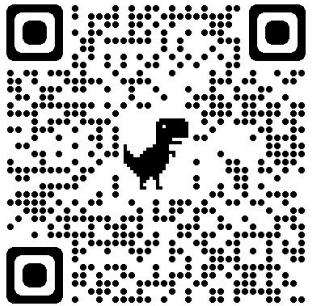
Guía para implementar experimentos de ML

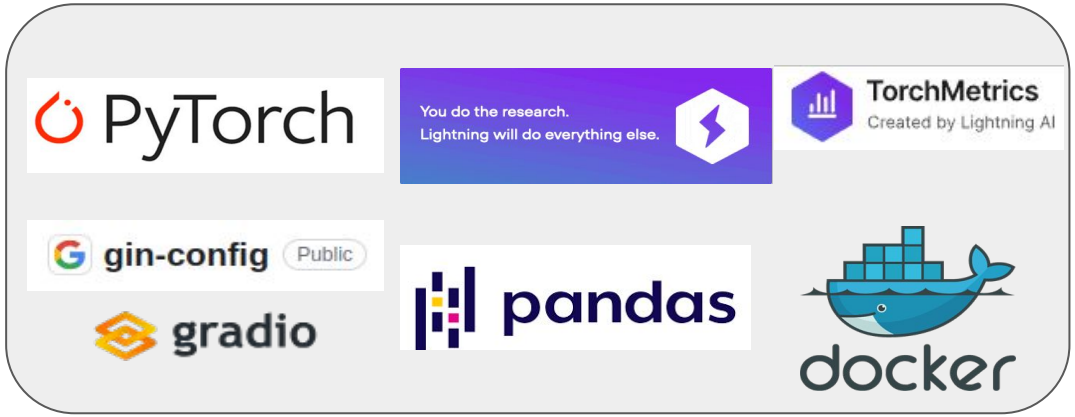
Leonardo Pepino



https://github.com/mrpep/ml-pipeline-tutorial

Disclaimer

- Es la metodología que mejor funciona para **MI hasta el momento.**
- Para una empresa de AI: le falta funcionalidades y capacidades para escalar.
- Para alguien que hace un tutorial o entrena ocasionalmente un modelo: probablemente sea un overhead innecesario.
- Para alguien que entrena regularmente modelos y hace research con una infraestructura de hardware no muy grande: probablemente sea adecuado.
- Librerías específicas pero intentaré que la idea sea general.
- Algunas cosas probablemente sean polémicas (no soy computólogo) o iluminadoras. Veremos…
- En las slides + el repo está TODO lo necesario para correr y entender el ejemplo. Es posible que no sea procesable en una hora de charla, no se traben leyendo código durante la presentación.

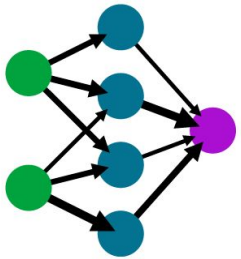


Pantallazo

Datos de entrenamiento

Dame una Instancia

Dame varias y te armo un batch



Hago algo con el batch para aprender. Normalmente:

- Paso el batch (x) por la red (forward)
- Calculo la loss entre lo que sale y los labels (y).
- Propago error y ajusto pesos con optimizador (backward)

Pantallazo

Datos de entrenamiento

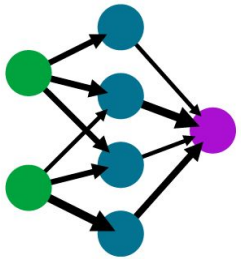
pd.DataFrame

torch.utils.data .Dataset

Dame una Instancia

torch.utils.data .DataLoader

Dame varias y te armo un batch



pl.LightningModule

pl.Trainer

Hago algo con el batch para aprender. Normalmente:

- Paso el batch (x) por la red (forward)
- Calculo la loss entre lo que sale y los labels (y).
- Propago error y ajusto pesos con optimizador (backward)

Pantallazo

Datos de entrenamiento

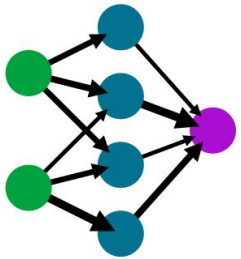
pd.DataFrame

torch.utils.data .Dataset

Dame una Instancia

torch.utils.data .DataLoader

Dame varias y te armo un batch



pl.LightningModule

pl.Trainer

Hago algo con el batch para aprender. Normalmente:

- Paso el batch (x) por la red (forward)
- Calculo la loss entre lo que sale y los labels (y).
- Propago error y ajusto pesos con optimizador (backward)

Repito hasta que termina una epoch o K steps. Luego corro validación, guardo checkpoints, etc… Repito

Pasos

1) Setear seeds
2) Cargar metadata de datasets
3) Armar datasets y dataloaders
4) Entrenar modelo
5) Evaluar modelo

Problema específico: entrenar un clasificador de género musical

Setear seeds

```python
import random
import numpy as np
import torch

def set_seed(state, seed=42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    state.seed = seed
    return state
```

Setear seeds

```python
import random
import numpy as np
import torch

def set_seed(state, seed=42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    state.seed = seed
    return state
```

¿Qué rayos es **state**?

Quiero uniformizar cómo se comparten datos entre funciones del pipeline. Una forma es usando un diccionario en el que cada función puede guardar algo (que luego será o no utilizado por otras funciones), y también cada función puede leer datos de ahí. Toda función del pipeline va a recibir y devolver un **state.**

Si ahora no se entiende, **no se preocupen**, más tarde cuando veamos cómo se conecta todo se va a entender.

Cargar metadata de datasets

- Esto va a ser distinto para cada dataset, para cada modalidad (texto, audio, imagen, etc…).
- Acá vamos a apegarnos al ejemplo del clasificador de género musical.

GTZAN sigue una estructura de directorio:

gtzan_path/rock/rock.00097.wav

Cargar metadata de datasets

```python
from pathlib import Path
from tqdm import tqdm
import soundfile as sf
import pandas as pd

def load_dataset(state, reader_fn,
                 cache=True,
                 filters=[],
                 key_out='dataset_metadata'):

    if not (cache and key_out in state):
        if not isinstance(reader_fn, list):
            reader_fn = [reader_fn]
        dfs = [fn() for fn in reader_fn]
        df = pd.concat(dfs).reset_index()
        state[key_out] = df
    else:
        logger.info('Caching dataset metadata from state')

    for f in filters:
        state[key_out] = f(state[key_out])

    return state
```

```python
def load_gtzan(data_dir):
    all_wavs = Path(data_dir).rglob('*.wav')
    metadata = []
    for w in tqdm(all_wavs):
        wav_info = sf.info(w)
        mi = {'filename': str(w.resolve),
              'frames': wav_info.frames,
              'duration': wav_info.duration,
              'sample_id': w.stem.split('.')[1],
              'genre': w.stem.split('.')[0],
              'dataset': 'gtzan'}
        metadata.append(mi)
    return pd.DataFrame(metadata)
```

Cargar metadata de datasets

```python
from pathlib import Path
from tqdm import tqdm
import soundfile as sf
import pandas as pd

def load_dataset(state, reader_fn,
                 cache=True,
                 filters=[],
                 key_out='dataset_metadata'):

    if not (cache and key_out in state):
        if not isinstance(reader_fn, list):
            reader_fn = [reader_fn]
        dfs = [fn() for fn in reader_fn]
        df = pd.concat(dfs).reset_index()
        state[key_out] = df
    else:
        logger.info('Caching dataset metadata from state')

    for f in filters:
        state[key_out] = f(state[key_out])

    return state
```

Acá mando una lista de funciones que devuelvan dataframes.

