista de módulos
  - Se combinan entre si de forma que la salida de uno se envía a la entrada del siguiente

Modules como bloques constructores

- Los *modules* se pueden combinar de forma recursiva en Pytorch
- Salvo para modelos muy simples, es mejor utilizar módulos ad-hoc

```
import torch.nn.functional as F

class Net(nn.Module):
  def __init__(self):
    super().__init__()
    self.l0 = MyLinear(4, 3)
    self.l1 = MyLinear(3, 1)
  def forward(self, x):
    x = self.l0(x)
    x = F.relu(x)
    x = self.l1(x)
    return x
```

Modules como bloques constructores

- Se pueden recorrer los componentes de un módulo de varias formas:
  - Métodos *children* y *named_children* para recorrer los componentes de primer nivel (sin realizar un recorrido en profundidad en el grafo)
  - Métodos *modules* y *named_modules* para realizar un recorrido en profundidad del grafo de componentes del módulo
  - Métodos parameters y named_parameters permiten recorrer todos los parámetros aprendibles de un módulo y sus submódulos
- Con *ModuleList* y *ModuleDictionary* se pueden definir listas y diccionarios de módulos de forma dinámica

API de Container en Pytorch

Containers

`Module`	Base class for all neural network modules
`Sequential`	A sequential container.
`ModuleList`	Holds submodules in a list.
`ModuleDict`	Holds submodules in a dictionary.
`ParameterList`	Holds parameters in a list.
`ParameterDict`	Holds parameters in a dictionary.

Modules como bloque constructores

- Los Modules pueden reflejar una arquitectura que no es necesariamente secuencial

```python
class ParallelCNN(nn.Module):
    def __init__(self):
        super(ParallelCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)  # conv 1 sobre la entrada
        self.conv2 = nn.Conv2d(3, 6, 3)  # conv 2 sobre la entrada
        self.fc = nn.Linear(12 * 14 * 14, 10)  # capa lineal común

    def forward(self, x):
        x1 = F.relu(self.conv1(x))
        x2 = F.relu(self.conv2(x))
        # concatena la salida de las dos convolucionales
        x = torch.cat((x1, x2), dim=1)
        x = x.view(x.size(0), -1)
        # capa lineal sobre la concatenización de las convs
        x = self.fc(x)
        return x
```

Ejemplo de definición realista de un modelo complejo

- *Llama LLM* en Pytorch:

https://github.com/facebookresearch/llama/blob/main/llama/model.py

Modules como bloques constructores

- Podemos mover todos los parámetros de una red a un dispositivo *Model.to(device='cuda')*
- Podemos cambiar la precisión de todos los parámetros de una red *Model.to(dtype=torch.float64)*
- El método *apply* permite aplicar una función a un módulo y todos sus submódulos *Model.apply(custom_init_func())*

Modules como bloques constructores

- Un módulo por defecto está en modo entrenamiento
- Podemos alternar entre modo de entrenamiento y de evaluación usando los métodos *train()* y *eval()*
  - Su comportamiento es diferentes
    - En modo entrenamiento se mantiene y actualiza cierta información estadística, y de otro tipo, necesaria para este proceso
    - Sólo se debería poner un módulo en modo evaluación cuando se ponga en producción (para inferencia)

Herramientas para visualizar modelos: torchsummary

```
!pip install torchsummary
from torchsummary import summary
#Visualizamos la arquitectura de la red
model = Net().to(device)
summary(model, input_size=(1, 28, 28))
```

✓ 0.0s

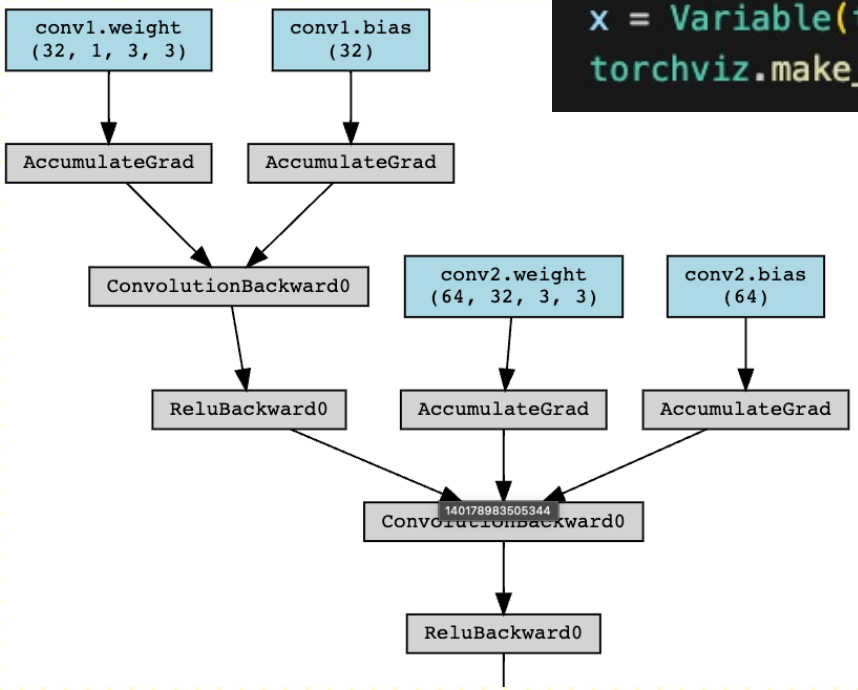
Orden de ejecución

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 26, 26]             320
            Conv2d-2           [-1, 64, 24, 24]          18,496
         Dropout2d-3           [-1, 64, 12, 12]               0
            Linear-4                 [-1, 128]       1,179,776
         Dropout2d-5                 [-1, 128]               0
            Linear-6                  [-1, 10]           1,290
================================================================
Total params: 1,199,882
Trainable params: 1,199,882
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.52
Params size (MB): 4.58
Estimated Total Size (MB): 5.10
----------------------------------------------------------------
```

Visualización de modelos: torchviz

```
!pip install torchviz
import torchviz
from torch.autograd import Variable

model = Net().to(device)
x = Variable(torch.randn(1, 1, 28, 28)).to(device)
torchviz.make_dot(model(x), params=dict(model.named_parameters()))
```



citic

Herramientas para visualizar modelos

- Tensorboard
  - Como es una herramienta que sirve para más cosas que para la visualización de la arquitectura del modelo, la veremos más adelante

Bucles de entrenamiento

Ejemplo de bucle de entrenamiento

```
# Create the network (from previous section) and optimizer
net = Net()
optimizer = torch.optim.SGD(net.parameters(), lr=1e-4, weight_decay=1e-2, momentum=0.9)

# Run a sample training loop that "teaches" the network
# to output the constant zero function
for _ in range(10000):
  input = torch.randn(4)
  output = net(input)
  loss = torch.abs(output)
  net.zero_grad()
  loss.backward()
  optimizer.step()

