Automated Machine Learning

Diego Andrade Canosa

Roberto López Castro

Índice

- Introducción
- Ray-Tune
- AutoPytorch
- Otras herramientas

Introducción

- Automated ML es un proceso que involucra la automatización de varias tareas de ML que son especialmente costosas
- Involucra varias tareas relacionadas
  - Ajuste de hiperparámetros: batch, learning rate, etc…
  - Ajuste paramétrico de la arquitectura del modelo
    - Network Architecture Search (NAS)
- Dominios:
  - NLP
  - Visión
  - Clasificación
  - Regresión
  - Series temporales

Fuente: What is automated ML? AutoML - Azure Machine Learning | Microsoft Learn

Soluciones AutomatedML

- Herramientas código abierto
  - Ludwig: ludwig-ai/ludwig
  - AutoKeras: AutoKeras
  - AutoPytorch: automl/Auto-PyTorch
  - Auto-Sklearn: auto-sklearn
  - Ray Tune: Tune: Scalable Hyperparameter Tuning
  - Mllib: MLlib | Apache Spark
  - Hyperopt: Hyperopt Documentation

Fuente: windmaple/awesome-AutoML

Soluciones AutomatedML

- Herramientas cloud y comerciales
  - Google Cloud AutoML: Cloud AutoML
    - Usa el ajuste de hiperparámetros | AI Platform Training | Google Cloud
  - Abacus AI: Abacus.AI
  - Amazon SageMaker Autopilot: AutoML
  - Azure Machine Learning Studio: Azure Machine Learning
  - NASLib: automl/NASLib

Fuente: windmaple/awesome-AutoML

Referencias AutoML

- Libro Open Access AutoML_Book.pdf

HPO: Hyperparameters Optimization

- Es una disciplina que consiste en hacer varios entrenamientos sucesivos hasta dar con la combinación de hiperparámetros que genera la mejor solución
  - Learning Rate
  - Tamaño de Batch
  - Número de Epochs
  - En algunos modelos:
    - Número de capas ocultas
    - Número de capas total

HPO: Hyperparameters Optimization

- La función que tratamos de minimizar normalmente es la precisión de la red resultante (no siempre)
- Estrategias de exploración del espacio de búsqueda:
  - Búsqueda en anchura (grid)
  - Métodos de optimización Bayesiana
  - Búsqueda aleatoria
  - Búsqueda genética
  - **Búsqueda manual**
- Posible optimizaciones
  - Acotar el espacio de búsqueda inicial con un entrenamiento inicial de pocas epochs (*early stopping*)
    - Solo las combinaciones de hiperparámetros más aptas sobreviven a las sucesivas etapas
    - También se puede entrenar todas las epochs pero con menos datos
  - Podar el espacio de búsqueda inicial mediante heurísticas, manteniendo solo las mejores combinaciones

Model selection

- Consiste en probar varios modelos diferentes, o varias variantes del mismo modelo diferentes hasta dar con la más apta

Network Architecture Search (NAS)

- Son métodos que tratan de buscar la red más adecuada para resolver un problema contrayéndola de forma iterativa
- Es necesario definir:
  - Un espacio de búsqueda de candidatos
  - Una estrategia de búsqueda
  - Una métrica para medir el rendimiento de cada punto del espacio de búsqueda

NAS

- Estrategias de búsqueda:
  - Reinforcement Learning (RL): La evolución de la red está guiada por un algoritmo de RL
  - Algoritmos genéticos: Partimos de un conjunto de candidatos generados de forma aleatoria, luego se generan nuevos individuos por mutación y combinación de los más aptos de cada generación
  - Optimización bayesiana: Se utiliza una función para evaluar cada red, y se busca la siguiente variante en base a un proceso de optimización bayesiano
  - Hill-climbing: La red se va construyendo de forma incremental, añadiéndole más complejidad en cada paso
  - Búsqueda multiobjetivo: La búsqueda de la red no tiene que estar guiada solo por la precisión de la misma, sino que pueden tenerse en cuenta de forma simultánea otros objetivos como el tamaño de la red, la memoria que ocupa, el tiempo de inferencia
  - **Pruning de redes**: Entrenamos una red muy grande para resolver el problema, y posteriormente vamos quitando partes de la misma intentando mantener la precisión

RAY

- RAY (https://www.ray.io/) es un *framework* unificado que permite escalar fácilmente cargas de trabajo de
  - IA
  - Python
- Versión actual 2.5.1
  - Versión 2.0.0 (agosto 2022)
  - Versión 1.0.0 (septiembre 2020)
  - Versión 0.1.0 (mayo 2017)
- Docs: https://docs.ray.io/en/latest/index.html
- Github educational materials: https://github.com/ray-project/ray-educational-materials

RAY

- Se integra con los frameworks de ML más populares
  - Pytorch
  - TensorFlow
    - Keras
  - Scikit-Learn
  - XGBoost
  - ...

RAY: Motivación

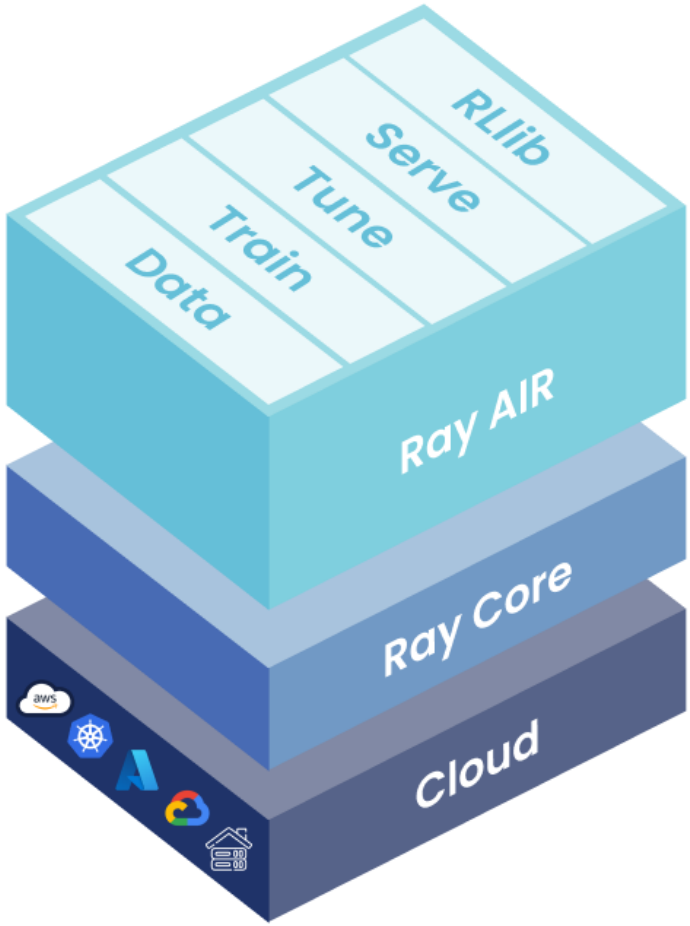
- Proporcionado también como SaaS: https://www.anyscale.com
- Ray Summit: https://raysummit.anyscale.com/speakers
- Patrocinadores y usuarios: IBM, AWS, OpenAI, Linkedin, Pinterest, Google, Netflix

Ray: Usos posibles

- Inferencia
- Entrenamiento distribuido
- Servidor de modelos
- **Ajuste distribuido de hiperparámetros**
- Creación de una plataforma de ML

