Javier Quiles Torregrosa

Inferència de models a gran escala amb Ray

TREBALL DE FI DE GRAU

dirigit per Marc Sánchez Artigas

Grau d'Enginyeria Informàtica

UNIVERSITAT ROVIRA I VIRGILI

Tarragona

2024

Agraïments

Aquest Treball de Fi de Grau és el resultat de diversos mesos de treball i aprenentatge. M'agradaria mostrar el meu agraïment a totes aquelles persones que han fet possible la seva realització.

En primer lloc, al meu tutor Marc Sánchez Artigas, doctor en Enginyeria Informàtica i investigador de la URV, per proposar-me aquest tema i donar-me suport al llarg de tot el procés de recerca.

També m'agradaria donar les gràcies al grup de recerca CloudLab per proporcionar-me les eines necessàries per a la realització del treball. Especialment a l'investigador Josep Calero Santo, per la seva valuosa ajuda, col·laboració i paciència amb els dubtes sorgits.

Finalment, però no menys important, agraeixo als meus companys, a la meva mare i al meu avi pel seu suport incondicional i motivació durant tot el procés.

Resum.

En els darrers anys, la digitalització i l'automatització de processos han provocat un augment exponencial en la quantitat de dades generades en diversos àmbits, plantejant el repte de com processar i analitzar aquesta informació per extreure'n valor útil. Una de les solucions és l'aprenentatge automàtic, una branca de la intel·ligència artificial que se centra en el desenvolupament de models capaços de fer prediccions automàticament. La realització d'aquestes prediccions es pot fer de diverses formes, una d'elles es coneix com a inferència per lots fora de línia. El procés d'inferència és lent, amb un gran cost computacional i difícilment paral·lelitzable.

Recentment, el Laboratori Europeu de Biologia Molecular ha desenvolupat el model d'aprenentatge automàtic "Off-sample" per a la identificació de mostres d'espectrometria de masses contaminades durant el tractament previ necessari per a la seva anàlisi. En aquest treball, es vol trobar la millor manera d'utilitzar aquest model. En primer lloc, s'ha fet una recerca d'informació sobre les característiques que ofereix Ray i el seu funcionament. A continuació, s'ha implementat el codi per fer la inferència per lots fora de línia utilitzant el model Off-sample. Posteriorment, s'ha desplegat un clúster de Ray amb Kubernetes. Finalment, s'han dut a terme diversos experiments amb el clúster per estudiar-ne el comportament del codi desenvolupat, optimitzar els paràmetres configurables i trobar-ne la millor configuració pel clúster.

Resumen.

En los últimos años, la digitalización y la automatización de procesos han provocado un aumento exponencial en la cantidad de datos generados en varios ámbitos, planteando el reto de cómo procesar y analizar esta información para extraer valor útil. Una de las soluciones es el aprendizaje automático, una rama de la inteligencia artificial que se centra en el desarrollo de modelos capaces de hacer predicciones automáticamente. La realización de estas predicciones se puede hacer de varias formas, una de ellas se conoce como inferencia por lotes fuera de línia. El proceso de inferencia es lento, con un gran costo computacional y difícilmente paralelizable.

Recientemente, el Laboratorio Europeo de Biología Molecular ha desarrollado el modelo de aprendizaje automático "Off-sample" para la identificación de muestras de espectrometría de masas contaminadas durante el tratamiento previo necesario para su análisis. En este trabajo, se quiere encontrar la mejor manera de utilizar este modelo. En primer lugar, se ha realizado una búsqueda de información sobre las características que ofrece Ray y su funcionamiento. A continuación, se ha implementado el código para hacer la inferencia por lotes fuera de línia utilizando el modelo Off-sample. Posteriormente, se ha desplegado un clúster de Ray con Kubernetes. Finalmente, se han llevado a cabo diversos experimentos con el clúster para estudiar el comportamiento del código desarrollado, optimizar los parámetros configurables y encontrar la mejor configuración para el clúster.

Abstract.

In recent years, digitization and process automation have led to an exponential increase in the amount of data generated in various fields, posing the challenge of how to process and analyze this information to extract useful value. One solution to this challenge is Machine Learning

, a branch of artificial intelligence that focuses on developing models capable of making predictions automatically. Making these predictions can be achieved through various methods; one such method is known as offline batch inference. However, the inference process is slow, computationally expensive, and hardly parallelizable.