# After training, switch the module to eval mode to do inference, compute performance metrics, etc.
# (see discussion below for a description of training and evaluation modes)
...
net.eval()
...
```

Bucle de entrenamiento

- Creación de la red (instanciación del módulo)
- Creación del optimizador: asociándole los parámetros de la RN
- Bucle de entrenamiento
  - Entrada
  - Inferencia
  - Cálculo de la pérdida (*loss*)
  - Reinicializar (poner a 0) los gradientes de los parámetros de la RN
  - Se llama a la función *loss.backward* para recalcular los gradientes
  - Se llama a la función *optimizar.step* para actualizar los gradientes

Tipos de capas en Pytorch

Definición de capas en Pytorch

- Pytorch proporciona un completo mecanismo de tensores que nos permite definir las estructuras de datos necesarias para construir una RN
- Hoy en día existen ciertas capas estándar que se pueden utilizar como bloques constructores de RNs de diferentes tipos
- Pytorch proporciona dentro de su paquete *torch.nn* una serie de capas predefinidas que podemos utilizar como bloques constructores de nuestros modelos

Familias de capas

- Linear
- Convolutional
- Pooling
- Padding
- Normalization
- Dropout
- Recurrent
- Vision
- Transformers
- Embedding
- Activaciones
- Sparse
- Shuffle

Capas lineales

- Aplican transformaciones lineales entre unos parámetros de entrada y otros de salida
  - *nn.Linear:* Realiza una transformación lineal entre la entrada y la salida $y = x \cdot A^T + b$
  - *nn.Bilinear:* Realiza una transformación bilineal entre la entrada y la salida $y = x_1^T \cdot Ax_2 + b$
  - *nn.LazyLinear:* Similar a *Linear* pero las características de entradas se infieren
  - *nn.Identity:* Operador identidad

Capas Lazy en Pytorch

- Son métodos especiales de creación de capas a los que no es necesario proporcionarles la forma (*shape*) de la entrada que reciben
  - La infieren a través de la primera entrada que reciben en inferencia
  - (A veces) requieren de una ejecución en seco (*dry-run*) antes de resultar operativas

Capas convolucionales

- *nn.Conv1D*, *Conv2D* o *Conv3D*: Aplican una convolución sobre una señal (a menudo una imagen) de 1, 2 o 3 dimensiones (a menudo canales)

$$out\left(N_i, C_{out_j}\right) = bias\left(C_{out_j}\right) + \sum_{k=0}^{C_{in}-1} weight\left(C_{out_j}, k\right) \cdot input(N_i, k)$$

  - Versiones traspuestas: *ConvTranspose1d, 2d o 3d*
  - Versiones Lazy: *LazyTranspose1d, 2d*, o *3d*
    - Versiones *Lazy-Transposed*
- Operaciones *fold y unfold:* Permiten extraer los bloques de un tensor en lotes (*batched)*, y devolverlos al formato original, respectivamente

Capas de tipo Pooling

- La operación de Pooling consiste en reducir el tamaño de una capa combinando un número de elementos consecutivo configurable en uno, mediante:
  - El cálculo del máximo (MaxPooling)
  - La media (AveragePooling)
- Pytorch proporciona métodos para hacer los 2 tipos de Pooling (Max, Avg) en tensores de 1, 2 y 3 dimensiones
  - Y para hacer la operación inversa
  - Ejemplos: *MaxPool2d, AvgPool2d, MaxUnpool2d*

Capas de tipo Padding

- La operación de Padding (o rellenado) consiste en rellenar los bordes de un tensor con unos determinados valores.
- Hay diferentes tipos en función de la estrategia usada para rellenar:
  - Zero
  - Constant
  - Reflection
  - Replication
- Ejemplos: *ReplicationPad2d, ZeroPad2d, ReflectionPad2d*

Capas de Normalización

- La operación de normalización ajusta los valores de los parámetros de la red de acuerdo a una transformación estadística
  - Un ejemplo sencillo sería normalizar los parámetros usando la media de un valor y sus valores colindantes
  - Es un campo activo de investigación donde se han impuesto varias aproximaciones
  - Algunas de ellas están implementadas en Pytorch
  - Ejemplos: *BatchNorm2d*, *GroupNorm*, *InstanceNorm2d*

Capas de Dropout

- El mecanismo de Dropout consiste en poner algunos parámetros de la RN a cero
  - La elección de los parámetros anulados se hace forma aleatoria siguiendo una determinada distribución de probabilidad
- Ejemplos: *Dropout*, *Dropout2d*, *AlphaDropout*, *FeatureAlphaDropout*

Capas para redes recurrentes

- Las redes recurrentes son aquellas en las que la información de las entradas previas se añade a la información de la entrada actual
- Pytorch proporciona bloques constructores completos para este tipo de redes
  - Basados en aproximaciones populares
    - Long short-term Memory LSTM
    - Multi-layer Gated Recurrent Unit (GRU)
  - Proporcionan formas de generar la red completa
    - O células (*Cells*) individuales
  - Ejemplos: *RNN, LSTM, GRU, LSTMCell, GRUCell*

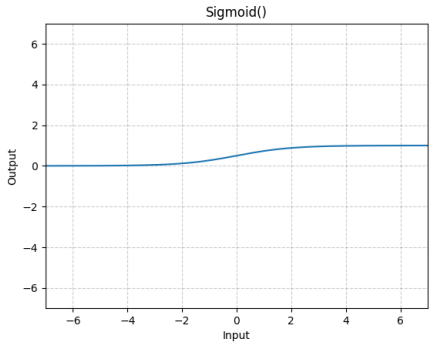
Capas para redes Transformer

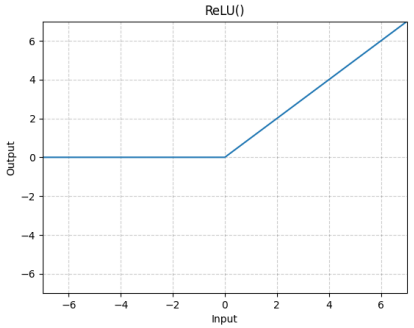
- Los modelos Transformer son populares en aplicaciones de NLP (Procesamiento del Lenguaje Natural)
- Los bloques constructores básicos:
  - Codificadores (Encoders): Se encargan de la lectura del texto
  - y decodificadores (Decoders): Se encargan de la generación de las predicciones de texto
- Pytorch proporciona capas para construir modelos completos, porciones de modelos o capas individuales
  - Ejemplos: *Transformer*, *TransformerEncoder*, *TransformerDecoder*, *TransformerEncoderLayer*, *TransformerDecoderLayer*

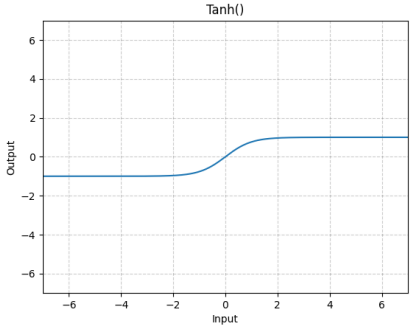
Capas Sparse

- Son capas que transforman un vector de valores de una variable discreta que representa datos categóricos -> un vector de valores de una variable continua
- Se utilizan para representar textos, cada porción de texto se representa mediante uno de esos vectores
- Ejemplo: *torch.nn.Embedding*

Capas de activación no lineal







- Son capas que transforman un valor numérico continuo en un valor en el intervalo (0,1)
- Son un bloque constructor esencial para la construcción de RNs
- Existen multitud de ellas
  - Algunas son más adecuadas que otras para cierto tipo de aplicaciones
  - Funciones más populares
    - *torch.nn.Softmax*
    - *torch.nn.RELU*
    - *torch.nn.Tanh*
    - *torch.nn.Sigmoid*

Laboratorio

- Revisa el notebook training.ipynb
- Realiza el **Ejercicio** al final del notebook

Optimizadores y funciones de pérdida

Optimizadores y funciones de pérdida

- Recordemos que:
  - Un modelo está compuesto de muchos parámetros aprendibles (*learnable parameters*)
  - Estos parámetros se actualizan en cada paso (*step*) del bucle de entrenamiento
  - El proceso empieza comparando la salida obtenida por el modelo (en inferencia) con la salida deseada
    - La diferencia entre ambas se calcula utilizando una función de pérdida (*loss function)*
  - El optimizador utiliza la salida de esta función de pérdida para calcular los gradientes (modificaciones) de los parámetros del modelo
  - Estos gradientes se usan para optimizar los parámetros del modelo

Funciones de pérdida

- Calculan la diferencia entre la salida obtenida por el modelo y la salida deseada
- Hay muchas formas de calcularla
- Pytorch implementa alrededor de 20 funciones diferentes que se pueden usar para hacer este cálculo
- Existen dos tipos principales de funciones de pérdida:
  - Regresión: Son útiles para tareas en las que el objetivo es predecir un valor numérico en un intervalo continuo
    - Ejemplo: Predecir la probabilidad de lluvia
  - Clasificación: Son útiles para tareas en las que el objetivo es predecir un valor entre varios posibles valores categóricos
    - Predecir la especie de un animal que aparece en una fotografía

Funciones de pérdida regresivas

- Mean Square Error (MSE)/Quadratic/L2 Loss

$$\text{MSE} = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}$$

Captura la media de la diferencia entre la salida teórica y la salida obtenida sin tener en cuenta si la diferencia es positiva o negativa

Ventaja: Debido a ciertas propiedades del cuadrado, el cálculo de los gradientes (derivada) se simplifica

Problema: Amplifica errores grandes debido al uso del cuadrado

Función de Pytorch torch.nn.MSELoss

Funciones de pérdida regresivas

- Mean Absolute Error (MAE)/L1 Loss

$$MAE = \frac{\sum_i^n |y_i - \widehat{y_i}|}{n}$$

Usa el valor absoluto para no tener en cuenta si la diferencia es positiva o negativa

Ventaja: No amplifica las diferencias grandes

Función de Pytorch: torch.nn.L1Loss

Funciones de pérdida regresivas

- Mean Bias Error (MBE)

$$MBE = \frac{\sum_i^n (y_i - \hat{y}_i)}{n}$$

No usa el valor absoluto ni el cuadrado, por lo que tiene en cuenta el signo del error.

Ventaja: Fácil de implementar. Se puede utilizar para capturar el signo del error.

Problema: Las diferencias podrían cancelarse y eso lo hace menos preciso

Función de Pytorch: Ninguna, fácil de calcular de forma directa.