RAY

- Componentes principales:
  - Ray AIR (librerías de alto nivel para escalado de cargas de ML)
    - Datasets: Preprocesado de datos distribuido
    - Train: Entrenamiento distribuido
    - Tune: Ajuste de hiperparámetros distribuido
    - Rlib: *Reinforcement Learning* escalable
    - Serve: *Serving* escalable y programable. Se usa para construir APIs de inferencia online.
  - Ray Core (framework de computación distribuida de bajo nivel)
    - Tasks: Funciones sin estado ejecutadas en un clúster
    - Actors: Procesos trabajadores sin estado creados en un clúster
    - Objects: Valores inmutables accesibles a través de un clúster
  - Ray cluster: Abstrae la capa de escalado de modelo que se corresponde con varios trabajadores (nodos-dispositivos)



high-level libraries which enable simple scaling of AI workloads

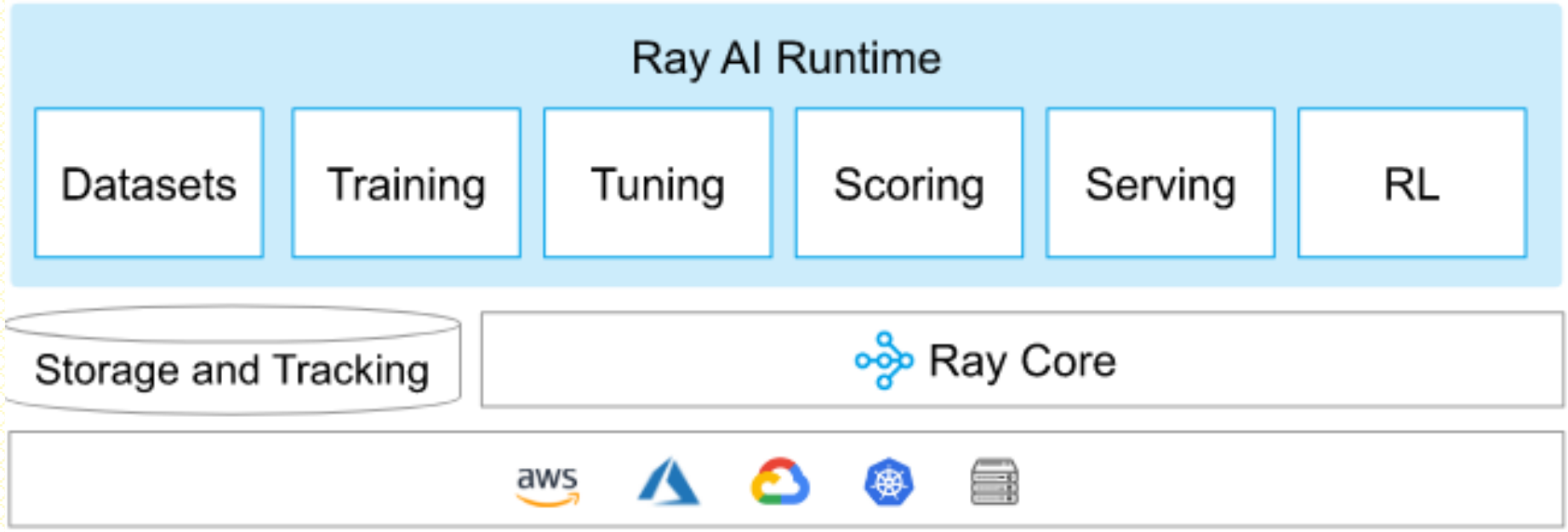
a low-level distributed computing framework with a concise core, Python-first API

citic

RAY: Escalando entrenamientos de IA

Código:

```
import ray.train as train
from ray.train import Trainer
import torch

(...)

def train_func():
   (…)

trainer = Trainer(backend="torch", num_workers=4)
trainer.start()
results = trainer.run(train_func)
trainer.shutdown()
```

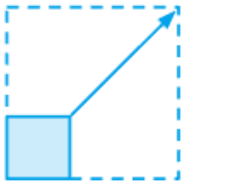

Ray AI Runtime
Datasets
Training
Tuning
Scoring
Serving
RL
Storage and Tracking
Ray Core
aws

Ray AIR

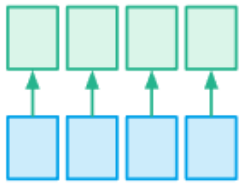
Seamless scaling: El código escala de forma simple

Unified ML API: Su API hace sencillo cambiar de un framework de ML a otro

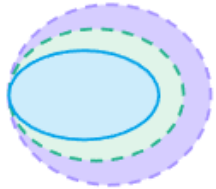
Open and Extensible: Es código abierto y permite integrar componentes propios



Seamless Scaling

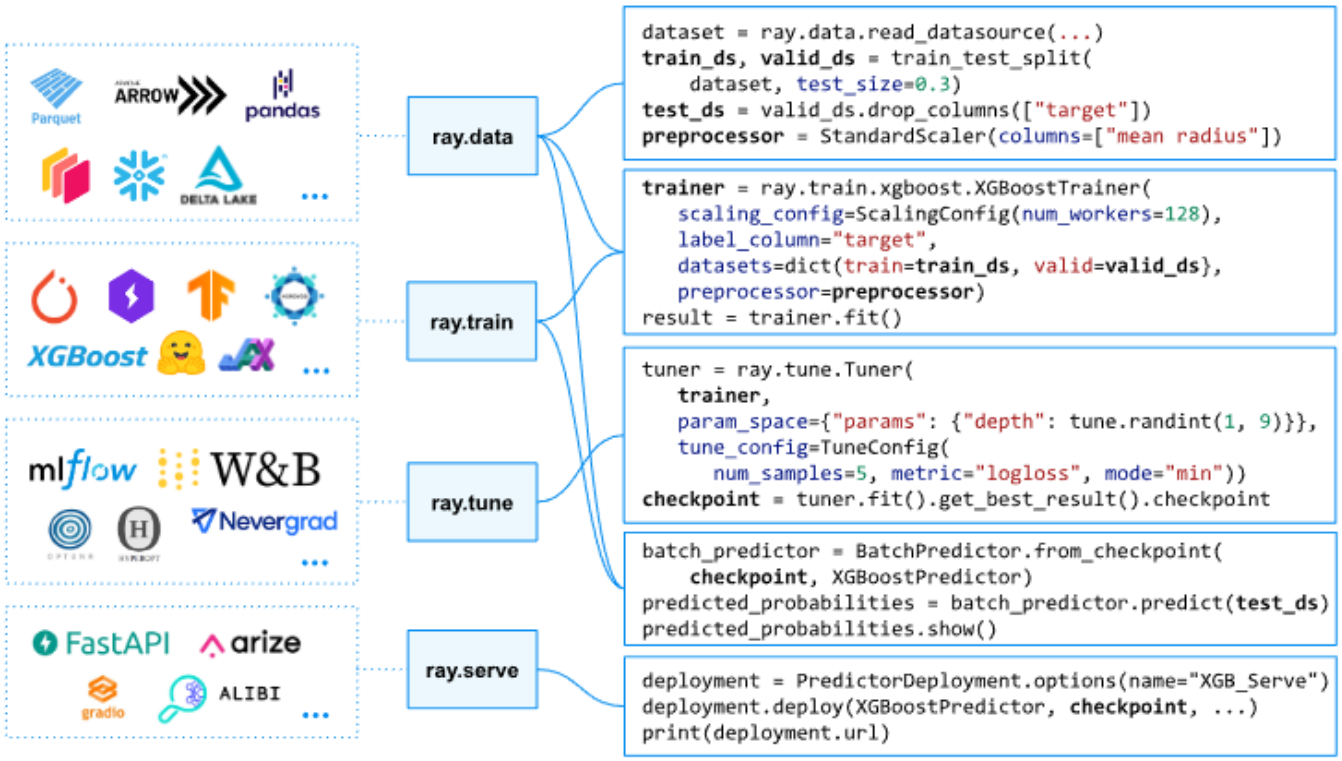


Unified ML API



Open and Extensible

RAY AIR: Inicio rápido



Fuente: [Ray AI Runtime (AIR) — Ray 2.1.0]

citic

Ray Datasets

- Es el método estándar para cargar e intercambiar datos en Ray AIR
- Se usan en:
  - Carga de datos
  - Preprocesado
  - Infererencia batch