Acá combino los dataframes (multiples datasets)

Finalmente guardo el dataframe en el estado.

Puedo mandar funciones para descartar por ejemplo audios más largos que X duración.

```python
def load_gtzan(data_dir):
    all_wavs = Path(data_dir).rglob('*.wav')
    metadata = []
    for w in tqdm(all_wavs):
        wav_info = sf.info(w)
        mi = {'filename': str(w.resolve),
              'frames': wav_info.frames,
              'duration': wav_info.duration,
              'sample_id': w.stem.split('.')[1],
              'genre': w.stem.split('.')[0],
              'dataset': 'gtzan'}
        metadata.append(mi)
    return pd.DataFrame(metadata)
```

En este ejemplo llamaría load_dataset(state, [load_gtzan])

Cargar metadata de datasets

```python
from pathlib import Path
from tqdm import tqdm
import soundfile as sf
import pandas as pd

def load_dataset(state, reader_fn,
                 cache=True,
                 filters=[],
                 key_out='dataset_metadata'):

    if not (cache and key_out in state):
        if not isinstance(reader_fn, list):
            reader_fn = [reader_fn]
        dfs = [fn() for fn in reader_fn]
        df = pd.concat(dfs).reset_index()
        state[key_out] = df
    else:
        logger.info('Caching dataset metadata from state')

    for f in filters:
        state[key_out] = f(state[key_out])

    return state
```

¿Dónde estoy pasándole los argumentos? Ej: data_dir

¿Dónde estoy pasándole los argumentos? Ej: máxima duración.

```python
def load_gtzan(data_dir):
    all_wavs = Path(data_dir).rglob('*.wav')
    metadata = []
    for w in tqdm(all_wavs):
        wav_info = sf.info(w)
        mi = {'filename': str(w.resolve),
              'frames': wav_info.frames,
              'duration': wav_info.duration,
              'sample_id': w.stem.split('.')[1],
              'genre': w.stem.split('.')[0],
              'dataset': 'gtzan'}
        metadata.append(mi)
    return pd.DataFrame(metadata)
```


WTF!?

Archivos de Configuración

- Filosofía: desacoplar código y argumentos con los que se llaman funciones.

```
set_seed(seed=666)
load_dataset(reader_fn=[load_gtzan], filters=[limit_max_audio_duration])
load_gtzan(data_dir='/mnt/data/gtzan')
state[key_out] = limit_max_audio_duration(state[key_out], max_duration=10)
```

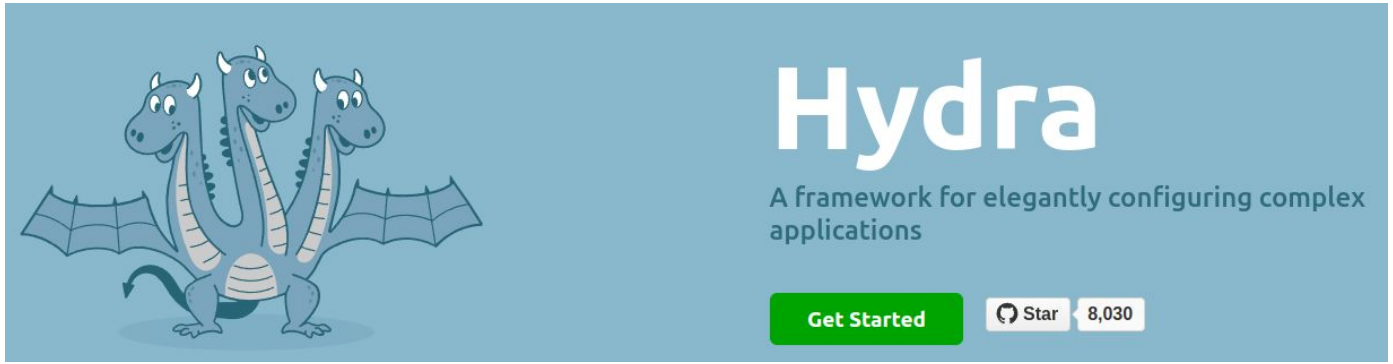
Código .py

```
set_seed()
load_dataset()
```

Archivo de configuración

```
set_seed.seed=666
load_dataset:
    reader_fn = [@load_gtzan]
    filters = [@limit_max_audio_duration]
load_gtzan.data_dir='/mnt/data/gtzan'
limit_max_audio_duration.max_duration=10
```

Archivos de Configuración



gin-config Public

fiddle Public

YACS

YAML

Welcome to the Gin

- Configura los default args y kwargs de funciones según un config.

```
@gin.configurable
def dnn(inputs,
        num_outputs,
        layer_sizes=(512, 512),
        activation_fn=tf.nn.relu):
  ...
```

```
# Inside "config.gin"
dnn.layer_sizes = (1024, 512, 128)
```

- También funciona con clases.

```
@gin.configurable
class DNN(object):
  # Constructor parameters become configurable.
  def __init__(self,
               num_outputs,
               layer_sizes=(512, 512),
               activation_fn=tf.nn.relu):
    ...

  def __call__(inputs):
    ...
```

```
# Inside "config.gin"
DNN.layer_sizes = (1024, 512, 128)
```

Welcome to the Gin

- Con solo agregar el decorador a cada función y cargar el config antes de llamar a las funciones, pudimos modificar la manera en la que se llama cada función.

```
gin.parse_config_file('config.gin')
```

- También se puede pasar funciones o clases en el config:

```
# Inside "config.gin"
dnn.activation_fn = @tf.nn.tanh
```

- O pasar el resultado de evaluarla:

```
# Inside "config.gin"
build_model.network_fn = @DNN()
```

Welcome to the Gin

- A veces queremos llamar una misma función / instanciar un objeto, varias veces en distintos lugares con distintos argumentos. Gin provee scopes.

```
# Inside "config.gin"
build_model.generator_network_fn = @generator/dnn
build_model.discriminator_network_fn = @discriminator/dnn

generator/dnn.layer_sizes = (128, 256)
generator/dnn.num_outputs = 784

discriminator/dnn.layer_sizes = (512, 256)
discriminator/dnn.num_outputs = 1

dnn.activation_fn = @tf.nn.tanh
```