```
(y-yhat).mean()
```

Funciones de pérdida categóricas

- Hinge/Multiclass SVM Loss

$$SVM = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

En redes en las que se clasifica algo (ej. una imagen) en una de varias categorías. La salida de la red suele ser un vector con tantos elementos como categorías, y donde cada elemento contiene la probabilidad (o score) asociado a esa categoría

Una predicción se considera correcta si el score de la categoría correcta es mayor que la suma de los scores de las demás categorías por un margen seguro (normalmente al menos 1)

Funciones de pérdida categóricas

- Hinge/Multiclass SVM Loss

$$SVM = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

La fórmula de esta función de pérdida captura esta condición

Dará valores más altos cuanto más equivocada sea la predicción

Dará cero si la predicción es correcta

Función de Pytorch: torch.nn.MultiLabelMarginLoss

Funciones de pérdida categóricas

- Hinge/Multiclass SVM Loss



	Image #1	Image #2	Image #3
Dog	-0.39	-4.61	1.03
Cat	1.49	3.28	-2.37
Horse	4.21	1.46	-2.27

```
## 1st training example
max(0, (1.49) - (-0.39) + 1) + max(0, (4.21) - (-0.39) + 1)
max(0, 2.88) + max(0, 5.6)
2.88 + 5.6
8.48 (High loss as very wrong prediction)

## 2nd training example
max(0, (-4.61) - (3.28)+ 1) + max(0, (1.46) - (3.28)+ 1)
max(0, -6.89) + max(0, -0.82)
0 + 0
0 (Zero loss as correct prediction)