Fuente: Key Concepts — Ray 2.1.0

Ray Preprocessing

- Se usan para transformar datos de entrada en características (*features*)
- AIR proporciona una colección de *preprocessors* predefinidos

```python
import numpy as np

from ray.data.preprocessors import Concatenator, Chain, StandardScaler

# Create a preprocessor to scale some columns and concatenate the result.
preprocessor = Chain(
    StandardScaler(columns=["mean radius", "mean texture"]),
    Concatenator(exclude=["target"], dtype=np.float32),
)
```

Ray Preprocessing

- La clase *Preprocessor* tiene 4 métodos públicos que pueden ser usados por un *trainer*
  - *fit():* Calcula información de estado de un *dataset* (ej. La media o la deviación estándar de una columna). Esta información se usa para ejecutar una *transform()*
    - Se llama durante el entrenamiento
  - *transform():* Aplica una transformación sobre un *Dataset*
    - Se llama durante el entrenamiento, validación o inferencia offline
  - *transform_batch()*: Aplica una transformación sobre un solo *batch*
    - Se llama durante la inferencia online u offline
  - *fit_transform():* Se usa para llamar en secuencia *fit()* y *transform()*

Ray Preprocessing

- *Preprocessors* existentes en Ray
  - Genéricos (ej. BatchMapper)
  - Categorical encoders (ej. OneHotEncoder)
  - Feature scalers (ej. MaxAbsScaler)
  - Text encoders (ej. Tokenizer)

Ray Train

- Ray Train es una librería de Ray que permite realizar entrenamientos en entornos distribuidos
  - Todavía en Beta
- Soporte para muchas tecnologías
  - DL: Pytorch, Tensorflow, Horovod
  - Tree-based: Xgboost, LightGBM
  - Otros: HugginFace, Scikit-Learn, Rlib
- Mecanismos incorporados
  - Callbacks para *early-stopping*
  - Checkpointing
  - Integración con TensorBoard, Weight/Biases, y Mlflow
  - Jupyter
- Integración con el ecosistema Ray: Data, Tune y Cluster

https://docs.ray.io/en/latest/train/train.html

1 Creación del dataset usando el API de Data

3 Configuración del trainer

2 Configuración de los recursos a utilizar

4 Lanzamiento del entrenamiento

```python
def train_loop_per_worker():
    dataset_shard = session.get_dataset_shard("train")
    model = NeuralNetwork()
    loss_fn = nn.MSELoss()
    optimizer = torch.optim.SGD(model.parameters(), lr=0.1)

    model = train.torch.prepare_model(model)

    for epoch in range(num_epochs):
        for batches in dataset_shard.iter_torch_batches(
            batch_size=32, dtypes=torch.float
        ):
            inputs, labels = torch.unsqueeze(batches["x"], 1), batches["y"]
            output = model(inputs)
            loss = loss_fn(output, labels)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            print(f"epoch: {epoch}, loss: {loss.item()}")

        session.report(
            {},
            checkpoint=Checkpoint.from_dict(
                dict(epoch=epoch, model=model.state_dict())
            ),
        )


train_dataset = ray.data.from_items([{"x": x, "y": 2 * x + 1} for x in range(200)])
scaling_config = ScalingConfig(num_workers=3, use_gpu=use_gpu)
trainer = TorchTrainer(
    train_loop_per_worker=train_loop_per_worker,
    scaling_config=scaling_config,
    datasets={"train": train_dataset},
)
result = trainer.fit()
```

citic

```python
def train_loop_per_worker():
    dataset_shard = session.get_dataset_shard("train")
    model = NeuralNetwork()
    loss_fn = nn.MSELoss()
    optimizer = torch.optim.SGD(model.parameters(), lr=0.1)

    model = train.torch.prepare_model(model)

    for epoch in range(num_epochs):
        for batches in dataset_shard.iter_torch_batches(
            batch_size=32, dtypes=torch.float
        ):
            inputs, labels = torch.unsqueeze(batches["x"], 1), batches["y"]
            output = model(inputs)
            loss = loss_fn(output, labels)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            print(f"epoch: {epoch}, loss: {loss.item()}")

        session.report(
            {},
            checkpoint=Checkpoint.from_dict(
                dict(epoch=epoch, model=model.state_dict())
            ),
        )


train_dataset = ray.data.from_items([{"x": x, "y": 2 * x + 1} for x in range(200)])
scaling_config = ScalingConfig(num_workers=3, use_gpu=use_gpu)
trainer = TorchTrainer(
    train_loop_per_worker=train_loop_per_worker,
    scaling_config=scaling_config,
    datasets={"train": train_dataset},
)
result = trainer.fit()
```

5 Shard del dataset

6 Modelo+loss+ optimizer

7 Preparación del modelo

8 Bucle de entrenamiento

Rain Train: Trainers, Checkpoint y Predictors

Trainer Class	Checkpoint Class	Predictor Class
`TorchTrainer`	`TorchCheckpoint`	`TorchPredictor`
`TensorflowTrainer`	`TensorflowCheckpoint`	`TensorflowPredictor`
`HorovodTrainer`	(Torch/TF Checkpoint)	(Torch/TF Predictor)
`XGBoostTrainer`	`XGBoostCheckpoint`	`XGBoostPredictor`
`LightGBMTrainer`	`LightGBMCheckpoint`	`LightGBMPredictor`
`SklearnTrainer`	`SklearnCheckpoint`	`SklearnPredictor`
`TransformersTrainer`	`TransformersCheckpoint`	`TransformersPredictor`
`RLTrainer`	`RLCheckpoint`	`RLPredictor`

Portar código Pytorch a Ray Train

- Existen guías paso a paso para portar código en Pytorch, TensorFlow u Horovod a Ray Train

https://docs.ray.io/en/latest/train/dl_guide.html#train-porting-code

Portar código Pytorch a Ray Train

Usa la función *prepare_model()* para mover los datos del modelo al dispositivo correcto

```
 import torch
 from torch.nn.parallel import DistributedDataParallel
+from ray.air import session
+from ray import train
+import ray.train.torch


 def train_func():
-    device = torch.device(f"cuda:{session.get_local_rank()}" if
-        torch.cuda.is_available() else "cpu")
-    torch.cuda.set_device(device)

     # Create model.
     model = NeuralNetwork()

-    model = model.to(device)
-    model = DistributedDataParallel(model,
-        device_ids=[session.get_local_rank()] if torch.cuda.is_available() else
None)

+    model = train.torch.prepare_model(model)

     ...
```

Portar código Pytorch a Ray Train

Usa la función *prepare_data_loader()* para añadir un *DistributedSampler* a tu *DataLoader*

```
 import torch
 from torch.utils.data import DataLoader, DistributedSampler
+from ray.air import session
+from ray import train
+import ray.train.torch


 def train_func():
-    device = torch.device(f"cuda:{session.get_local_rank()}" if
-        torch.cuda.is_available() else "cpu")
-    torch.cuda.set_device(device)

     ...

-    data_loader = DataLoader(my_dataset, batch_size=worker_batch_size,
sampler=DistributedSampler(dataset))

+    data_loader = DataLoader(my_dataset, batch_size=worker_batch_size)
+    data_loader = train.torch.prepare_data_loader(data_loader)

     for X, y in data_loader:
-        X = X.to_device(device)
-        y = y.to_device(device)
```