Welcome to the Gin

- A veces queremos definir una variable y utilizarla en varios lugares del config. Gin provee Macros.

```
num_layers = 10
network.num_layers = %num_layers
```

- Otras funcionalidades son:
  - Reutilizar una instancia en varios lugares de un config: Singletons.
  - Printear bonitamente el config resultante.
  - Registrar funciones/clases de librerías

Gin -> GinPipe

Para adecuar a mi pipeline algunas cosas de Gin, cree GinPipe. Algunas mejoras son:

- Permite registrar librerías enteras, ejemplo torch, y ser utilizadas en el config sin tener que decorar.
- Permite componer múltiples archivos de configuración y modificar/agregar variables desde la consola.
- Ejecuta una lista de funciones, va pasando el state entre ellas, y guardando el resultado en una carpeta del experimento.
- Tiene algunos sugar syntaxs.
- Tiene un logging bonito, y los configs finales quedan guardados en la carpeta del experimento.
- Permite cachear resultados guardados en el estado.

Organizando el proyecto

```
├── setup.cfg
├── setup.py
├── tutorial_ml
│   ├── configs
│   │   ├── base
│   │   │   └── train_dnn.gin
│   │   ├── datasets
│   │   │   └── gtzan_genre.gin
│   │   └── imports
│   ├── scripts
│   │   └── run.sh
│   └── tasks
│       └── __init__.py
```

Hacemos que sea una librería de Python.
En tasks metemos las funciones que definimos

configs/base/train_dnn.gin

```
SEED=42

execute_pipeline:
    tasks = [@tasks.set_seed,
             @tasks.load_dataset]
    execution_order = 'sequential'

set_seed.seed=%SEED
```

configs/datasets/gtzan_genre.gin

```
tasks.load_dataset.reader_fn+=[@tasks.load_gtzan]
tasks.load_gtzan.data_dir='/mnt/ssd4T/datasets/gtzan'
```

configs/imports

```
tutorial_ml.tasks: tasks
```

Finalmente llamo a ginpipe asi:

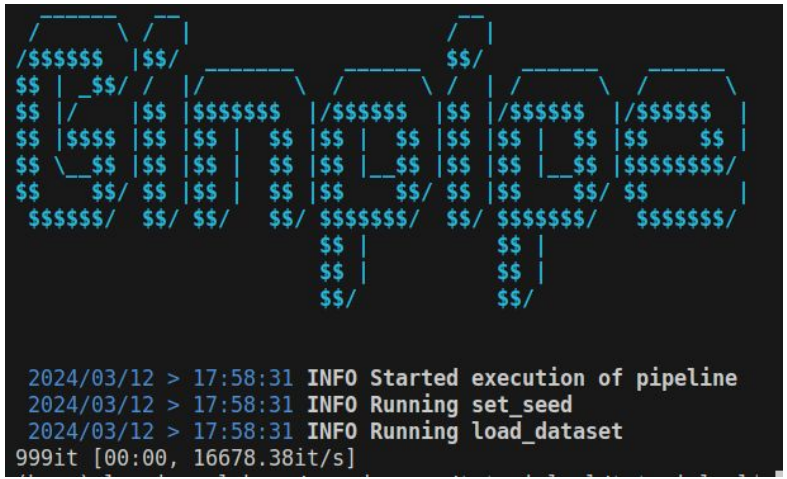
```
ginpipe configs/base/train_dnn.gin \
        configs/datasets/gtzan_genre.gin \
        --module_list configs/imports \
        --project_name tutorial \
        --experiment_name gtzan
```

Ejecutando GinPipe

1)

```
ginpipe configs/base/train_dnn.gin \
        configs/datasets/gtzan_genre.gin \
        --module_list configs/imports \
        --project_name tutorial \
        --experiment_name gtzan
```

2)



3) Aparece esto:

Ejecutando GinPipe: Artefactos

config.gin

```
# Macros:
# ==============================================================
EXPERIMENT_NAME = 'gtzan'
OUTPUT_DIR = 'experiments/tutorial/gtzan'
PROJECT_NAME = 'tutorial'
SEED = 42

# Parameters for execute_pipeline:
# ==============================================================
execute_pipeline.execution_order = 'sequential'
execute_pipeline.tasks = [@tasks.set_seed, @tasks.load_dataset]

# Parameters for load_dataset:
# ==============================================================
load_dataset.reader_fn = [@tasks.load_gtzan]

# Parameters for load_gtzan:
# ==============================================================
load_gtzan.data_dir = '/mnt/ssd4T/datasets/gtzan'

# Parameters for set_seed:
# ==============================================================
set_seed.seed = %SEED
```

Ejecutando GinPipe: Artefactos

```
In [1]: import joblib

In [2]: state = joblib.load('experiments/tutorial/gtzan/state.pkl')

In [3]: state.keys()
Out[3]: dict_keys(['flags', 'output_dir', 'operative_config', 'config_str', 'seed', 'execution_times', 'dataset_metadata

In [4]: state['dataset_metadata']
Out[4]:
     index                                          filename  frames   duration sample_id      genre dataset
0        0        /mnt/ssd4T/datasets/gtzan/pop/pop.00038.wav  720005  30.000208     00038        pop   gtzan
1        1        /mnt/ssd4T/datasets/gtzan/pop/pop.00014.wav  720005  30.000208     00014        pop   gtzan
2        2        /mnt/ssd4T/datasets/gtzan/pop/pop.00042.wav  720005  30.000208     00042        pop   gtzan
3        3        /mnt/ssd4T/datasets/gtzan/pop/pop.00053.wav  720005  30.000208     00053        pop   gtzan
4        4        /mnt/ssd4T/datasets/gtzan/pop/pop.00017.wav  720005  30.000208     00017        pop   gtzan
..     ...                                                ...     ...        ...       ...        ...     ...
994    994  /mnt/ssd4T/datasets/gtzan/classical/classical....  720320  30.013333     00013  classical   gtzan
995    995  /mnt/ssd4T/datasets/gtzan/classical/classical....  720192  30.008000     00043  classical   gtzan
996    996  /mnt/ssd4T/datasets/gtzan/classical/classical....  720320  30.013333     00044  classical   gtzan
997    997  /mnt/ssd4T/datasets/gtzan/classical/classical....  720320  30.013333     00084  classical   gtzan
998    998  /mnt/ssd4T/datasets/gtzan/classical/classical....  720320  30.013333     00062  classical   gtzan

[999 rows x 7 columns]
```

Ejecutando GinPipe: Mods

- Supongamos que queremos correr exactamente el mismo experimento pero con otro seed. Facil:

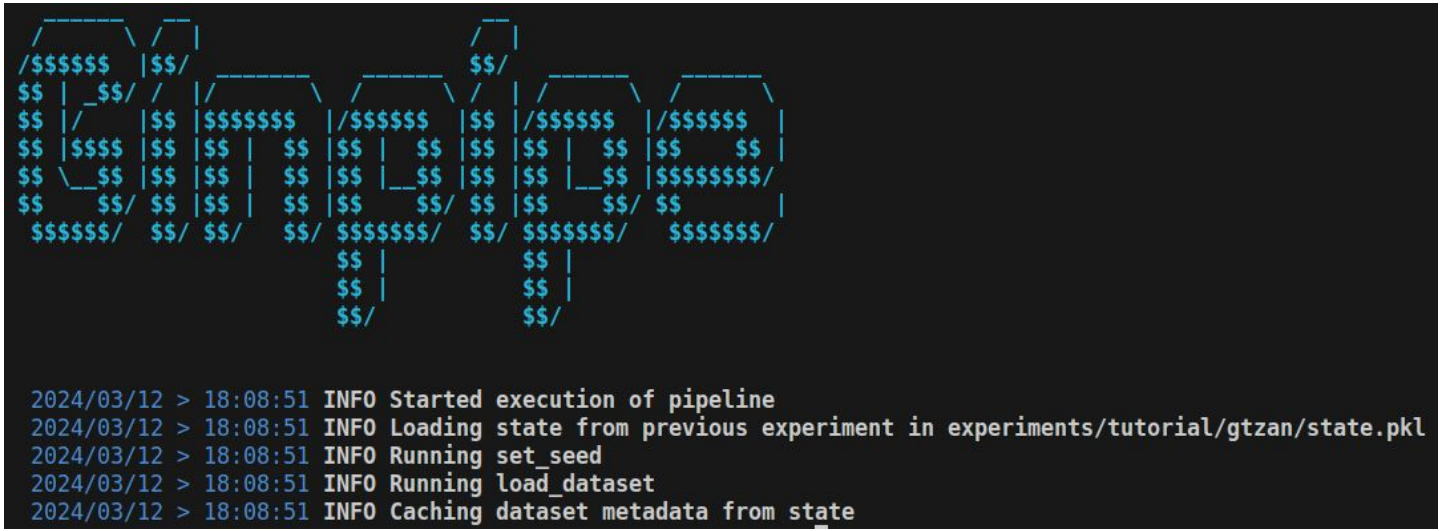
```
ginpipe configs/base/train_dnn.gin \
        configs/datasets/gtzan_genre.gin \
        --module_list configs/imports \
        --project_name tutorial \
        --experiment_name gtzan \
        --mods SEED=10
```

- Podemos poner tantos mods como querramos. Si fueran muchos conviene armar un nuevo config con mods y agregarlo al llamado de ginpipe.

- El orden de configs importa!

Ejecutando GinPipe: Caching

- Si ejecutamos lo mismo de nuevo:



- Ahora agarra el state previo y en load_dataset como ya esta el dataframe, cachea.
- El state se guarda al finalizar la ejecucion de cada funcion, por ende si se corta la luz zafamos.

Volvamos al Pipeline

Dataset: Partición y labels

```python
def partition_by_re(state, column_in, key_in='dataset_metadata', res=None, column_out='partition'):
    def match(x, res):
        for k,v in res.items():
            if re.match(v,x):
                return k

    state[key_in][column_out] = state[key_in][column_in].apply(lambda x: match(x, res))
    return state
```

Uso para validación instancias terminadas en 8 y para test en 9.

```python
def make_labels(state, column_in, key_in='dataset_metadata', key_out='class_map', column_out='classID'):
    class_map = {c: i for i,c in enumerate(sorted(state[key_in][column_in].unique()))}
    state[key_out] = class_map
    state[key_in][column_out] = state[key_in][column_in].apply(lambda x: class_map[x])
    return state
```

Actualizo configs

```
tasks.load_dataset.reader_fn+=[@tasks.load_gtzan]
tasks.load_gtzan.data_dir='/mnt/ssd4T/datasets/gtzan'
tasks.partition_by_re:
    column_in='sample_id'
    res = {'validation': '.*8$', 'test': '.*9$', 'train': '^.*[^89]$'}
tasks.make_labels:
    column_in='genre'
```

```
execute_pipeline:
    tasks = [@tasks.set_seed,
             @tasks.load_dataset,
             @tasks.partition_by_re,
             @tasks.make_labels]
```

Dataset: Partición y labels

```
classID
7    100
2    100
8    100
9    100
6    100
4    100
3    100
0    100
1    100
5     99
Name: count, dtype: int64
```

```
'class_map': {'blues': 0,
 'classical': 1,
 'country': 2,
 'disco': 3,
 'hiphop': 4,
 'jazz': 5,
 'metal': 6,
 'pop': 7,
 'reggae': 8,
 'rock': 9}}
```

Dataset: Dame una instancia

Datos

Index	Filename	Duration	MusicGenre	SR	Partition	ClassID
GTZAN_000	xxx.wav	10	Rock	44100	Train	1
GTZAN_001	xxx.wav	15	Jazz	44100	Train	2
…	…	…	…	…		
GTZAN_999	xxx.wav	12	Classical	44100	Test	8

999

processors = [read_audio]

torch.utils.data.Dataset

GTZAN_999	xxx.wav	12	Classical	44100	Test	8

Aplico processors

{‘wav’: [0.2,0.1,-0.1,...],’classID’: 3}

Dataset: Dame una instancia

```python
class DataFrameDataset(Dataset):
    def __init__(self, metadata, out_cols, preprocessors=None):
        self._metadata = metadata
        self._out_cols = out_cols
        self._preprocessors = [p() for p in preprocessors]

    def __getitem__(self, idx):
        row = copy.deepcopy(self._metadata.iloc[idx])
        for p in self._preprocessors:
            row = p(row)
        out = {k: row[k] for k in self._out_cols}
        return out

    def __len__(self):
        return len(self._metadata)
```

Un problemita de Gin

Si uno quiere picklear un objeto que tiene guardado otro objeto configurado con Gin, falla. Ejemplo: si en vez de hacer una clase ProcessorReadAudio, lo implementaba como función, esa función está wrappeada por gin y al guardarla en self.preprocessors se rompe la serialización. En cambio si es una clase, y la instancio dentro de mi clase, al instanciarla deja de estar wrappeada por Gin y se resuelve.

```python
class ProcessorReadAudio:
    def __init__(self, max_duration=None,
                       key_in='filename',
                       key_out='wav',
                       key_duration='duration',
                       key_sr='sr'):

        self.max_duration = max_duration
        self.key_in, self.key_out, self.key_duration, self.key_sr = key_in, key_out,

    def __call__(self, row):
        if self.max_duration is not None:
            dur = row[self.key_duration]
            if dur > self.max_duration:
                max_frames = int(self.max_duration*row[self.key_sr])
                frames = int(dur*row[self.key_sr])
                start = random.randint(0, frames-max_frames)
                stop = start + max_frames
        else:
            start=0
            stop=None

        x, fs = sf.read(row[self.key_in], start=start, stop=stop, dtype=np.float32)
        row[self.key_out] = x
        return row
```