## 3rd training example
max(0, (1.03) - (-2.27)+ 1) + max(0, (-2.37) - (-2.27)+ 1)
max(0, 4.3) + max(0, 0.9)
4.3 + 0.9
5.2 (High loss as very wrong prediction)
```

Funciones de pérdida categóricas

- Cross Entropy/Negative Likelihood Loss

$$LCE = -\sum_i y_i \cdot \log_2(\hat{y}_i)$$

Es la función categórica más popular

Multiplica el logaritmo de la probabilidad predicha por la clase verdadera

Su valor aumenta cuando la probabilidad de la categoría correcta diverge de 1

Si es 1, desaparece el segundo término

Si es 0, desaparece el primer término

Funciones de pérdida categóricas

- Cross Entropy/Negative Likelihood Loss

$$LCE = -\sum_i y_i \cdot \log_2(\hat{y}_i)$$

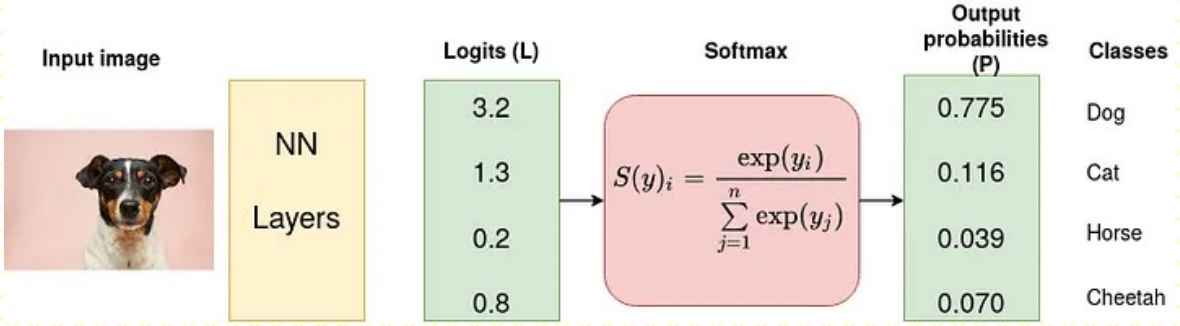
Penaliza predicciones incorrectas que el modelo proporciona con seguridad

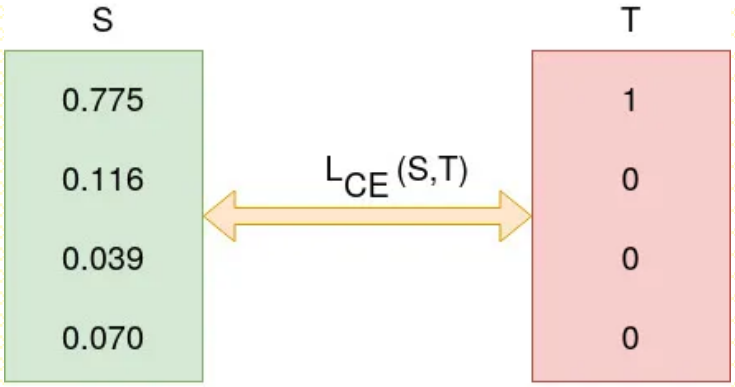
El logaritmo amplifica el hecho de que se le asocie una probabilidad muy baja a la clase correcta

Función en Pytorch: torch.nn.CrossEntropyLoss

o LogSoftMax + NLLLoss

Cross Entropy Loss





$$
\begin{aligned}
L_{CE} &= -\sum_{i=1} T_i \log(S_i) \\
&= -\left[1 \log_2(0.775) + 0 \log_2(0.126) + 0 \log_2(0.039) + 0 \log_2(0.070)\right] \\
&= -\log_2(0.775) \\
&= 0.3677
\end{aligned}
$$

citic

Laboratorio

- Revisa el notebook losses.ipynb
- Realiza el **Ejercicio** al final del notebook

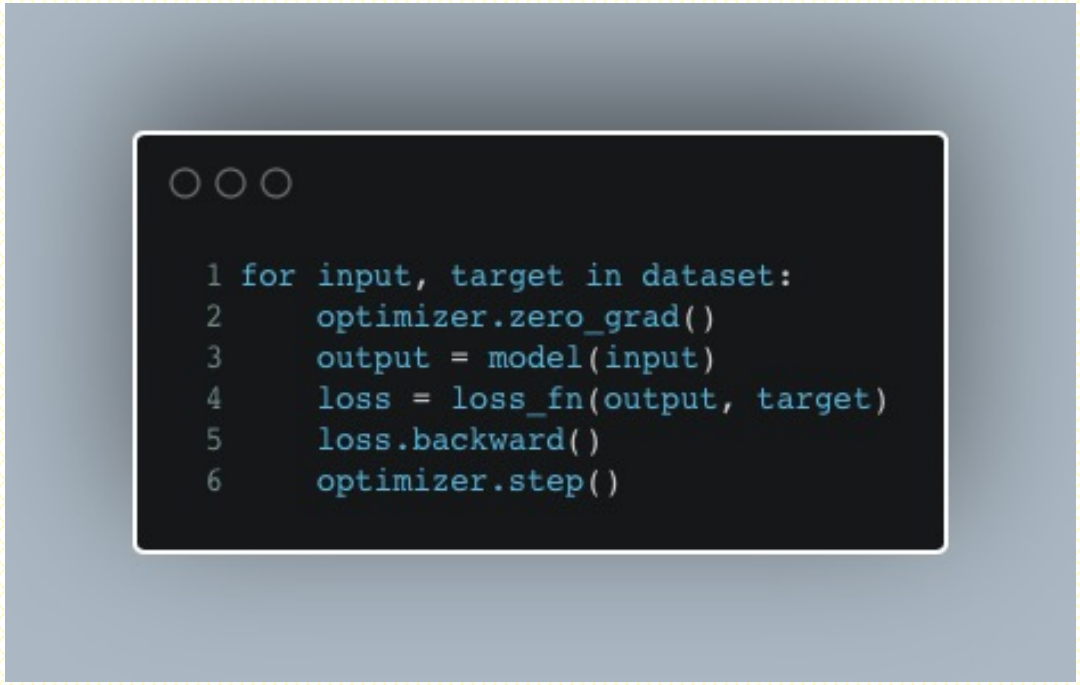
Optimizadores

- Pytorch implementa varios algoritmos de optimización en su paquete torch.optim
- Se basa en la construcción de un objeto optimizador que mantendrá el estado actual y actualizará los parámetros usando un cálculo de sus gradientes
- Interfaz común que facilita
  - El cambio sencillo de algoritmo de optimización
  - La implementación de optimizadores ad-hoc
- El optimizador recibe un iterable con los parámetros del modelo
  - Recibe otros parámetros, como por ejemplo el learning rate o el momentum

Optimizadores

- El optimizador hace el cálculo de los gradientes al invocar el método **backward()** sobre el cálculo de la función de pérdida
- Los gradientes calculados por el optimizador se aplican a los parámetros del modelo usando el método **step()** del optimizador
- Si queremos realizar los dos pasos de nuevo (en un bucle de optimización) los gradientes se deben poner a *cero* usando la función **zero_grad()** del optimizador

Optimizadores



citic

Optimizadores

- La clase base en Pytorch es optim.Optimizer y recibe dos parámetros en su constructor
  - params (iterable): Parámetros (tensores) a optimizar
  - defaults (dict): Contiene parámetros de configuración del optimizador, ej. lr (learning rate)

Algoritmos de optimización

- Principales tipos de algoritmos de optimización implementados en Pytorch
  - SGD (Stochastic Gradient Descent)
  - RMSProp
  - Adagrad (Adaptive Gradient)
  - Adam (Adaptive Moment Estimation)
  - Adadelta (Adaptive Delta)

Algoritmos de optimización

- Varios algoritmos tienen implementaciones optimizadas para conseguir rendimiento, legibilidad o genericidad
- Hay 2 categorías de implementación
  - for-loop: es la más directa pero la más lenta. Hace un recorrido (for-loop) de las porciones (*chunks*) de computación
  - foreach: combinan varios tensores en un multi-tensor y ejecutan todas las computaciones a la vez. Ahorran llamadas a kernels
  - Fused (solo en algunos optimizadores): combinan varias porciones de computación en un solo kernel
- Rendimiento: fused>foreach>for-loop

API del optimizador

`Optimizer.add_param_group`	Add a param group to the `Optimizer` s *param_groups*.
`Optimizer.load_state_dict`	Loads the optimizer state.
`Optimizer.state_dict`	Returns the state of the optimizer as a `dict`.
`Optimizer.step`	Performs a single optimization step (parameter update).
`Optimizer.zero_grad`	Sets the gradients of all optimized `torch.Tensor` s to zero.

Algorithm	Default	Has foreach?	Has fused?
`Adadelta`	foreach	yes	no
`Adagrad`	foreach	yes	no
`Adam`	foreach	yes	yes
`AdamW`	foreach	yes	yes
`SparseAdam`	for-loop	no	no

Algoritmos SGD

- Es uno de los tipos más básicos y más ampliamente utilizados
- Actualiza los parámetros del modelo en la dirección opuesta al gradiente de la función de pérdida
  - Este ajuste es proporcional al Learning Rate (lr) o tasa de aprendizaje
- Funciones en Pytorch: optim.SGD, optim.ASGD (averaged)

Algoritmos SGD

- El optimizador SGD implementa el algoritmo Momentum/Nesterov