Portar código Pytorch a Ray Train

Crea un Ray Train Trainer con un configuración específica y ejecuta el entrenamiento

```python
from ray.air import session, ScalingConfig
from ray.train.torch import TorchTrainer

def train_func(config):
    for i in range(config["num_epochs"]):
        session.report({"epoch": i})

trainer = TorchTrainer(
    train_func,
    train_loop_config={"num_epochs": 2},
    scaling_config=ScalingConfig(num_workers=2)
)
result = trainer.fit()
print(result.metrics["num_epochs"])
# 1
```

Función de entrenamiento

Creación del trainer

Configuración

Ejecución del entrenamiento

Imprimir resultados

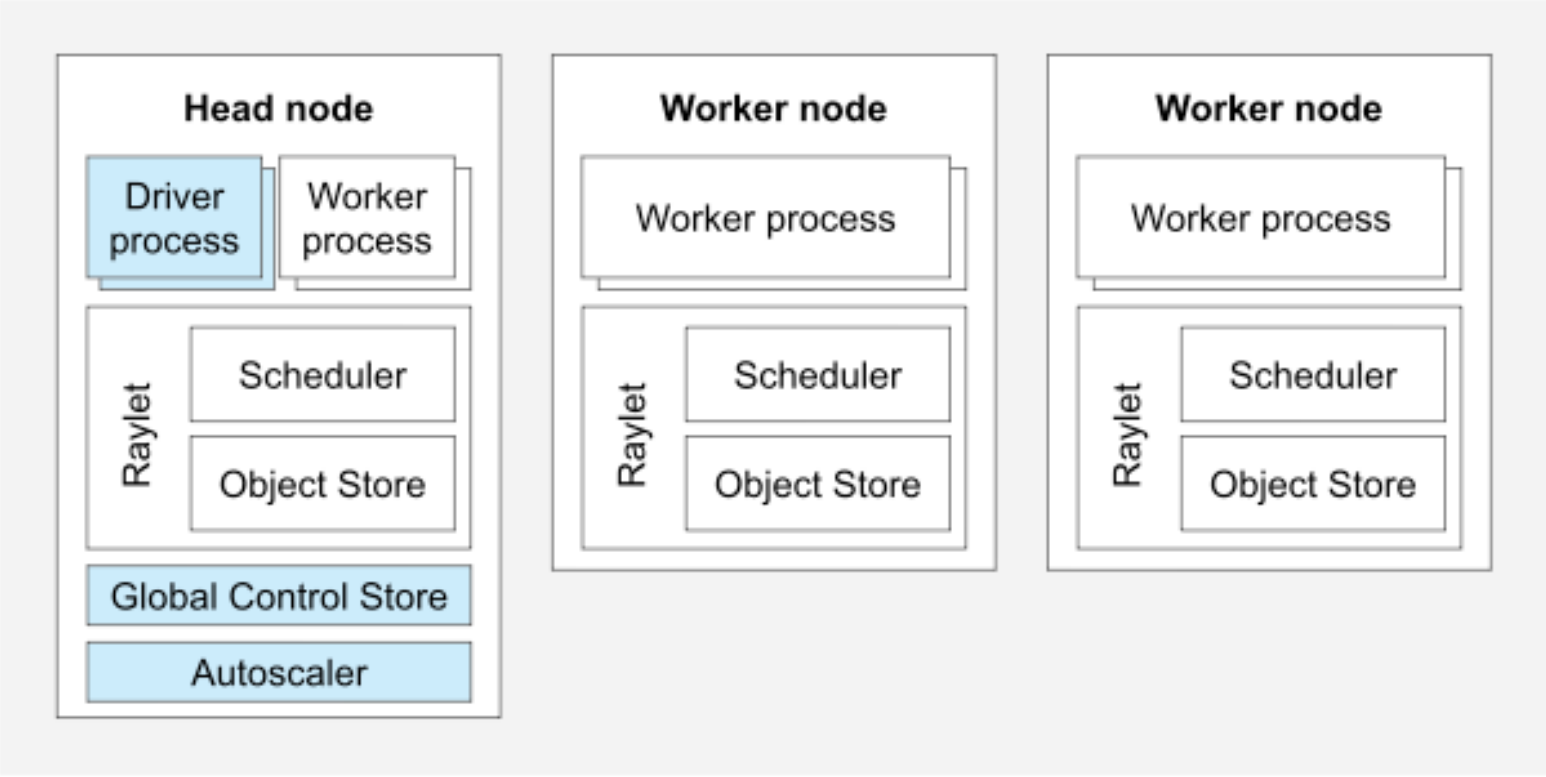
Otros conceptos de Ray

- Checkpoints: Permiten generar checkpoints del entrenamiento del modelo
- Batch predictor: Se pueden utilizar a partir de un checkpoint de un modelo para generar inferencia sobre ese estado del modelo
- Deployments: Sirven para desplegar el modelo como un servicio a través de un servidor web

Ray Cluster

- Un Clúster Ray está formado por un solo *head node* y un número arbitrario de *worker nodes* conectados
  - *Head node*: Se comporta como un *worker node* excepto que también se encarga de las tareas de gestión del clúster
  - Worker node: Forman parte del clúster y pueden participar en cualquier cálculo distribuido que se envíe al clúster
    - El *head node* también actúa como *worker node*
- Ray incorpora un mecanismo de autoescalado que permite aumentar o reducir dinámicamente el número de nodos trabajadores
- Nos podemos conectar a un cluster RAY existente usando *ray.init*