Recently, the European Molecular Biology Laboratory has developed the "Off-sample" Machine Learning model for identifying contaminated mass spectrometry samples during the pre-treatment necessary for their analysis. In this work, our goal is to determine the optimal utilization of this model. First, we conducted research to gather information about the features offered by Ray and assess its performance. Then, we implemented the code to perform offline batch inference using the Off-sample model. Subsequently, we deployed a Ray cluster using Kubernetes. Finally, we conducted several experiments with the cluster to study the behavior of the developed code, optimize configurable parameters, and find the best configuration for the cluster.

Índex

Índex de taules

Índex de figures

Índex de codi

Índex de gràfics

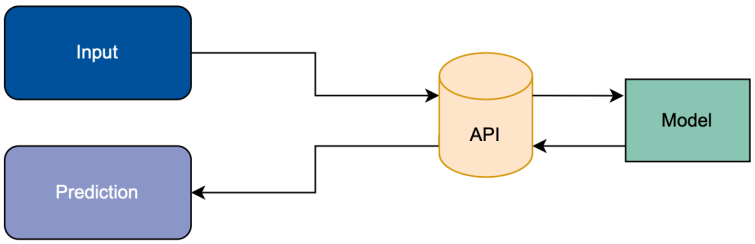
1 Introducció

1.1 Context històric

En els darrers anys, donada la digitalització de la societat i l'automatització dels processos s'ha experimentat un augment exponencial de la quantitat de dades generades en molts àmbits com la ciència, la tecnologia, el comerç i la medicina. Això està ocasionant que les organitzacions s'enfrontin al repte d'haver de processar i analitzar aquests volums d'informació per extreure'n valor útil pel desenvolupament de les seves activitats. L'anàlisi manual d'aquestes dades per obtenir informació rellevant cada vegada és més difícil i laboriós.

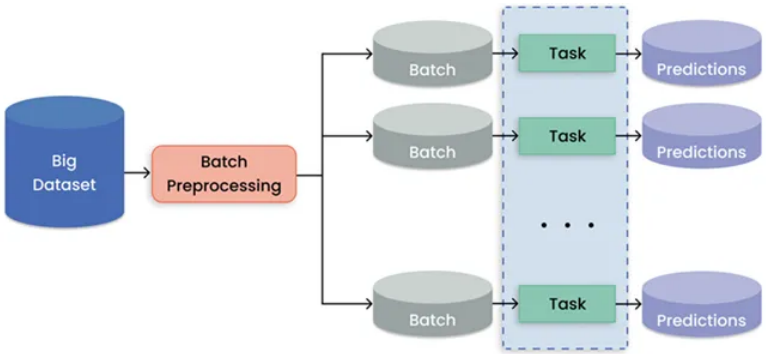
Per abordar aquest problema, sorgeix el Machine Learning (ML) com una resposta a la creixent complexitat i volum de dades generades en l'actual era digital. El ML és un camp de la intel·ligència artificial (IA) que se centra a desenvolupar models que permetin fer prediccions. Per l'aplicació d'aquest, hi ha dos enfocaments diferents, la inferència en línia o per lots fora de línia:

- La **inferència en línia** o **online inference** implica el processament de dades en temps real, permetent a les aplicacions i serveis respondre immediatament als canvis i esdeveniments. Aquest tipus d'inferència és necessària per a aplicacions que requereixen respostes instantànies, com la detecció de frau, la personalització de continguts i els sistemes de recomanació en temps real.



Il·lustració 1. Online inference

- La **inferència per lots fora de línia** o **offline batch inference**, es realitza sobre dades prèviament recopilades i emmagatzemades. Aquest enfocament resulta útil per a anàlisis que no requereixen respostes immediates, però que necessiten un processament intensiu, donat el volum de les dades a analitzar.



Il·lustració 2. Offline batch inference

1.2 Tendències

L'interès de les empreses en el Machine Learning ha crescut exponencialment en els darrers anys. La necessitat de processar i analitzar grans volums de dades de manera eficient és una prioritat en les organitzacions. No obstant això, històricament, aplicar el ML ha estat un procés complicat i poc intuïtiu. Desplegar models requeria equips amb coneixements avançats. A més del repte afegit de la paral·lelització del procés, ja que les tasques de ML requereixen un processament intensiu. Això complica encara més la seva utilització, pel fet de que no totes les empreses estan preparades per gestionar aquest tipus de càrregues de treball. Aquesta barrera tècnica limita l'adopció generalitzada del Machine Learning.