$$v_{t+1} = \mu \times v_t + g_{t+1}$$
$$p_{t+1} = p_t - lr \times v_{t+1}$$

Donde p,g,v y μ son los parámetros, gradientes, velocidad y momentum respectivamente

Algoritmo Adagrad

- Adaptive Gradient Algorithm
- Características
  - Usa un lr distinto para cada parámetro
  - Modifica el lr en cada step basándose en los valores históricos de cada parámetro
- Ventajas
  - Buen rendimiento con datos dispersos
    - Asocia lr pequeños a parámetros frecuentes
    - Asociar lr grandes a parámetros infrecuentes
- Debilidades
  - Puede dejar de entrenar muy pronto debido a la acumulación de gradientes cuadrados lo que puede probar que el lr converja a cero
- Función de Pytorch: Adagrad

Algoritmo RMSProp

- Root Mean Square Propagation
- Es un algoritmo de optimización que incluye un mecanismo de adaptación del lr (evolución de Adagrad)
- Trata de resolver el problema del gradiente monotónicamente decreciente de Adagrad:
  - Uso de una media variable del cuadrado de los gradientes en vez de una suma acumulada (AdaGrad)
  - Introduce un factor de decaimiento
  - Solo se consideran una porción de los gradientes pasados, dándole más peso a los últimos
- Función en Pytorch: optim.RMSProp

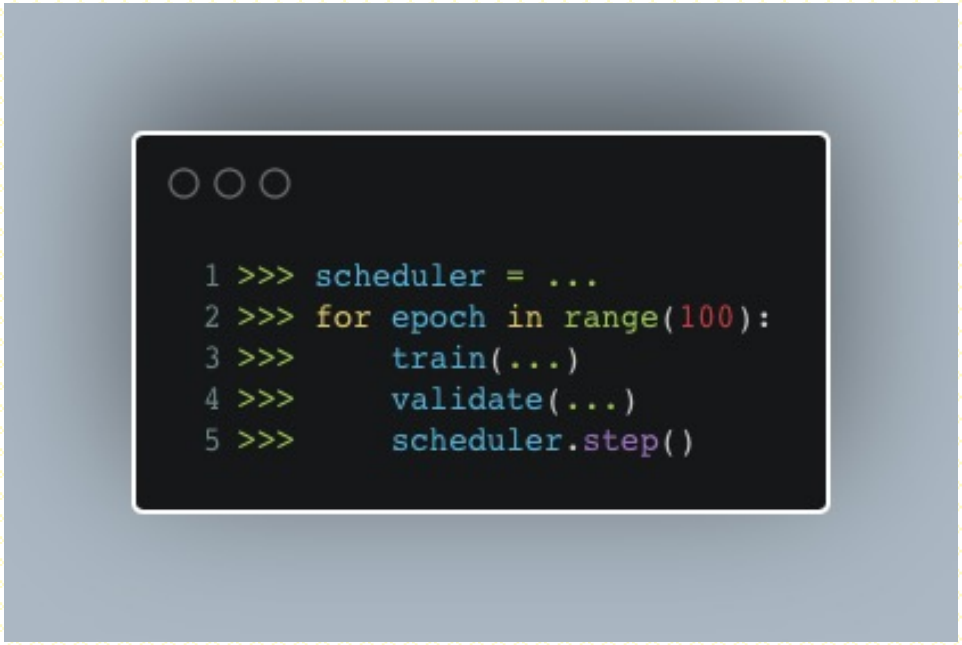
Algoritmos Adam

- Adam, Adaptative Moment Estimation combina las ventajas de
  - AdaGrad: el manejo de gradientes dispersos para problemas que son problemáticos para otros algoritmos
  - RMSProp: el manejo de objetivos no estacionarios (cambian con el tiempo)
- Para ello
  - Usa medias de los gradientes
    - Introduce el concepto de momentum sumando una fracción de los gradientes previos
  - Realiza una corrección del bias
- Funciones de Pytorch: Adam, AdamW, SparseAdam, Adamax, RAdam, NAdam

Ajuste del Learning Rate

- La tasa de aprendizaje o Learning Rate es uno de los parámetros de configuración del optimizador más relevantes
- Si es demasiado pequeña, el modelo tardará en ser entrenado
  - Pequeños pasos hacia una solución
- Si es demasiado grande, el modelo podría no entrenarse bien, o tardar en entrenarse
  - Pasos muy grandes que podrían pasar por alto una solución
- Pytorch proporciona varios planificadores (*schedulers*) para ajustar el lr
  - Estáticamente, antes de iniciar un epoch de entrenamiento y basándose en el número de epoch
  - Dinámicamente, durante el entrenamiento, y basándose en varias métricas de validación
- Se pueden encandenar entre si

Ajuste del Learning Rate



citic

Ajuste del Learning Rate

optim.lr_scheduler.StepLR

- Aplica un factor de reducción al lr en intervalos fijos
- Se puede configurar
  - step_size: Cada cuántas epochs ajustamos el lr
  - gamma: Es el factor multiplicador del learning rate (<1)

Ajuste del Learning Rate

- Optim.lr_scheduler.MultiStepLR
  - Reduce el lr en intervalos fijos permitiendo la especificación de múltiples puntos de cambio
  - Se puede especificar
    - milestones: lista de epochs en las que hacer el cambio
    - gamma: factor multiplicador del lr

Ajuste del Learning Rate

- Optim.lr_scheduler.ExponentialLR
  - Reducción exponencial del lr en cada epoch
  - Se puede configurar
    - gamma: factor multiplicador del lr

Ajuste del Learning Rate

- optim.lr_scheduler.CosineAnnealingLR: Suaviza la reducción del lr siguiendo una función de coseno en ciclos de annealing
- optim.lr_scheduler.ReduceLROnPlateau: Ajusta el lr cuando la métrica de validación deja de mejorar
- ...

Promediado de pesos

- Stochastic Weight Averaing (SWA): es una técnica para mejorar la capacidad de generalización de un modelo
- Para ello, realiza un promediado de los pesos obtenidos a través de un número determinado de iteraciones
  - Para el promedio, se cogen varios puntos de la trayectoria de optimización
  - Se basa en que el proceso de optimización encuentra varias soluciones buenas en la vecindad de la solución óptima
  - Promediar los pesos de esas soluciones buenas mejora la capacidad de generalización del modelo

Promediado de pesos

- Se puede crear una versión promediada de un modelo

  swa_model = AveragedModel(model)

- Si se quieren actualizar los parámetros podemos usar el comando

  swa_model.update_parameters(model)

- Se puede utilizar un lr fijo, o establecer una estrategia de ajuste de lr

  swa_scheduler = torch.optim.swa_utils.SWALR(optimizer,
  anneal_strategy="linear", anneal_epochs=5, swa_lr=0.05)

Promediado de pesos: ejemplo completo