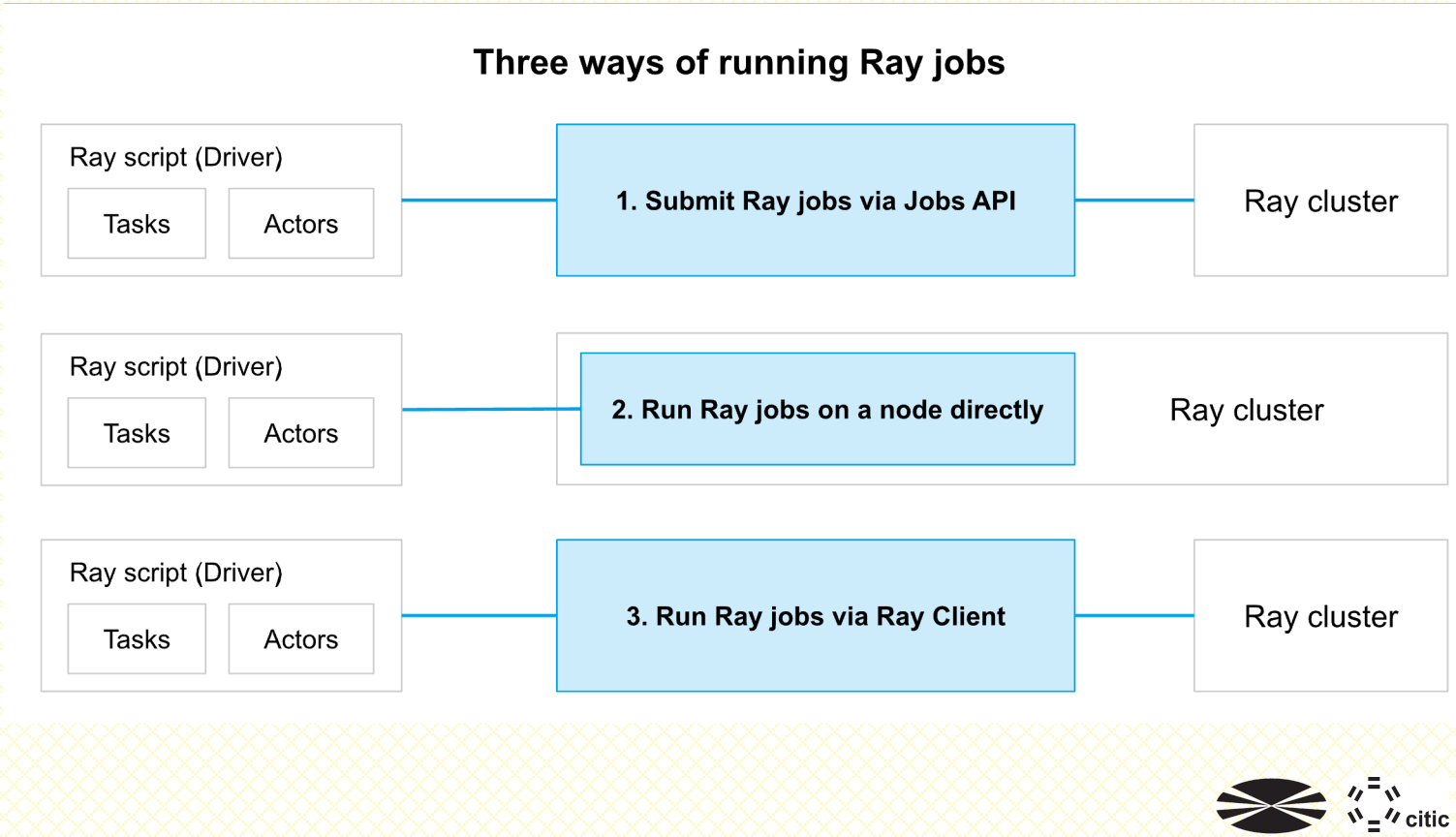
Ray Cluster



Ray Cluster

- Un *ray job* es una aplicación que implica la ejecución de:
  - Una colección de *Ray tasks*, *objects* y *actors* que se originan en el mismo script
- Formas de ejecutar un *ray job*:
  - Usar el API de *ray Jobs*
  - Ejecutar el script del trabajo dentro de cualquier nodo del clúster de Ray
  - Usar *Ray Client* para conectar remotamente al clúster dentro de un script

Ray Cluster



citic

Ray Cluster: Soporte Slurm

- Soporte para Slurm en proceso:
  - Poco intuitivo pero usable
- https://docs.ray.io/en/latest/cluster/vms/user-guides/community/slurm.html
- Para desplegar Ray en Slurm se debe desplegar varias copias del mismo código en varios nodos

Despliegue de Ray en Slurm

- Pasos a seguir:
  - Incluir las cabeceras adecuadas en el script batch
  - Cargar los módulos requeridos del computador
  - Consulta la lista de nodos de computación disponibles en el trabajo y sus direcciones ip
  - Lanza un proceso *head* en uno de los nodos
  - Lanza n-procesos *workers* en cada uno de los dos nodos
    - Conéctalos al nodo head indicando su dirección ip
  - Una vez el cluster está disponible, ya se puede enviar la tarea de usuario

Fuente: Deploying on Slurm — Ray 2.1.0

Despliegue de Ray en Slurm

- Incluir las cabeceras adecuadas en el script batch
  - Son las cabeceras que especifican los recursos computacionales solicitados

```
#SBATCH –N 2
#SBATCH –gres=gpu:a100:2
...
```

Despliegue de Ray Slurm

- Cargar los módulos requeridos del computador
  - Cargar módulos del computador

    module load ...
  - y/o cargar un entorno conda predefinido creado previamente

    module activate myrayenv

Despliegue de Ray Slurm

- Lanza el proceso *head* en uno de los nodos: consiguiendo la ip

```
# Getting the node names
nodes=$(scontrol show hostnames "$SLURM_JOB_NODELIST")
nodes_array=($nodes)

head_node=${nodes_array[0]}
head_node_ip=$(srun --nodes=1 --ntasks=1 -w "$head_node" hostname --ip-address)

# if we detect a space character in the head node IP, we'll
# convert it to an ipv4 address. This step is optional.
if [[ "$head_node_ip" == *" "* ]]; then
IFS=' ' read -ra ADDR <<<"$head_node_ip"
if [[ ${#ADDR[0]} -gt 16 ]]; then
  head_node_ip=${ADDR[1]}
else
  head_node_ip=${ADDR[0]}
fi
echo "IPV6 address detected. We split the IPV4 address as $head_node_ip"
fi
```

Despliegue de Ray Slurm

- Lanza el proceso *head* en uno de los nodos: lanzamiento

```
port=6379
ip_head=$head_node_ip:$port
export ip_head
echo "IP Head: $ip_head"

echo "Starting HEAD at $head_node"
srun --nodes=1 --ntasks=1 -w "$head_node" \
    ray start --head --node-ip-address="$head_node_ip" --port=$port \
    --num-cpus "${SLURM_CPUS_PER_TASK}" --num-gpus "${SLURM_GPUS_PER_TASK}" --block &
```

Despliegue de Ray Slurm

- Lanza los procesos *workers*

```
# optional, though may be useful in certain versions of Ray < 1.0.
sleep 10

# number of nodes other than the head node
worker_num=$((SLURM_JOB_NUM_NODES - 1))

for ((i = 1; i <= worker_num; i++)); do
    node_i=${nodes_array[$i]}
    echo "Starting WORKER $i at $node_i"
    srun --nodes=1 --ntasks=1 -w "$node_i" \
        ray start --address "$ip_head" \
        --num-cpus "${SLURM_CPUS_PER_TASK}" --num-gpus "${SLURM_GPUS_PER_TASK}" --
block &
    sleep 5
done
```

Despliegue de Ray Slurm

- Enviar el script de usuario

```
# ray/doc/source/cluster/doc_code/simple-trainer.py python -u simple-trainer.py "$SLURM_CPUS_PER_TASK"
```

- Existe un script (*slurm_launch.py*) que autogenera los scripts de slurm para un entorno en base a una cierta configuración de parámetros hecha por los usuarios

Ejemplo:

```
python slurm-launch.py --exp-name test --command "python your_file.py" --num-nodes 3 --node NODE_NAMES
```

Ray Dashboard

- Es un panel de control de RAY que permite monitorizar y depurar la ejecución de aplicaciones Ray
- Se inicia como un servicio web (por defecto en el puerto 8265)
  - Single-node: en la IP del equipo
  - Clúster: en la IP del *head node*
- Vistas disponibles:
  - Metrics y Cluster view: Permiten analizar, monitorizar y visualizar el estado y la utilización de recursos lógicos y físicos
  - Jobs view: Monitorizar el estado y el progreso de tareas y trabajos
  - Jobs y Logs view: Mensajes de error de tareas fallidas
  - Metrics y Cluster view: Analizar el uso de CPU y memoria de las diferentes tareas y actores
  - Server view: Monitorizar una aplicación-servidor

Ray Tune: Intro

- Librería del ecosistema Ray para la definición de tareas de Automated ML
  - Centrada en la tarea de Hyper-Parameters Optimization (HPO)