1.3 Aplicabilitat

El Laboratori Europeu de Biologia Molecular (EMBL) [1] treballa diàriament amb una gran quantitat d'informació que històricament ha sigut analitzada per experts en la matèria. Un tipus d'imatges amb les quals treballen són les obtingudes amb l'espectrometria de masses (MS) [2], que és una tècnica que permet la identificació i quantificació precisa de molècules d'una mostra determinada. La metodologia més utilitzada és la desorció/ionització làser assistida per matriu (MALDI), que utilitza un làser per convertir les molècules d'una mostra en ions, facilitant l'anàlisi. No obstant això, durant el procés de preparació de les mostres, sovint es produeix una contaminació amb ions que no formen part de la mostra original sinó del procés de preparació de la mostra, coneguts com a ions fora de la mostra (off-sample ions). Aquesta contaminació pot complicar l'anàlisi, dificultant la identificació correcta de les substàncies d'interès i distorsiona els resultats. En la següent figura obtinguda del repositori Metaspace [3] es pot visualitzar un exemple de mostra correcta i una fora de mostra.

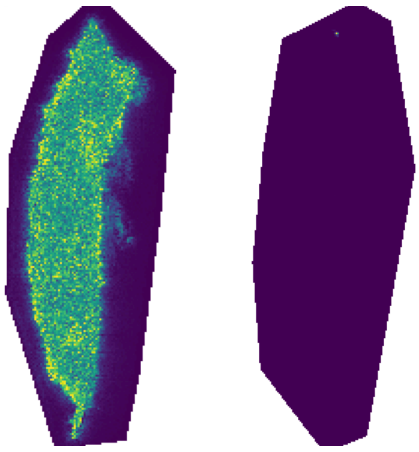


Figura 1. On-sample vs Off-sample

El flux de treball és el següent. Les mostres preparades passen al procés d'anotació de metabòlits que implica calcular patrons isotòpics, extreure imatges d'ions, calcular puntuacions de coincidència, filtrar i guardar els resultats a una base de dades. També es converteixen les imatges a PNG. Finalment s'identifiquen si les mostres són correctes. Donats els avanços recents en IA, es vol començar a aplicar el ML en aquest procés d'identificació. Per aquest motiu, l'EMBL han entrenat un model d'aprenentatge automàtic a partir d'una gran quantitat de dades prèviament etiquetades per experts obtingudes del seu repositori Metaspace. El model creat es coneix com Off-sample [4]. Ara s'enfronten amb el problema de com posar en producció aquest model. Han decidit que donada la gran quantitat de dades que volen processar la millor opció és realitzar la inferència per lots.