```
loader, optimizer, model, loss_fn = ...
swa_model = torch.optim.swa_utils.AveragedModel(model)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=300)
swa_start = 160
swa_scheduler = SWALR(optimizer, swa_lr=0.05)

for epoch in range(300):
      for input, target in loader:
          optimizer.zero_grad()
          loss_fn(model(input), target).backward()
          optimizer.step()
      if epoch > swa_start:
          swa_model.update_parameters(model)
          swa_scheduler.step()
      else:
          scheduler.step()

# Update bn statistics for the swa_model at the end
torch.optim.swa_utils.update_bn(loader, swa_model)
# Use swa_model to make predictions on test data
preds = swa_model(test_input)
```

Laboratorio

- Revisa el notebook optimizers.ipynb
- Realiza el **Ejercicio** al final del notebook

Técnicas de Score de modelos en Pytorch

- Las técnicas de Score (o puntuación) de modelos evalúan el rendimiento y efectividad de un modelo de ML
- Miden la capacidad de generalización de nuestro modelo
  - Sobre datos para los que no ha sido entrenado
- Permiten comparar objetivamente nuestro modelo con otros diseñados para realizar la misma tarea
- Generan confianza en el modelo por parte de sus usuarios potenciales
- Se basan habitualmente en medir la distancia entre las predicciones del modelo y la salida deseada (*ground-truth*)

Score de regresión

Técnicas de precisión de Modelos

- La métricas MSE (*Mean Square Error*) que se usó para guiar al optimizador también se puede utilizar para calcular la precisión final de un modelo de regresión.

Score de clasificación

Técnicas de Score de modelos en Pytorch

- Precisión, *Recall (*exhausitividad*)*, *F1-Score*
  - Útiles en problemas de clasificación binaria
    - Extensibles a problemas de clasificación multiclase tratando cada clase por separado y promediando la media
  - Basados en el concepto de *confusión matrix*, dividen las instancias en:
    - *True Positive (TP)*: El modelo predice correctamente la clase positiva
    - *False Positive (FP)*: El modelo predice incorrectamente la clase positiva
    - y *False Negative (FN)*: El modelo predice incorrectamente la clase negativa

		predicción	
		0	**1**
realidad	**0**	70	10
	1	15	5

		predicción	
		0	**1**
realidad	**0**	TN	FP
	1	FN	TP

Confusion matrix

Técnicas de Score de modelos en Pytorch

- Precisión (*accuracy*)
  - Mide la calidad de las respuestas

$$precision = \frac{TP}{TP + FP}$$

		predicción	
		0	1
realidad	0	TN	FP
	1	FN	TP

Técnicas de Score de modelos en Pytorch

- *Recall* (exhaustividad)
  - Mide la cantidad que el modelo es capaz de identificar

$$recall = \frac{TP}{TP + FN}$$

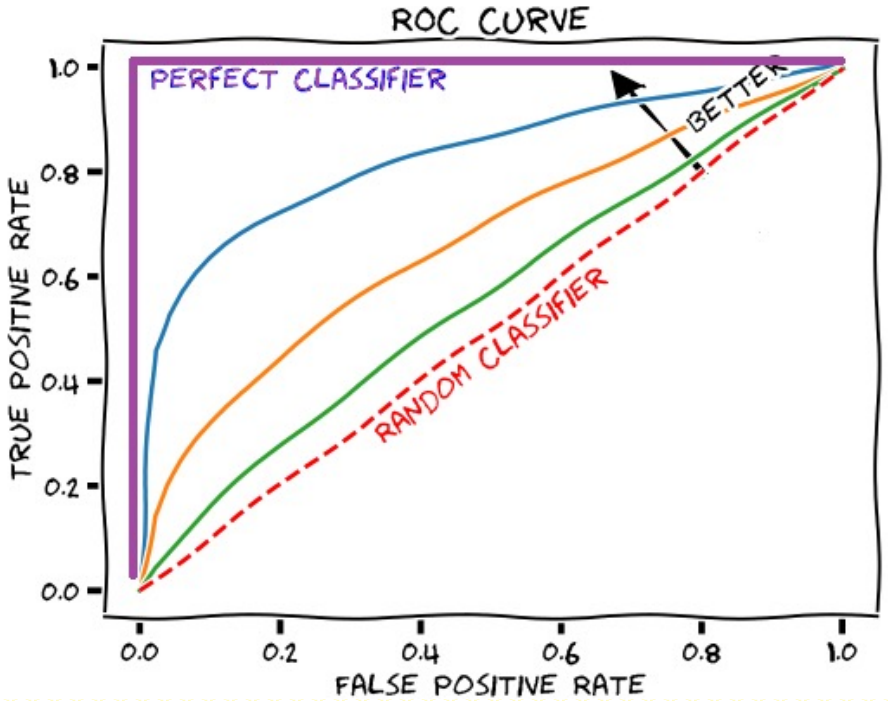
		predicción	
		0	1
realidad	0	TN	FP
	1	FN	TP

Técnicas de Score de modelos en Pytorch

- *F1-score*
  - Combina las dos métricas anteriores

$$\mathrm{f1} - score = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

Técnicas de Score de modelos en Pytorch



Curva ROC: Es el balance entre el *true positive rate* (TPR) y el *false positive rate* (FPR) para distintos umbrales de clasificación

- TPR se representa en el eje-y

$$TPR = \frac{TP}{TP+FN}$$

- FPR se representa en el eje-x

$$FPR = \frac{FP}{FP+TN}$$

citic

Técnicas de Score de modelos en Pytorch

- AUC: Area debajo de la curva ROC
  - Mide al área debajo de la curva ROC integrando la curva ROC como una función en el intervalo 1
  - Una forma de interpretar el AUC es como la probabilidad de que el modelo clasifique como positivo un ejemplo positivo aleatorio más alto que un ejemplo negativo aleatorio
  - El AUC varía en valor de 0 a 1:
    - AUC de 0.0 indica predicciones 100% incorrectas
    - AUC de 1.0 indica predicciones 100% correctas

Técnicas de Score de modelos en Pytorch

- El AUC es una forma de agregación de valores de métricas, se puede aplicar a otras métricas que no sea la curva ROC
- Otras métricas de agregación:
  - Suma
  - Media
  - ...

Técnicas de Score de modelos en Pytorch

- Precisión (*accuracy*) del modelo
  - Se usa para tareas de clasificación
  - Mide la proporción de predicciones que son correctas

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

  - Puede ser una métrica problemática en conjuntos de datos desbalanceados, donde una clase es más habitual que otras

Score de modelos de lenguaje

Técnicas de Score de modelos en Pytorch

- Perplexity: Es una métrica de uso habitual en modelos de lenguaje
- Mide cómo de bien predice el modelo datos de ejemplo, mediante la siguiente fórmula:

$$ppl = \exp(sum\ of\ negative \log \frac{likelihood}{number}\ of\ tokens$$

$$\mathrm{PPL}(X) = \exp\left\{-\frac{1}{t}\sum_{i}^{t}\log p_\theta(x_i|x_{<i})\right\}$$

```
>>> metric=Perplexity()
>>> input = torch.tensor([[[0.3659, 0.7025, 0.3104]], [[0.0097, 0.6577, 0.1947]],
[[0.5659, 0.0025, 0.0104]], [[0.9097, 0.0577, 0.7947]]])
>>> target = torch.tensor([[2],  [1], [2],  [1]])
>>> metric.update(input, target)
>>> metric.compute()
tensor(3.5257, dtype=torch.float64)
```

citic

Técnicas de Score de modelos en Pytorch: Perplexity

```
>>> metric=Perplexity(ignore_index=1)
>>> input = torch.tensor([[[0.3659, 0.7025, 0.3104]], [[0.0097, 0.6577, 0.1947]],
[[0.5659, 0.0025, 0.0104]], [[0.9097, 0.0577, 0.7947]]])
>>> target = torch.tensor([[2],  [1], [2],  [1]])
>>> metric.update(input, target)
>>> metric.compute()
tensor(3.6347, dtype=torch.float64)
```

```
>>> metric1=Perplexity()
>>> input = torch.tensor([[[0.5659, 0.0025, 0.0104]], [[0.9097, 0.0577, 0.7947]]])
>>> target = torch.tensor([[2],  [1], ])
>>> metric1.update(input, target)
>>> metric1.compute()
tensor(4.5051, dtype=torch.float64)
```

```
>>> metric2=Perplexity()
>>> input = torch.tensor([[[0.3659, 0.7025, 0.3104]], [[0.0097, 0.6577, 0.1947]]])
>>> target = torch.tensor([[2],  [1]])
>>> metric2.update(input, target)
>>> metric2.compute())
tensor(2.7593, dtype=torch.float64)
```

citic

Técnicas de Score de modelos en Pytorch

- Bleu (Bilingual Evaluation Understudy) Score: Es un score útil para evaluar modelos de traducción
  - Se basa en la comparación entre las predicciones candidatas del modelo y las predicciones candidatas tomadas como referencia
    - Traducciones de referencias: varias traducciones posibles generadas por un humano
    - Traducciones máquina: varias traducciones candidatas generadas por el modelo evaluado
  - Las traducciones se comparan contando cuántas palabras coinciden
    - Las palabras deben coincidir en la misma parte de la frase
    - Las cadenas de texto deben tener longitudes similares
    - A veces se comparan grupos de palabras que suelen ir juntas (sujeto+verbo , preposición + nombre)
  - Toma valores entre 0 y 1: valores más altos indicando mayor precisión

Técnicas de Score de modelos en Pytorch

- Bleu (Bilingual Evaluation Understudy) Score: Es un score útil para evaluar modelos de traducción