Dataset: Dame una instancia

configs/features/wav_classification.gin

```
tasks.DataFrameDataset:
    out_cols=['wav', 'classID']
train/tasks.DataFrameDataset.preprocessors=[@train/tasks.ProcessorReadAudio]
val/tasks.DataFrameDataset.preprocessors=[@val/tasks.ProcessorReadAudio]

train/tasks.ProcessorReadAudio:
    max_duration = %MAX_AUDIO_DURATION
val/tasks.ProcessorReadAudio:
    max_duration = None
```

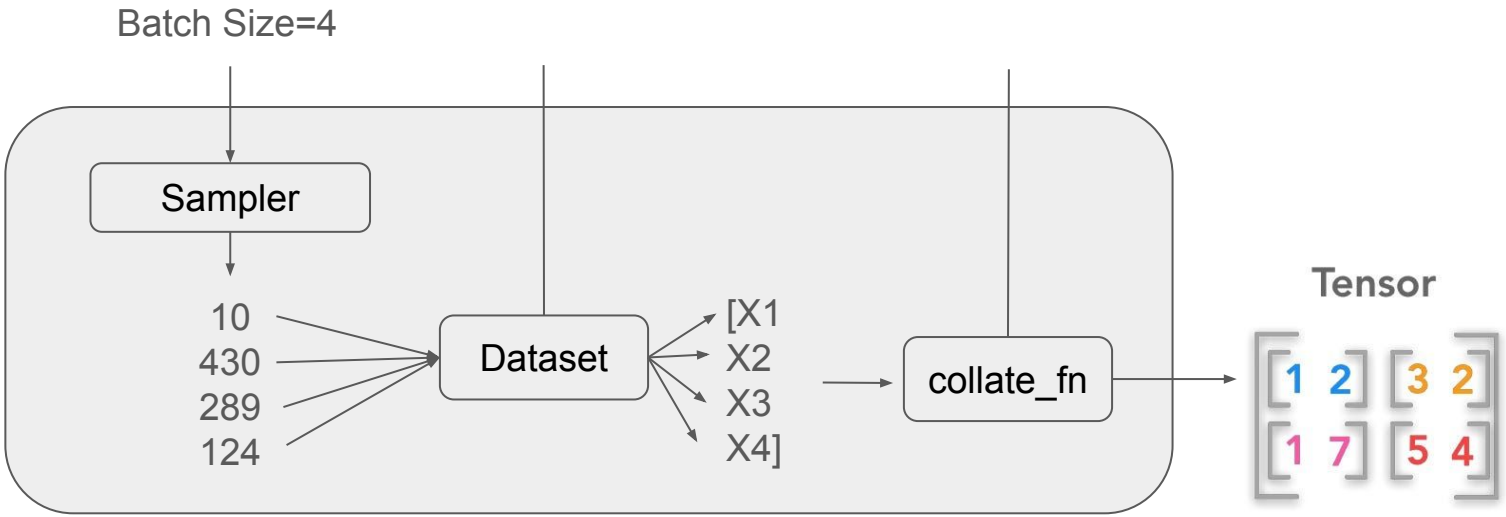
Config con especificación de cómo armar una instancia: Leo audio agarrando columna filename de la fila de mi dataframe, y escupo lo que sale en wav. Termino devolviendo un dict con las claves wav y classID.

En train agarro segmentos de MAX_AUDIO_DURATION, en val agarro el audio entero.

Dataset: Dame una instancia

```
In [5]: state['datasets']['train'][0]
Out[5]:
{'wav': array([-0.08007812, -0.08843994, -0.1008606 , ...,  0.02798462,
         0.02246094,  0.02178955], dtype=float32),
 'classID': 7}

In [6]: state['datasets']['train'][0]['wav'].shape
Out[6]: (96000,)

In [7]: state['datasets']['validation'][0]['wav'].shape
Out[7]: (720005,)
```

DataLoader: Armame un batch



- El sampler puede ser ordenado si no hacemos shuffling.
- Collate_fn por default stackea y convierte numpy arrays a torch tensors. Si las dimensiones no coinciden, hay que implementar un collate_fn custom.
- En mi ejemplo uso un collate_fn BatchDynamicPadding que busca el tensor con largo maximo del batch y paddea el resto a esa longitud. A su vez guarda las longitudes originales como claves extras en el dict.

Detallecitos

- Si un dataset es muy grande no va a entrar todo en RAM y tenemos que ir leyéndolo de disco. Importante tener los datos en un disco rápido (SSD, NVME)
- Suele haber un tradeoff CPU/IO. Ejemplo: si guardo mis datos en .wav van a ocupar más espacio y más tiempo de lectura (-CPU +IO). A su vez, si los guardo comprimidos, van a ocupar menos pero tengo que decodificarlos (+CPU -IO).
- Usar multiprocessing. Permite hacer procesamientos pesados on the fly (ej Data Augmentation) sin causar bottlenecks. PyTorch Dataloader lo maneja solo.
- Cada worker en un dataloader va a incrementar la memoria utilizada (el dataset se replica en cada proceso)
- prefetch_factor permite controlar cuantos batches por adelantado arma cada worker. Puede servir si hay un bottleneck, pero ojo que también come más memoria.

Dejando listos los dataloaders

```python
def get_dataloaders(state, dataset_cls, dataloader_cls,
                    key_dataset='dataset_metadata',
                    column_partition='partition',
                    key_dataset_out='datasets',
                    key_dataloaders_out='dataloaders'):

    df_metadata = state[key_dataset]
    state[key_dataset_out] = {k: v(df_metadata.loc[df_metadata[column_partition]==k]) for k,v in dataset_cls.items()}
    state[key_dataloaders_out] = {k: v(state[key_dataset_out][k]) for k,v in dataloader_cls.items()}

    return state
```

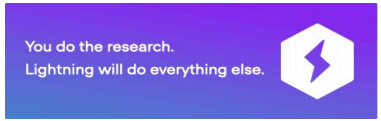
Esta función recibe el dataset y dataloader de train/val, como diccionarios {‘train’: dataset_cls_train, ‘val’: dataset_cls_val}
Luego las instancia pasando la metadata correspondiente a train y val. Eso lo guarda en el estado.

Lo lindo es que me quedan guardados los datasets y dataloaders.