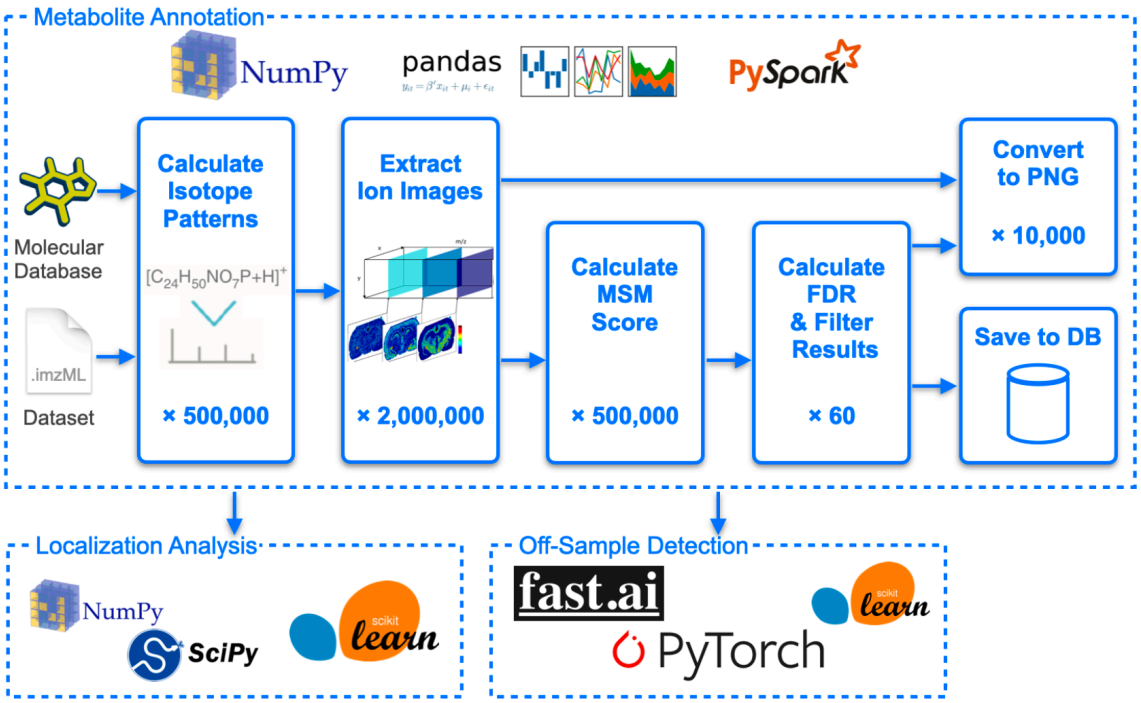


Figura 2. Flux de treball EMBL [5]

1.4 Reptes actuals

Els principals reptes actuals resideixen en la dificultat de la programació d'un codi distribuït i en com servir eficientment els models de Machine Learning. La computació distribuïda a gran escala té problemes com:

- Gestionar la infraestructura distribuïda.
- Paral·lelitzar el procés d'inferència.
- Optimitzar la utilització de recursos.
- Ajustar-se a la variabilitat de la demanda.

Per resoldre aquests problemes, i donada la creixent demanda de solucions de ML, han sorgit múltiples eines i plataformes que pretenen simplificar considerablement el procés d'utilització dels models de ML, això inclou l'entrenament i el desplegament de models a gran escala.

1.5 Eines actuals

Algunes de les eines que han sorgit per satisfer la demanda del mercat en la inferència per lots fora de línia són:

- **Amazon SageMaker** [6]. Ofereix l'API Batch Transform que teòricament permet un processament per lots, però internament utilitza l'arquitectura de la inferència online, inicia un servidor HTTP, desplega el model com un endpoint on es fa una sol·licitud per cada fitxer. A més, arquitectònicament té un límit de 100 MB per sol·licitud, la qual cosa provoca una infrautilització dels recursos.

- **Apache Spark** [7]. Ofereix la llibreria d'alt nivell MLib que amb l'API Dataframe permet escalar la inferència per lots fora de línia. També es pot fer a més baix nivell amb el mòdul Spark SQL que té les User Defined Funcions (UDF)
- **Ray** [8]. Ofereix la llibreria d'alt nivell Ray Data amb una API, que amplia la base de Ray Core, facilitant l'escalat de la inferència per lots fora de línia i el preprocessament i ingesta de dades per a l'entrenament de models ML. També es pot fer amb baix nivell amb Ray Core.

Com s'ha vist, en general totes les plataformes tenen el propòsit de millorar el processament de dades en paral·lel.

1.6 Estudis previs

Un cop identificades les tecnologies, s'ha fet una recerca d'informació a internet i no s'ha trobat gairebé cap estudi comparatiu detallat. La informació disponible és escassa i es limita principalment a l'article "*Offline Batch Inference: Comparing Ray, Apache Spark, and SageMaker*" [9] publicat en el bloc d'Anyscale que són els creadors de Ray. En aquest article es compara Amazon SageMaker i Apache Spark amb Ray i es conclou que Ray és la millor eina per la inferència per lots fora de línia. Davant d'aquesta afirmació, s'ha decidit buscar més informació sobre Ray. No obstant, no s'ha trobat cap estudi detallat que aprofundeixi en les seves característiques i el seu rendiment.

1.7 Objectius de la recerca

Aquest treball de final de grau pretén oferir una visió detallada de com implementar la inferència per lots fora de línia amb Ray, que segons la informació trobada és la millor eina. Concretament, es centrarà a explorar les següents qüestions clau:

- Implementar la paral·lelització de la inferència per lots fora de línia utilitzant Ray per un cas d'ús real utilitzant el model Off-sample del Laboratori Europeu de Biologia Molecular per la identificació d'imatges fora de mostra.
- Implementar en un entorn real un clúster de Ray amb Kubernetes.
- Buscar els millors paràmetres de configuració de la inferència per lots fora de línia per optimitzar el rendiment.