```
>>> import torch
>>> from torcheval.metrics.functional.text import bleu
>>> candidates = ["the squirrel is eating the nut"]
>>> references = [["a squirrel is eating a nut", "the squirrel is
eating a tasty nut"]]
>>> bleu_score(candidates, references, n_gram=4)
tensor(0.53728497)
>>> candidates = ["the squirrel is eating the nut", "the cat is on the
mat"]
>>> references = [["a squirrel is eating a nut", "the squirrel is
eating a tasty nut"], ["there is a cat on the mat", "a cat is on the
mat"]]
>>> bleu_score(candidates, references, n_gram=4)
tensor(0.65341892)
```

Técnicas de Score the modelos en Pytorch

- Word-Error-Rate (WER): Es una técnica de Score de modelos apta para aplicaciones de reconocimiento de habla (*speech recognition*)
  - Compara la salida esperada (secuencia de palabras) con la salida del modelo (secuencia obtenida)
  - Cuenta la suma de tres tipos de errores:
    - Sustituciones: Ocurre cuando el modelo sustituye una palabra por otra
    - Borrados: Ocurre cuando el modelo se "salta" una palabra
    - Inserciones: Ocurre cuando el modelo "añade" una palabra que no existe en la locución
  - WER= (Suma de los tres tipos de errores / Número de palabras totales) * 100
  - Desventaja: Todos los errores reciben un peso similar en el cálculo

Score de modelos de ranking

Técnicas de score de modelos en Pytorch

- Los modelos de *rankings* son aquellos que ordenan unos documentos (imágenes, vídeos, enlaces, etc…) en base a una petición (*query)* determinada
  - El objetivo principal es que el usuario acceda a los documentos que aparecen en los primeros lugares
    - Es un síntoma de que el algoritmo de *ranking* funciona bien
- Ámbitos de aplicación
  - Sistemas recomendadores
    - Tiendas online
    - Plataformas de vídeo
  - Motores de búsqueda
  - Tiendas online

Técnicas de score de modelos en Pytorch

- *ClickThroughRate*: Calcula el ratio de veces que se accedió a un enlace

```
>>> import torch
>>> from torcheval.metrics.functional import click_through_rate
>>> input = torch.tensor([0, 1, 0, 1, 1, 0, 0, 1])
>>> click_through_rate(input)
tensor(0.5)
>>> weights = torch.tensor([1.0, 2.0, 1.0, 2.0, 1.0, 2.0, 1.0, 2.0])
>>> click_through_rate(input, weights)
tensor(0.58333)
>>> input = torch.tensor([[0, 1, 0, 1], [1, 0, 0, 1]])
>>> weights = torch.tensor([[1.0, 2.0, 1.0, 2.0],[1.0, 2.0, 1.0, 1.0]])
>>> click_through_rate(input, weights, num_tasks=2)
tensor([0.6667, 0.4])
```

Técnicas de score de modelos en Pytorch

- *Hit_rate:* Calcula el número de veces que se accedió a una clase de entre las *n* clasificadas en los primeros lugares

```
>>> import torch
>>> from torcheval.metrics.functional import hit_rate
>>> input = torch.tensor([[0.3, 0.1, 0.6], [0.5, 0.2, 0.3], [0.2, 0.1, 0.7], [0.3, 0.3,
0.4]])
>>> target = torch.tensor([2, 1, 1, 0])
>>> hit_rate(input, target, k=2)
tensor([1.0000, 0.0000, 0.0000, 1.0000])
```

Técnicas de score de modelos en Pytorch

- *reciprocal_rank:* Calcula el *rank* recíproco de la clase correcta respecto a las clases predichas con mayor probabilidad
  - *Reciprocal Rank: 1 dividido por la posición de la clase correcta en tu predicción*

```
>>> import torch
>>> from torcheval.metrics.functional import reciprocal_rank
>>> input = torch.tensor([[0.3, 0.1, 0.6], [0.5, 0.2, 0.3], [0.2, 0.1, 0.7], [0.3, 0.3,
0.4]])
>>> target = torch.tensor([2, 1, 1, 0])
>>> reciprocal_rank(input, target)
tensor([1.0000, 0.3333, 0.3333, 0.5000])
>>> reciprocal_rank(input, target, k=2)
tensor([1.0000, 0.0000, 0.0000, 0.5000])
```

Uso de torchmetrics

Técnicas de Score de modelos en Pytorch

- Existen varias implementaciones de estas métricas que se pueden usar dentro de Pytorch
- Lo más adecuado es utilizar la librería oficial para ello *torchmetrics*
  - Histórico
    - Versión 0.1.0 (febrero 2021)
    - Versión 1.0.0rc1 (29 de junio de 2023)
  - Implementa varias métricas existentes: https://torchmetrics.readthedocs.io/en/latest/all-metrics.html
    - Permite definir métricas propias a través de una interfaz estándar
  - Compatible con entrenamiento distribuido
    - Sincronización entre dispositivos
    - Acumulación a través de batches

Técnicas de Score de modelos en Pytorch

```python
from torchmetrics.classification import BinaryAccuracy

train_accuracy = BinaryAccuracy()
valid_accuracy = BinaryAccuracy()

for epoch in range(epochs):
    for x, y in train_data:
        y_hat = model(x)

        # training step accuracy
        batch_acc = train_accuracy(y_hat, y)
        print(f"Accuracy of batch{i} is {batch_acc}")

    for x, y in valid_data:
        y_hat = model(x)
        valid_accuracy.update(y_hat, y)

    # total accuracy over all training batches
    total_train_accuracy = train_accuracy.compute()

    # total accuracy over all validation batches
    total_valid_accuracy = valid_accuracy.compute()

    print(f"Training acc for epoch {epoch}: {total_train_accuracy}")
    print(f"Validation acc for epoch {epoch}: {total_valid_accuracy}")

    # Reset metric states after each epoch
    train_accuracy.reset()
    valid_accuracy.reset()
```

El API de torchmetrics proporciona para cada métrica varios métodos: *update()*, *compute()* y *reset()*

Técnicas de Score de modelos en Pytorch

Debemos mover las métricas al mismo dispositivo que la entrada de la métrica

Si definimos una métrica o varias dentro de un módulo, entonces se moverá automáticamente el mismo dispositivo que el módulo

```python
from torchmetrics.classification import BinaryAccuracy

target = torch.tensor([1, 1, 0, 0], device=torch.device("cuda", 0))
preds = torch.tensor([0, 1, 0, 0], device=torch.device("cuda", 0))

# Metric states are always initialized on cpu, and needs to be moved to
# the correct device
confmat = BinaryAccuracy().to(torch.device("cuda", 0))
out = confmat(preds, target)
print(out.device) # cuda:0
```

Técnicas de Score de modelos en Pytorch

Podemos usar MetricCollection para reunir varias métricas y calcularlas de forma conjunta durante el mismo entrenamiento