Dejando listos los dataloaders

```
execute_pipeline:
    tasks = [@tasks.set_seed,
             @tasks.load_dataset,
             @tasks.partition_by_re,
             @tasks.make_labels,
             @tasks.get_dataloaders,
             @tasks.fit_model]
    execution_order = 'sequential'

tasks.get_dataloaders.dataset_cls={'train': @train/tasks.DataFrameDataset, 'validation': @val/tasks.DataFrameData
tasks.get_dataloaders.dataloader_cls={'train': @train/torch.utils.data.DataLoader, 'validation': @val/torch.utils

train/torch.utils.data.DataLoader:
    batch_size = %TRAIN_BATCH_SIZE
    shuffle =True
val/torch.utils.data.DataLoader:
    batch_size = %VAL_BATCH_SIZE
    shuffle = False

torch.utils.data.DataLoader:
    collate_fn=@tasks.BatchDynamicPadding()
```

Es hora de armar el modelo



- Pytorch Lightning es un wrapper de PyTorch que simplifica algunas cosas sin perder control de bajo nivel.

pl.LightningModule

Hay que definir 2 métodos:

- training_step(batch, batch_idx): recibe el **batch** y debe devolver el valor de la **loss**.
- configure_optimizers(): debe devolver el **optimizador** (y LR scheduler).

Luego hacemos trainer.fit(modulo, dataloader_train, dataloader_val) y estamos entrenando.

pl.Trainer

Define la lógica de entrenamiento. Entre otras cosas le podemos decir:

- Qué device usar para entrenar. Podemos poner varios y hace solo el DDP.
- Mixed Precision
- Cantidad de épocas/steps
- Callbacks
- Gradient Accumulation
- Optimizaciones: deepspeed, etc…
- Loggers

Código minimal

```python
class AudioClassifier(pl.LightningModule):
    def __init__(self, optimizer,
                 loss=torch.nn.functional.cross_entropy):

        super().__init__()
        self.optimizer=optimizer
        self.loss=loss

    def training_step(self, batch, batch_idx):
        yhat = self(batch['wav'])
        y = batch['classID']
        loss = self.loss(yhat, y)
        self.log('train_loss', loss)
        return loss

    def validation_step(self, batch):
        yhat = self(batch['wav'])
        y = batch['classID']
        loss = self.loss(yhat, y)
        self.log('val_loss', loss)

    def configure_optimizers(self):
        return self.optimizer(self.trainer.model.parameters())
```

```python
class CNN1D(AudioClassifier):
    def __init__(self, optimizer, loss=None, metrics=None,
                 cnn_layer=torch.nn.Conv1d,
                 channels=[64,128,128,256,256,512],
                 kernel_sizes=[16,16,8,8,4,4], strides=[4,4,2,2,2,2],
                 pooling_type='mean', classification_layer=torch.nn.Linear,
                 num_classes=None, key_in='wav'):

        super().__init__(optimizer, loss)
        ch_ins = [1] + channels[:-1]
        ch_outs = channels

        self.encoder = torch.nn.Sequential(*[cnn_layer(ci,co,k,s) for ci,co,k,s in
        self.classification_layer = classification_layer(channels[-1], num_classes)
        self.pooling_type = pooling_type
        self.key_in = 'wav'

    def forward(self, x):
        #x (BS, T)
        embeddings = self.encoder(x.unsqueeze(1)) # (BS, C, T)
        embeddings = embeddings.mean(axis=-1) # (BS, C)
        return self.classification_layer(embeddings) # (BS, num_classes)
```

En el repo hay versión con más chiches: métricas, lr_scheduler, etc…

Luego en el config…

Paso todos los args al trainer

```
tasks.fit_model.trainer_cls = @pl.Trainer

pl.Trainer:
    logger=@pl.loggers.CSVLogger()
    devices=%DEVICE
    callbacks=[@pl.callbacks.ModelCheckpoint(), @pl.callbacks.LearningRateMonitor()]
    accelerator='gpu'
    accumulate_grad_batches=%GRAD_ACC
    num_sanity_val_steps=1
    precision=%PRECISION
    max_epochs=%TRAINING_EPOCHS
pl.callbacks.ModelCheckpoint:
    dirpath=%OUTPUT_DIR
    save_top_k=2 #Keep best 2 checkpoints
    monitor=%MONITOR_METRIC
    mode=%MONITOR_MODE
pl.loggers.CSVLogger:
    save_dir=%OUTPUT_DIR
    name='training_logs'
```

Y armo un config para el modelo

```
tasks.fit_model.model_cls=@tasks.models.CNN1D
tasks.models.CNN1D:
    optimizer=@torch.optim.AdamW
    loss=@torch.nn.functional.cross_entropy
    metrics=[@torchmetrics.classification.MulticlassAccuracy]
    classification_layer=@tasks.models.MLP
    cnn_layer=@tasks.models.Conv1DNormAct
tasks.models.MLP:
    hidden_dims=[256,128]
tasks.models.Conv1DNormAct:
    activation=@torch.nn.ReLU
    normalization=@torch.nn.BatchNorm1d
torch.optim.AdamW:
    lr=%MAX_LR
    betas=(0.9,0.95)
    weight_decay=0.05
```

Función para fittear el modelo

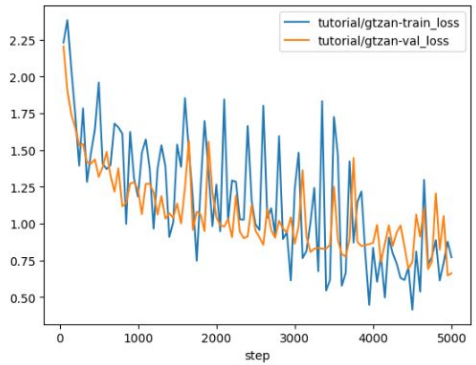
```python
def fit_model(state, model_cls=None, trainer_cls=None,
              key_dataloaders='dataloaders', key_out = 'model',
              from_checkpoint=None, checkpoint_folder='checkpoints'):

    #Automatically pass number of classes to model:
    if 'num_classes' in inspect.signature(model_cls.__init__).parameters and 'class_map' in state:
        kwargs = {'num_classes': len(state['class_map'])}
    else:
        kwargs = {}

    model = model_cls(**kwargs)
    trainer = trainer_cls()
    trainer.checkpoint_callback.dirpath = trainer.checkpoint_callback.dirpath + '/{}'.format(checkpoint_folder)
    trainer.fit(model,
                state[key_dataloaders]['train'],
                state[key_dataloaders]['validation'],
                ckpt_path=from_checkpoint)

    state[key_out] = model
    return state
```