2 Planificació

En la següent imatge es pot veure el diagrama de Gantt amb la planificació seguida durant aquest treball de fi de grau:

Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
Determinació tema del projecte	7 días	mar 20/02/24	lun 26/02/24	
Fixar objectius inicials	0 días	mar 27/02/24	mar 27/02/24	1
Recerca d'informació Ray i documentació	16 días	mar 27/02/24	mié 13/03/24	2
Preparació entorn desenvolupament	2 días	mar 27/02/24	mié 28/02/24	1
Desenvolupament codi base amb model ResNet50	7 días	jue 29/02/24	mié 06/03/24	4
Proves codi ResNet50	5 días	jue 07/03/24	lun 11/03/24	5
Adaptacions MinIO	2 días	mar 12/03/24	mié 13/03/24	6
Determinació següents passos	0 días	mié 13/03/24	mié 13/03/24	7;3
Adaptacions codi model EMBL	7 días	jue 14/03/24	mié 20/03/24	8
Proves codi EMBL	5 días	jue 21/03/24	lun 25/03/24	9
Desenvolupament codi grid search	7 días	mar 26/03/24	lun 01/04/24	10
Proves codi grid search	5 días	mar 02/04/24	sáb 06/04/24	11
Documentació codi	7 días	dom 07/04/24	sáb 13/04/24	12
Recerca d'informació Kubernetes	5 días	dom 14/04/24	jue 18/04/24	13
Desplegament clúster local Kubernetes	2 días	vie 19/04/24	sáb 20/04/24	14
Adaptacions codi clúster	5 días	dom 21/04/24	jue 25/04/24	15
Proves clúster local	5 días	vie 26/04/24	mar 30/04/24	16
Documentació Kubernetes	3 días	mié 01/05/24	vie 03/05/24	17
Permís utilització rack URV	0 días	vie 03/05/24	vie 03/05/24	18
Configuració entorn de treball rack	2 días	sáb 04/05/24	dom 05/05/24	19
Determinar experiments clúster	3 días	lun 06/05/24	mié 08/05/24	20
Adaptació codi experiments	4 días	jue 09/05/24	dom 12/05/24	21
Executar experiments clúster	7 días	lun 13/05/24	dom 19/05/24	22
Documentació experiments	7 días	lun 20/05/24	dom 26/05/24	23
Preparar presentació	7 días	lun 27/05/24	dom 02/06/24	24
Revisió documentació	5 días	lun 03/06/24	vie 07/06/24	25
Entrega	0 días	vie 07/06/24	vie 07/06/24	26