```python
from torchmetrics import MetricCollection
from torchmetrics.classification import MulticlassAccuracy, MulticlassPrecision, MulticlassRecall
target = torch.tensor([0, 2, 0, 2, 0, 1, 0, 2])
preds = torch.tensor([2, 1, 2, 0, 1, 2, 2, 2])
metric_collection = MetricCollection([
    MulticlassAccuracy(num_classes=3, average="micro"),
    MulticlassPrecision(num_classes=3, average="macro"),
    MulticlassRecall(num_classes=3, average="macro")
])
print(metric_collection(preds, target))
```

Otra opción: torcheval

- La librería torcheval de Pytorch también proporciona implementaciones de métricas de evaluación de modelos

https://pytorch.org/torcheval/main/metric_example.html

- Es un desarrollo muy reciente (Initial commit agosto, 2022)
  - 0.0.5 (octubre 2022)
  - 0.0.6 (enero 2023)
- Sigue también el modelo de *update(), reset(), compute()* seguido por torchmetrics

Laboratorio

- Revisa el notebook scores.ipynb
- Realiza el **Ejercicio** al final del notebook

TensorBoard en Pytorch

Tensorboard en Pytorch

- TensorBoard proporciona diversas herramientas de visualización relacionadas con el aprendizaje automático
  - Seguimiento y visualización de métricas como la pérdida (*loss*) o la precisión (*accuracy*)
  - Visualización del grafo de la arquitectura del modelo
  - Visualización de histogramas de pesos y bias en tiempo real
  - Visualización de métricas relacionadas con el rendimiento computacional (Tensorboard Profiler)
  - Aunque asociada al ecosistema TensorFlow, también disponible en Pytorch

Tensorboard en Pytorch

- Utilidades disponibles en el paquete *torch.utils.tensorboard*
- Permiten generar la información que se puede visualizar mediante Tensorboard
  - *Scalars*
  - Imágenes
  - Histogramas
  - Grafos (*graphs*)
  - *Embeddings*
- La clase *SummaryWriter* es el punto de entrada a través del cual registrar información

Tensorboard en Pytorch

```python
import torch
import torchvision
from torch.utils.tensorboard import SummaryWriter
from torchvision import datasets, transforms

# Writer will output to ./runs/ directory by default
writer = SummaryWriter()

transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])
trainset = datasets.MNIST('mnist_train', train=True, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)
model = torchvision.models.resnet50(False)
# Have ResNet model take in grayscale rather than RGB
model.conv1 = torch.nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3, bias=False)
images, labels = next(iter(trainloader))

grid = torchvision.utils.make_grid(images)
writer.add_image('images', grid, 0)
writer.add_graph(model, images)
writer.close()
```

Tensorboard en Pytorch

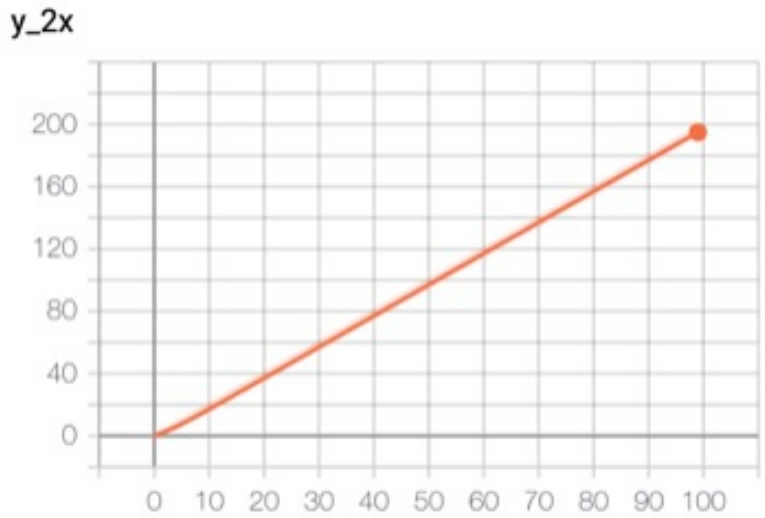
- La información se puede agrupar de forma jerárquica utilizando nombre de tipo:
  - Loss/train
  - Loss/test
  - Accuracy/train
  - Accuracy/test
- Para visualizar la información debemos instalar tensorboard y arrancarlos

```
pip install tensorboard

tensorboard --logdir=runs
```

```
from torch.utils.tensorboard import SummaryWriter
writer = SummaryWriter()
x = range(100)
for i in x:
    writer.add_scalar('y=2x', i * 2, i)
writer.close()
```

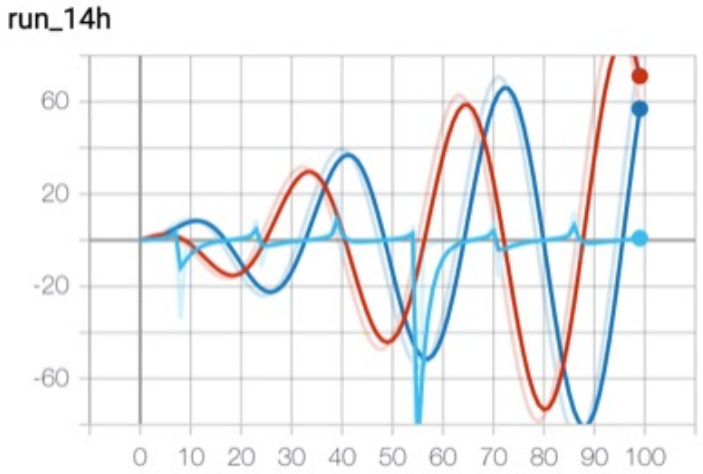
Expected result:



citic

```
from torch.utils.tensorboard import SummaryWriter
writer = SummaryWriter()
r = 5
for i in range(100):
    writer.add_scalars('run_14h', {'xsinx':i*np.sin(i/r),
                                   'xcosx':i*np.cos(i/r),
                                   'tanx': np.tan(i/r)}, i)
writer.close()
# This call adds three values to the same scalar plot with the tag
# 'run_14h' in TensorBoard's scalar section.
```

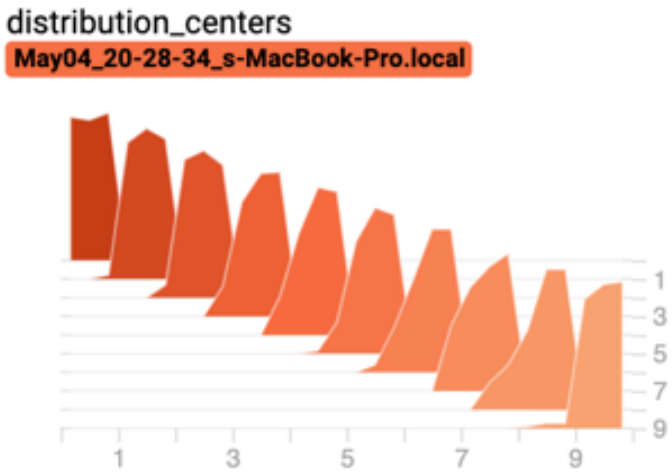
Expected result:



citic

```python
from torch.utils.tensorboard import SummaryWriter
import numpy as np
writer = SummaryWriter()
for i in range(10):
    x = np.random.random(1000)
    writer.add_histogram('distribution centers', x + i, i)
writer.close()
```

Expected result:



Tensorboard en Pytorch

- También tenemos métodos para
  - Añadir imágenes: *add_images*
  - Añadir una figura de matplotlib: *add_figure*
  - Otros medios: *add_video, add_audio, add_text*
  - *add_graph*
  - *add_embedding*
  - *add_pr_curve*
  - ...

Alternativa: Weight and Biases

- Existen también servicios para realizar el seguimiento de experimento a través de una plataforms de Software as a Service (Saas)
- El código se ejecuta en una máquina local, o en un supercomputador
- Los resultados se comunican al servicio a través de una API, usando un API key con llamadas que se introducen en el código Python
- Servicio popular: https://wandb.ai
  - Construyen dashboards para visualizar el entrenamiento de modelos

Otros servicios SaaS

- Neptune: https://neptune.ai
- Comet: https://www.comet.com
- Amazon SageMaker: https://aws.amazon.com/es/sagemaker/
- Frameworks open-source:
  - MLFlow (también paquete software open-source): https://mlflow.org
  - KubeFlow: https://www.kubeflow.org

Laboratorio

- Revisa el notebook tensorboard.ipynb
- Realiza el **Ejercicio** al final del notebook