Monitoreo

```
import pandas as pd
import matplotlib.pyplot as plt
import time
from IPython.display import clear_output
```

```
def monitor_metrics(experiments, x, y):
    while True:
        clear_output()
        exp_metrics = {}
        for exp in experiments:
            exp_metrics[exp] = pd.read_csv('experiments/{}/training_logs/version_0/metrics.csv'.format(exp))
        for panel_x, panel_y in zip(x,y):
            plt.figure()
            for k,v in exp_metrics.items():
                for yi in panel_y:
                    idx = ~v[yi].isna()
                    xval = v.loc[idx][panel_x]
                    yval = v.loc[idx][yi]
                    plt.plot(xval, yval, label='{}-{}'.format(exp, yi))
            plt.xlabel(panel_x)
            plt.legend()
            plt.show()
        plt.close()
        time.sleep(10)
```

```
monitor_metrics(['tutorial/gtzan'], x=['step','step'], y=[['train_loss','val_loss'],['train_MulticlassAccuracy','val_MulticlassAccuracy']])
```



Old School

- Las métricas quedan en un csv en mi compu (fáciles de leer en cualquier sistema y emprolijar para paper).
- Puedo customizar los plots como quiera.
- Overhead casi nulo al loggear.
- El smoothing se implementa fácil (ver scipy.signal.savgol_filter)
- Con un while se refresca con la frecuencia que quiera en un notebook.

New School

- Tensorboard
- Weights and Biases -> ojo que el overhead es perceptible.
- Neptune

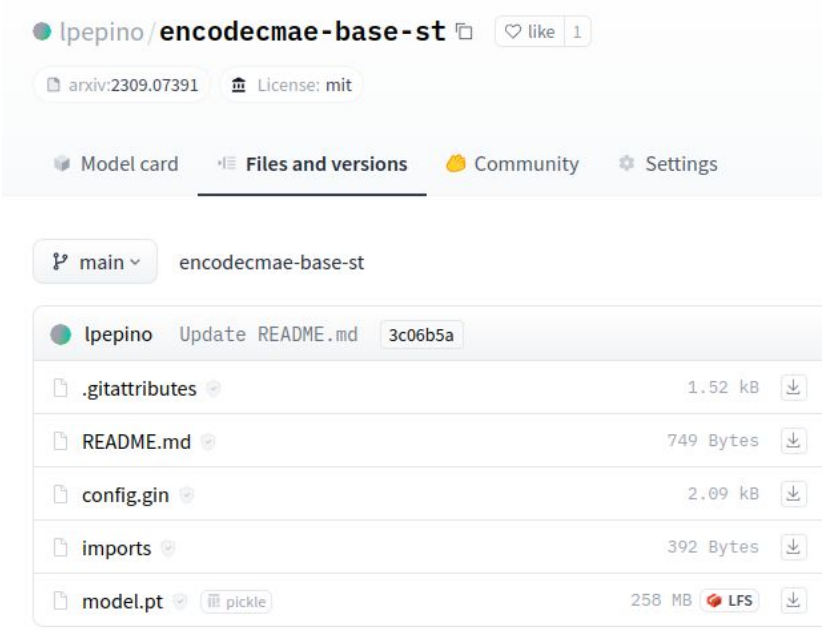
Test/Métricas

```
ml_tasks.eval_model:
    metrics=[@sklearn.metrics.accuracy_score, @sklearn.metrics.confusion_matrix]
```

Puedo facilmente mandarle metricas de sklearn.
Internamente uso inspect.signature para ver si la metrica espera y_pred o y_score y en base a eso mando decisiones o scores.

- Durante entrenamiento uso TorchMetrics.
- Es fácil loggear desde Pytorch-Lightning.
- Tiene bastantes métricas y puede calcularlas en GPU directamente.

Inferencia



Modelo como config de gin + pesos + imports. Storage en HFHub.

lpepino Create config.gin d1983e2

raw history blame edit delete No virus

```
NUM_ENCODEC_TARGETS=8
NUM_TOTAL_TARGETS=8
NUM_TARGET_TOKENS=1024
MASK_AMOUNT=150
MASK_GAP_SIZE=15
MASK_PROP=0.5
MODEL_DIM=768
NUM_ENCODER_LAYERS=10
NUM_ENCODER_HEADS=12
NUM_DECODER_LAYERS=2
NUM_DECODER_HEADS=12
MASKED_LOSS_WEIGHT=0.9
get_model.model=@models.EncodecMAE
models.EncodecMAE:
    wav_encoder = @models.encodecmae.encoders.EncodecEncoder
    target_encoder = @models.encodecmae.targets.EncodecQuantizer
    masker = @models.encodecmae.masking.TimeGapMask
    visible_encoder = @encoder/models.transformers.TransformerEncoder
    positional_encoder = @models.transformers.SinusoidalPositionalEmbeddings
    decoder = @decoder/models.transformers.TransformerEncoder
    head = @models.encodecmae.heads.FrameLevelClassificationHead
    optimizer=@torch.optim.AdamW
    lr_scheduler=None
    masked_weight=%MASKED_LOSS_WEIGHT
    quantizer_weights=[0.22407463, 0.1759858 , 0.14499009, 0.12150037, 0.1031566
    n_extra_targets=1
torch.optim.AdamW:
    lr=%PRETRAIN_MAX_LR
    betas=(0.9,0.95)
    weight_decay=0.05
models.encodecmae.targets.EncodecQuantizer:
    n = %NUM_ENCODEC_TARGETS
models.encodecmae.masking.TimeGapMask:
    mask_amount = %MASK_AMOUNT
    gap_size = %MASK_GAP_SIZE
    mask_prop = %MASK_PROP
encoder/models.transformers.TransformerEncoder:
    model_dim=%MODEL_DIM
```

Inferencia

```python
from huggingface_hub import hf_hub_download
from .tasks.models import EncodecMAE
from ginpipe.core import gin_configure_externals
import gin
import torch

models = ['base', 'small', 'base-st', 'large', 'large-st']
models = {k: 'lpepino/encodecmae-{}'.format(k) for k in models}

@gin.configurable
def get_model(model):
    return model

def load_model(model,mode='eval',device='cuda:0'):
    #Get model files
    config_str = gin.config_str()
    gin.clear_config()
    ckpt_file = hf_hub_download(repo_id=models[model],filename='model.pt')
    config_file = hf_hub_download(repo_id=models[model],filename='config.gin')
    import_file = hf_hub_download(repo_id=models[model],filename='imports')
    flag = {'module_list': [import_file]}
    gin_configure_externals(flag)
    gin.parse_config_file(config_file)
    model = get_model()()
    ckpt = torch.load(ckpt_file, map_location='cpu')
    model.load_state_dict(ckpt['state_dict'])
    gin.clear_config()
    gin.parse_config(config_str)
    if mode=='eval':
        model.eval()
    model.to(device)
    return model
```

Puedo levantar cualquier modelo en HFHub que siga ese formato, siempre y cuando tenga instaladas las librerías que aparecen en el imports.