Si no está instalada basta con instalar

```
pip install ray[tune]
```

Ray Tune: Intro

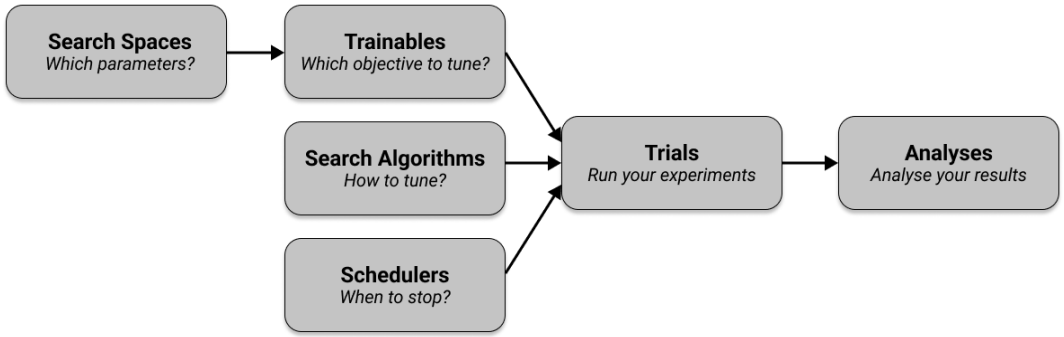
- Frameworks de ML soportados: Pytorch, TensorFlow, Keras, Scikit-Learn, XGBoost,...
- Algoritmos: Population Based Training (PBT) o HyperBand/ASHA
- Integración con otras herramientas de HPO: Ax, BayesOpt, Dragonfly, FLAML, HEBO, Hyperopt, Nevergrad, Optuna,...

Ray Tune: Ventajas

- Proyecto fuerte, bien establecido apoyado por una gran comunidad
- Requiere poco código para ser aplicado a tu proyecto de ML
  - Independientemente del framework de ML que se use
- Añade soporte para entornos distribuidos a gran escala de forma transparente al usuario
- Permite hacer uso de otras herramientas SOTA, añadiéndoles las capacidades propias de Ray Tune

RayTune

- Librería construida alrededor de varias abstracciones:
  - Definición de los hiperparámetros que queremos ajustar
    - Espacio de búsqueda
  - Trainable: Objetivo que hay que ajustar
  - Algoritmo de búsqueda
  - Scheduler: Permite detener algunas búsquedas de forma prematura
  - La definición de todo lo anterior es recibida por un Tuner



Fuente: Key Concepts — Ray 2.2.0

citic

Ray Tune: Trainable

- Un trainable puede ser creado de dos formas:
  - Usando la function API

```python
from ray.air import session


def objective(x, a, b):  # Define an objective function.
    return a * (x**0.5) + b


def trainable(config):  # Pass a "config" dictionary into your trainable.

    for x in range(20):  # "Train" for 20 iterations and compute intermediate scores.
        score = objective(x, config["a"], config["b"])

        session.report({"score": score})  # Send the score to Tune.
```

Ray Tune: Trainable

- Un trainable puede ser creado de dos formas:
  - Usando la class API

```python
from ray import tune


def objective(x, a, b):
    return a * (x**2) + b


class Trainable(tune.Trainable):
    def setup(self, config):
        # config (dict): A dict of hyperparameters
        self.x = 0
        self.a = config["a"]
        self.b = config["b"]

    def step(self):  # This is called iteratively.
        score = objective(self.x, self.a, self.b)
        self.x += 1
        return {"score": score}
```

Ray Tune: Espacios de búsqueda

- Un espacio de búsqueda (*search space*) define los valores a explorar de los hiperparámetros
  - Puede indicar también un método de *sampling* que permita muestrear los valores de cada hiperparámetro (distribución normal/uniforme/...)

Ray Tune: Espacios de búsqueda

```python
config = {
    "uniform": tune.uniform(-5, -1),  # Uniform float between -5 and -1
    "quniform": tune.quniform(3.2, 5.4, 0.2),  # Round to multiples of 0.2
    "loguniform": tune.loguniform(1e-4, 1e-1),  # Uniform float in log space
    "qloguniform": tune.qloguniform(1e-4, 1e-1, 5e-5),  # Round to multiples of
0.00005
    "randn": tune.randn(10, 2),  # Normal distribution with mean 10 and sd 2
    "qrandn": tune.qrandn(10, 2, 0.2),  # Round to multiples of 0.2
    "randint": tune.randint(-9, 15),  # Random integer between -9 and 15
    "qrandint": tune.qrandint(-21, 12, 3),  # Round to multiples of 3 (includes 12)
    "lograndint": tune.lograndint(1, 10),  # Random integer in log space
    "qlograndint": tune.qlograndint(1, 10, 2),  # Round to multiples of 2
    "choice": tune.choice(["a", "b", "c"]),  # Choose one of these options uniformly
    "func": tune.sample_from(
        lambda spec: spec.config.uniform * 0.01
    ),  # Depends on other value
    "grid": tune.grid_search([32, 64, 128]),  # Search over all these values
}
```

Ray Tune: Trials

- Los experimentos (***trials***) se generan a través de la ejecución **Tuner.fit** que recibe dos parámetros
  - Un *trainable*
  - Un diccionario indicando la configuración del espacio de búsqueda
- La función *fit* generará un conjunto de objetos de tipo *Trial* que contienen información sobre cada ejecución hecha con una combinación de valores de hiperparámetros distinta
  - Cada ejecución, que está asociada a un conjunto de valores de hiperparámetros distinto, se conoce con el nombre de ***sample***
- Ray Tune determina automáticamente cuál es la forma más eficiente de ejecutar todos los trials previsto en función de los recursos disponibles en el Ray Cluster
- Se puede especificar:
  - El número de trials distintos a ejecutar (*num_samples*)
  - Cuánto tiempo hay disponible para ejecutar el mayor número de *trials* posible (*time_budget_s*)

Tune Search Algorithms

- El algoritmo de búsqueda (*search algorithm*) define la estrategia usada para explorar el espacio de búsqueda y generar el mejor conjunto de samples posible, así como las configuraciones de hiperparámetros asociada a cada uno
- El algoritmo por defecto es generar *samples* con combinaciones aleatorias de hiperparámetros
- Tune dispone de un amplio abanico de algoritmos de búsqueda que se integran con librerías populares tales como Nevergrad, Hyperopt u Optuna
  - Tune hace de pasarela entre la especificación del espacio de búsqueda hecha en Tune y la especificación requerida por cada una de estas librerías

https://docs.ray.io/en/latest/tune/api/suggestion.html#tune-search-alg

Ray Tune

Algoritmos de búsqueda soportados

Fuente: Key Concepts — Ray 2.2.0

SearchAlgorithm	Summary	Website	Code Example
Random search/grid search	Random search/grid search		tune_basic_example
AxSearch	Bayesian/Bandit Optimization	[Ax]	AX Example
BlendSearch	Blended Search	[Bs]	Blendsearch Example
CFO	Cost-Frugal hyperparameter Optimization	[Cfo]	CFO Example
DragonflySearch	Scalable Bayesian Optimization	[Dragonfly]	Dragonfly Example
SkoptSearch	Bayesian Optimization	[Scikit-Optimize]	SkOpt Example
HyperOptSearch	Tree-Parzen Estimators	[HyperOpt]	Running Tune experiments with HyperOpt
BayesOptSearch	Bayesian Optimization	[BayesianOptimization]	BayesOpt Example
TuneBOHB	Bayesian Opt/HyperBand	[BOHB]	BOHB Example
NevergradSearch	Gradient-free Optimization	[Nevergrad]	Nevergrad Example
OptunaSearch	Optuna search algorithms	[Optuna]	Running Tune experiments with Optuna
ZOOptSearch	Zeroth-order Optimization	[ZOOpt]	ZOOpt Example
SigOptSearch	Closed source	[SigOpt]	SigOpt Example
HEBOSearch	Heteroscedastic Evolutionary Bayesian Optimization	[HEBO]	HEBO Example

Ray Tune

- Existe un API en Tune para implementar nuestros propios algoritmos de optimización
- Otras características asociadas a los algoritmos de búsqueda
  - Evaluaciones repetidas (*tune.search.Repeated*): permite la ejecución de cada configuración (o *sample*) con varias semillas aleatorias
  - Limitador de concurrencia (*ConcurrencyLimiter*): limita el número de *trials* concurrentes ejecutables en el proceso de optimización
  - *Shim instantiation* (*tune.create_searcher*): Permite la creación de un algoritmo de búsqueda a partir de un string

Ray Tune: Schedulers

- Los planificadores (*trial schedulers*) mejoran la eficiencia del proceso de búsqueda decidiendo en qué orden, y con qué grado de concurrencia, se ejecutarán los *trials* en el *Ray Cluster*
- El *scheduler* por defecto es una cola FIFO que recorre el conjunto de configuraciones de hiperparámetros posibles en el orden en el que fueron generadas y ejecutando cada una hasta el final (sin early stopping)
- *Schedulers* más sofisticados permiten parar, pausar y cambiar los valores de los hiperparámetros de un *trial* en tiempo dejecución
  - A menudo, incorporan mecanismos de *early stopping*
- Los *schedulers* se configuran para minimizar (o maximizar) una métrica configurable
- Pueden tener interacciones (o incluso incompatibilidades) con los algoritmos de búsqueda

Ray Tune

Schedulers soportados

Scheduler	Need Checkpointing?	SearchAlg Compatible?	Example
ASHA	No	Yes	Link
Median Stopping Rule	No	Yes	Link
HyperBand	Yes	Yes	Link
BOHB	Yes	Only TuneBOHB	Link
Population Based Training	Yes	Not Compatible	Link
Population Based Bandits	Yes	Not Compatible	Basic Example, PPO example

Fuente: Key Concepts — Ray 2.2.0

Ray Tune: Ejemplo rápido

- Minimización de la función $f(x) = a^2 + b$

```python
from ray import tune


def objective(config):  # ①
    score = config["a"] ** 2 + config["b"]
    return {"score": score}


search_space = {  # ②
    "a": tune.grid_search([0.001, 0.01, 0.1, 1.0]),
    "b": tune.choice([1, 2, 3]),
}

tuner = tune.Tuner(objective, param_space=search_space)  # ③

results = tuner.fit()
print(results.get_best_result(metric="score", mode="min").config)
```

Definición función objetivo

Definición espacio de búsqueda

Inicio de tune

citic

Ray Tune: Ejemplo pytorch

```python
import torch
from ray import tune, air
from ray.air import session
from ray.tune.search.optuna import OptunaSearch


def objective(config):  # ①
    train_loader, test_loader = load_data()  # Load some data
    model = ConvNet().to("cpu")  # Create a PyTorch conv net
    optimizer = torch.optim.SGD(  # Tune the optimizer
        model.parameters(), lr=config["lr"], momentum=config["momentum"]
    )

    while True:
        train(model, optimizer, train_loader)  # Train the model
        acc = test(model, test_loader)  # Compute test accuracy
        session.report({"mean_accuracy": acc})  # Report to Tune
```

Ray Tune: Ejemplo pytorch

```
search_space = {"lr": tune.loguniform(1e-4, 1e-2), "momentum": tune.uniform(0.1,
0.9)}
algo = OptunaSearch()  # ②

tuner = tune.Tuner(  # ③
    objective,
    tune_config=tune.TuneConfig(
        metric="mean_accuracy",
        mode="max",
        search_alg=algo,
    ),
    run_config=air.RunConfig(
        stop={"training_iteration": 5},
    ),
    param_space=search_space,
)
results = tuner.fit()
print("Best config is:", results.get_best_result().config)
```

Espacio de búsqueda

Algoritmo de búsqueda

Definición del tuner

Inicio del tuner

Ray Tune: Características adicionales

- Logging: por defecto, Tune almacena registros de todas sus ejecuciones en formatos TensorBoard, CSV y JSON
  - Permite también personalizar la información generada a muy bajo nivel

https://docs.ray.io/en/latest/tune/tutorials/tune-output.html

- Checkpointing: Se pueden almacenar puntos intermedios (*checkpoints*) de la ejecución de Tune (*session.get_checkpoint*)
  - A través de este mecanismo, también podemos implementar un mecanismo de tolerancia a fallos, por el cual la ejecución continúa ante un fallo en un punto intermedio

https://docs.ray.io/en/latest/tune/tutorials/tune-trial-checkpoints.html
https://docs.ray.io/en/latest/tune/tutorials/tune-fault-tolerance.html

Ray Dashboard

compute –c 32

source $STORE/conda/envs/mytorchdist/bin/activate

hostname –i #Copia la ip del nodo

ipython

```
>> import ray
>> ray.init(num_cpus=32,dashboard_host="node_ip",dashboard_port=65353)
```

Abre en el navegador la dirección http://*node_ip:65353*

Cierra ray

```
>> ray.shutdown()
```

Ray y Ray Tune

Sigue el README
https://github.com/diegoandradecanosa/Cesga2023Courses/tree/main/pytorch_dist/ray/000

Ray Train

Sigue el README:
https://github.com/diegoandradecanosa/Cesga2023Courses/tree/main/pytorch_dist/ray/001

Más RAY

- Ray Tune examples: https://docs.ray.io/en/latest/tune/examples/index.html#tune-general-examples

Otras herramientas AutoML

Ejemplos de herramientas: Ludwig

- Ludwig AutoML usa
  - Un conjunto de datos
  - Una columna objetivo
  - Un presupuesto (*Budget)* temporal
- Devuelve un modelo entrenado

- https://ludwig.ai/latest/

Entrada: fichero config yaml

```yaml
input_features:
    - name: data_column_1
      type: number
    - name: data_column_2
      type: category
    - name: data_column_3
      type: text
    - name: data_column_4
      type: image
    ...

output_features:
    - name: data_column_5
      type: number
    - name: data_column_6
      type: category
    ...
```

Command-Line Interface

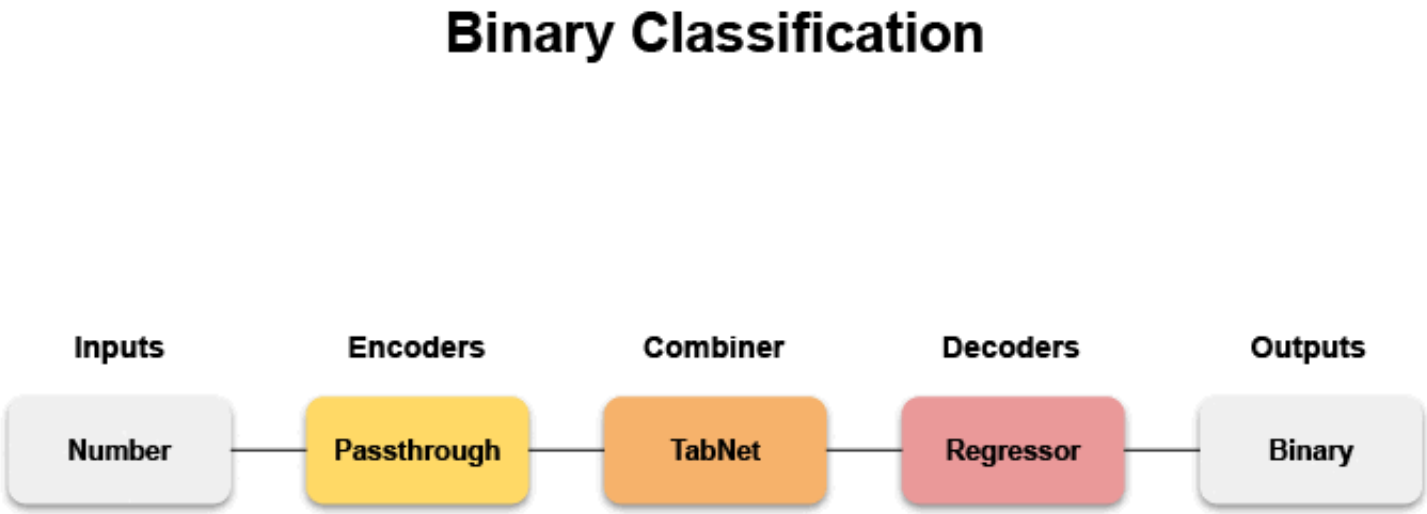
```
ludwig train --config config.yaml --dataset data.csv
ludwig predict --model_path results/experiment_run/model --dataset test.csv
ludwig eval --model_path results/experiment_run/model --dataset test.csv
```