Entrenamiento en Docker

```
from nvidia/cuda:11.7.1-cudnn8-devel-ubuntu20.04

ENV TZ=America/Argentina/Buenos_Aires
RUN ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && echo $TZ > /etc/timezone

COPY requirements.txt .
RUN apt-get --fix-missing update && apt-get update && apt-get install -y \
    python3.9 \
    python3-pip \
    git \
    sox
RUN pip3 install -r requirements.txt
```

Armo un dockerfile (que pasos debería ejecutar en una máquina pelada para que funcione el código. Ejemplo, instalar python, sox, librerías.

```
version: "3.9"
services:
  encodecmae:
    image: encodecmae
    container_name: encodecmae-train
    volumes:
      - /home/lpepino/encodecmae:/workspace/encodecmae
      - /mnt/ssd4T/datasets:/workspace/datasets
    ipc: host
    stdin_open: true
    tty: true
    deploy:
      resources:
        reservations:
          devices:
          - driver: nvidia
            device_ids: ['0','1']
            capabilities: [gpu]
```

Y un docker-compose.yml. Le da un nombre al container, le da acceso a las gpus y a las carpetas donde tengo el código y dataset.

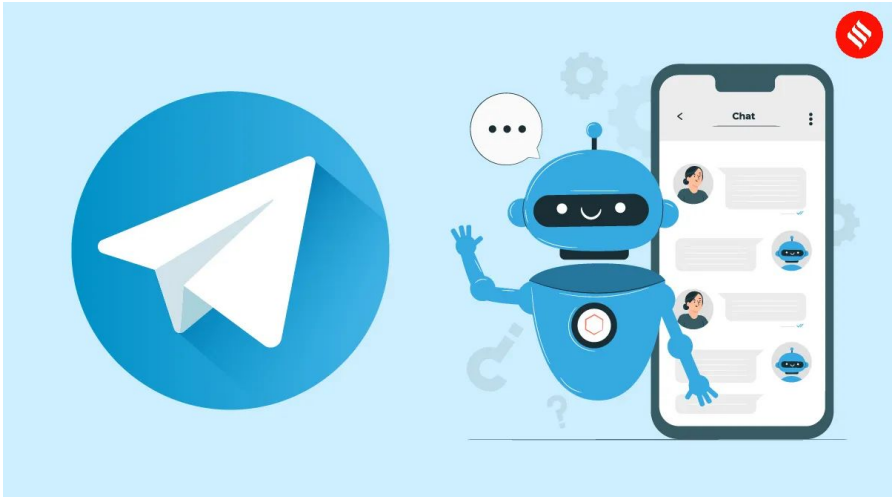
Entrenamiento en Docker

```
1    cd encodecmae
2    docker build -t encodecmae:latest .
3    docker compose up -d
4    docker attach encodecmae-train
```

En muy pocos pasos cualquiera puede setear el entorno necesario para entrenar un modelo que libere.

Demos

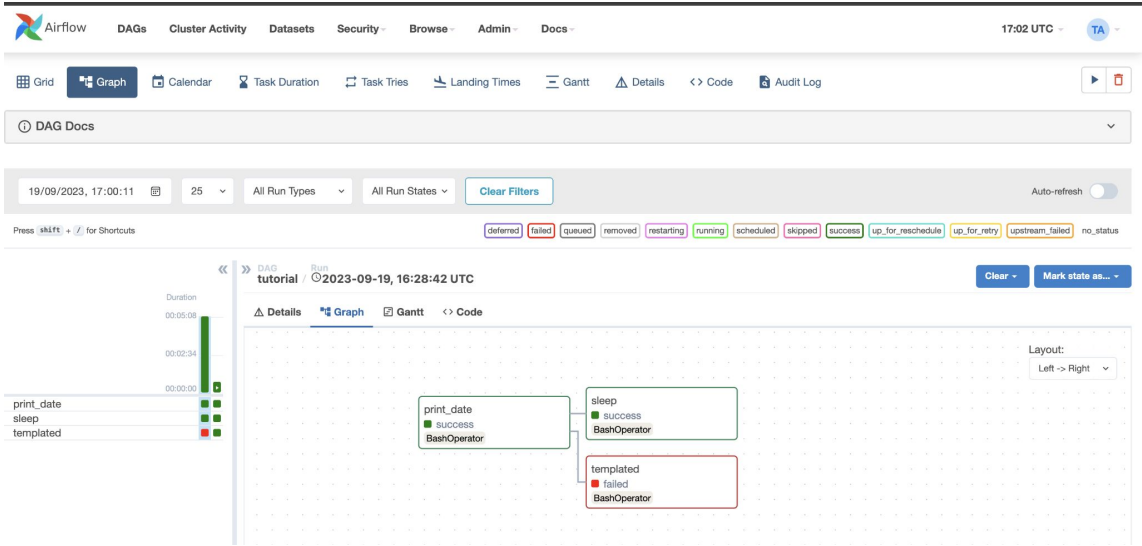
Gradio
Streamlit



Telegram Bots

Workflow Orchestrators

- Dashboards / Observabilidad
- Scheduling de tareas
- Tareas como DAG
- Fault tolerance
- Caching
- Paralelización



Documentar es Gratis

```python
class DataFrameDataset(Dataset):
    """
    Dataset class for handling data stored in Pandas DataFrame.

    Args:
        metadata (pandas.DataFrame): DataFrame containing the dataset.
        out_cols (list): List of column names in the DataFrame to be used as output.
        preprocessors (list, optional): List of preprocessor functions to apply on each row of the DataFrame. Defaults to None.
    """
    def __init__(self, metadata: pd.DataFrame, out_cols: List[str], preprocessors=Optional[List[Callable]] = None):
        self._metadata = metadata
        self._out_cols = out_cols
        self._preprocessors = [p() for p in preprocessors]

    def __getitem__(self, idx: int) -> Dict[str, any]:
        """
        Retrieves an item from the dataset.

        Args:
            idx (int): Index of the item to retrieve.

        Returns:
            dict: Dictionary containing data for the specified index.
        """
        row = copy.deepcopy(self._metadata.iloc[idx])
        for p in self._preprocessors:
            row = p(row)
        out = {k: row[k] for k in self._out_cols}
        return out

    def __len__(self):
        return len(self._metadata)
```

Para mejorar de GinPipe

- Ginception.
- Cacheo entre experimentos. Posibilidad de especificar path a un estado en otra carpeta y de ejecutar a partir de una tarea.
- Ejecución tipo DAG. Tiene sentido si se agrega paralelización de tareas.
- Opcion de ejecutar desde Python. Ej: hacer un grid search sin tener que codearlo desde bash.
- Expresiones en config (ej: poder hacer sumas).
- Detectar y alertar cambios de código/config.