Otras características

- También proporciona un API Python para poder invocar a Ludwig desde scripts Python
- Soporte para entornos distribuidos (multi-nodo, multi-gpu)
- Posibilidad de crear servidores de inferencia
- Ejecutar procesos de HPO nativamente (en Ludwig) o en combinación con Ray Tune

ludwig_config.yaml

```yaml
input_features:
- name: number_feature_name
  type: number

output_features:
- name: binary_feature_name
  type: binary

combiner:
  type: tabnet
```



Ejemplos de herramientas: Ludwig

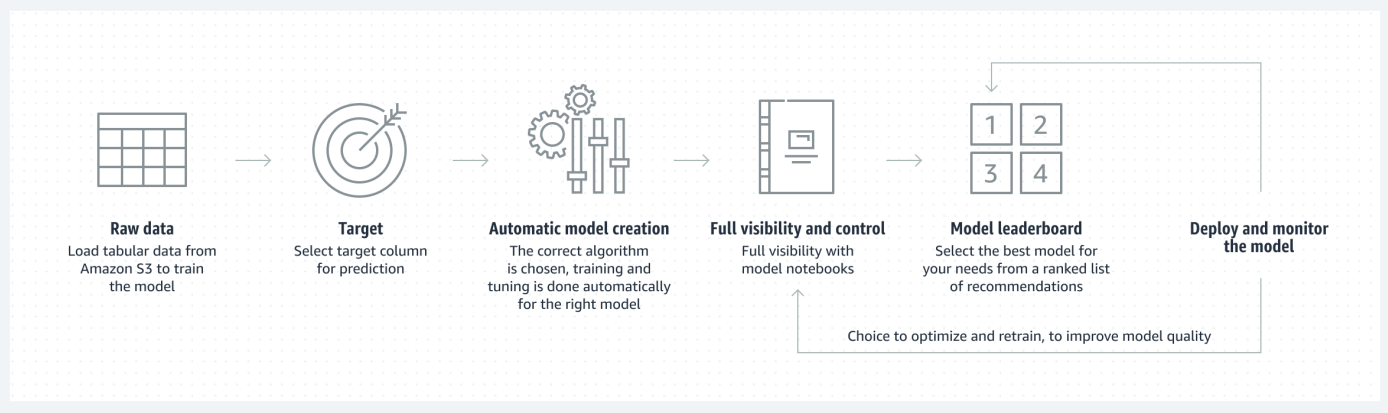
```
import logging
import pprint

from ludwig.automl import auto_train
from ludwig.datasets import mushroom_edibility
from ludwig.utils.dataset_utils import get_repeatable_train_val_test_split

mushroom_df = mushroom_edibility.load()
mushroom_edibility_df = get_repeatable_train_val_test_split(mushroom_df, 'class',
random_seed=42)

auto_train_results = auto_train(
    dataset=mushroom_edibility_df,
    target='class',
    time_limit_s=7200,
    tune_for_memory=False,
    user_config={'preprocessing': {'split': {'column': 'split', 'type': 'fixed'}}},
)

pprint.pprint(auto_train_results)
```

RapidMiner

- Es una herramienta comercial centrada en automatizar visualmente proyectos de ML y ciencia de datos
- Disponible para Windows, MacOS o Linux
- Permite definir flujos de datos visualmente a través de diagramas (tipo blueprint)
- https://rapidminer.com

Otras herramientas

- Amazon SageMaker AutoPilot: https://aws.amazon.com/es/sagemaker/autopilot/



- Google Cloud AutoML: https://cloud.google.com/automl
  - Parte de una solución mayor: Vertex AI

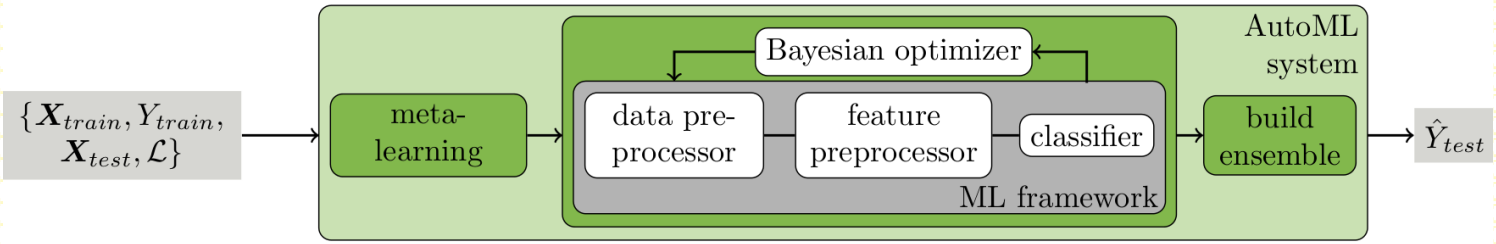
citic

Optuna

- Entorno de optimización de hiperparámetros (HPO)
- Herramientas soportadas:
  - Pytorch, TensorFlow, Keras
  - Chainer, MXNet, Scikit-Learn, XGBoost
- Definición de espacios de búsqueda en Python
- Mecanismos eficientes para
  - Elegir los mejores candidatos del espacio de búsqueda
  - Purgar candidatos poco prometedores
- Maneja los concpetos de estudio (*study*) y *trial*
- https://optuna.org

```python
import torch

import optuna

# 1. Define an objective function to be maximized.
def objective(trial):

    # 2. Suggest values of the hyperparameters using a trial object.
    n_layers = trial.suggest_int('n_layers', 1, 3)
    layers = []

    in_features = 28 * 28
    for i in range(n_layers):
        out_features = trial.suggest_int(f'n_units_l{i}', 4, 128)
        layers.append(torch.nn.Linear(in_features, out_features))
        layers.append(torch.nn.ReLU())
        in_features = out_features
    layers.append(torch.nn.Linear(in_features, 10))
    layers.append(torch.nn.LogSoftmax(dim=1))
    model = torch.nn.Sequential(*layers).to(torch.device('cpu'))
    ...
    return accuracy

# 3. Create a study object and optimize the objective function.
study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=100)
```

- Auto-SKLearn
  - Es una herramienta de AutoML propia para SKLearn
  - Permite definir un *budget* de tiempo por *trial* y uno global
  - Proyecto activo: https://github.com/automl/auto-sklearn



- Auto-Keras
  - Es un sistema de AutoML para Keras
  - Interfaz muy simple
  - Proyecto activo: https://github.com/keras-team/autokeras

citic