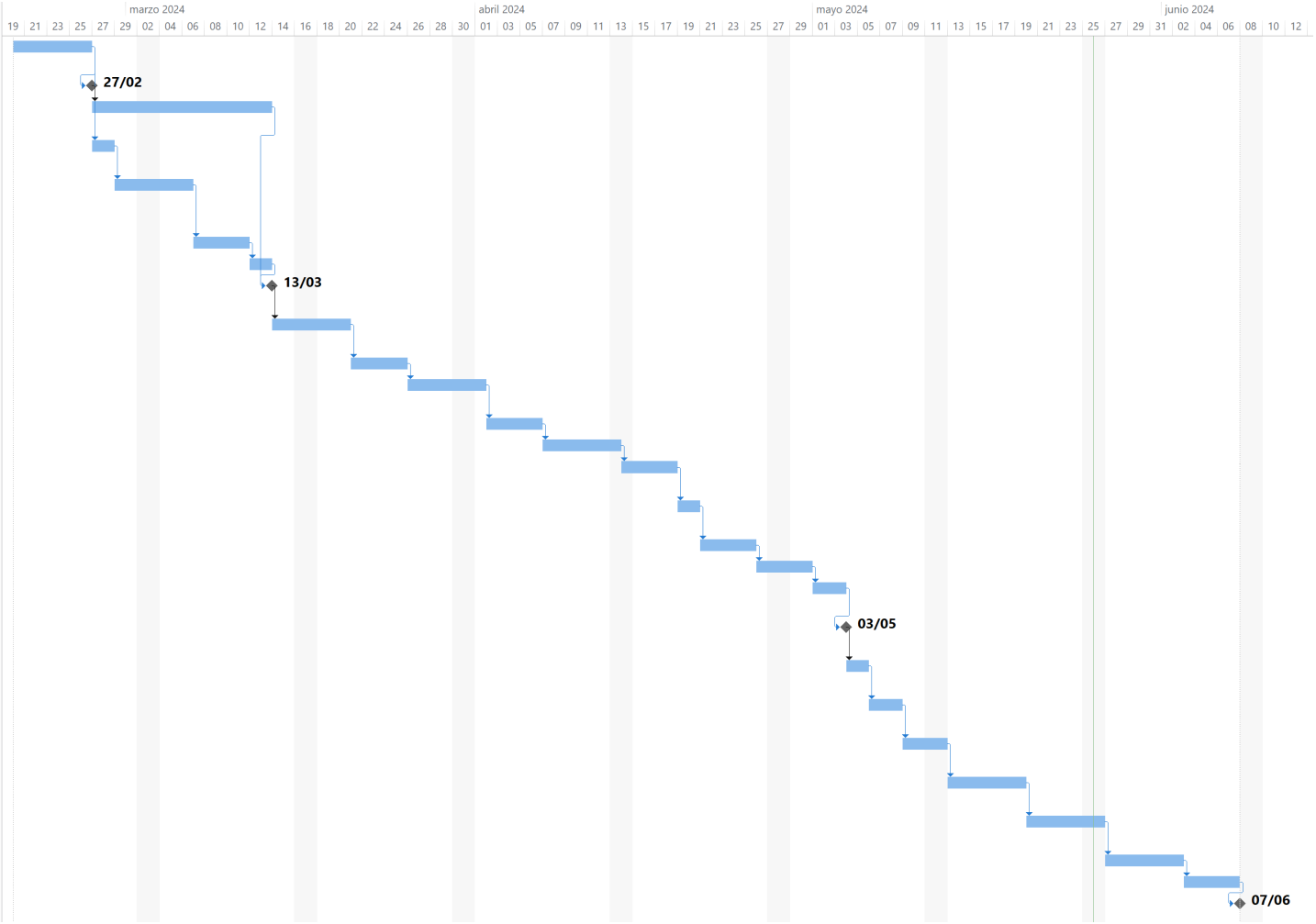


Figura 3. Diagrama de Gantt

3 Tecnologies utilitzades

Primerament, en aquesta secció s'han explorat les tecnologies que s'utilitzaran durant el desenvolupament d'aquest projecte, concretament el framework[1] de Ray, diversos models preentrenats[2], el llenguatge de programació Python, el servidor d'emmagatzematge MinIO i Kubernetes. La investigació s'ha dut a terme des d'una perspectiva més general fins a una més específica.

3.1 Ray

Inicialment, s'ha realitzat una recerca prèvia per comprendre l'abast del seu funcionament i les característiques que ofereix. A continuació s'analitza en detall.

3.1.1 Que és Ray?

Ray [8] és un framework unificat de codi obert per la computació paral·lela i distribuïda que facilita escalar càrregues de treball existents de Python i d'intel·ligència artificial, concretament en l'àrea de l'aprenentatge automàtic[3]. Permet passar des d'un computador a un clúster[4] ràpidament. Els seus inicis van començar al RISELab de la Universitat de Califòrnia (Berkeley), però actualment és desenvolupat per la startup Anyscale creada pels fundadors de Ray. Aquest framework es troba en un desenvolupament actiu amb constants canvis i evolució.

3.1.2 Capes

El framework de Ray [10] està format per tres capes.

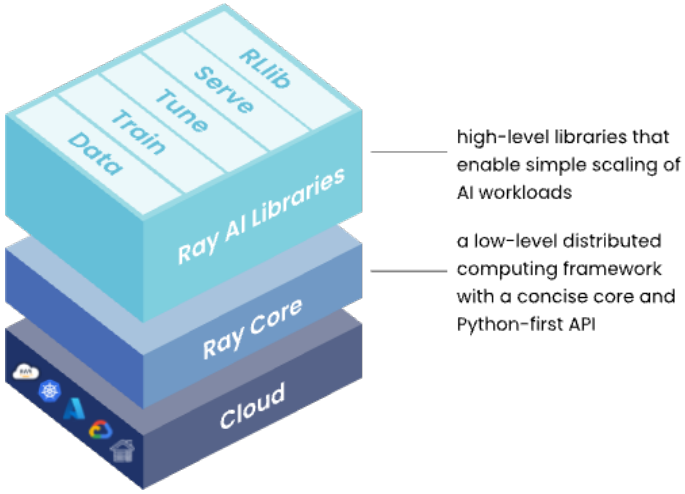


Figura 4. Visió general framework de Ray

[1] Un framework és un conjunt estructurat de directrius, eines i components predefinits que proporcionen una base per desenvolupar aplicacions de programari.

[2] Un model preentrenat és un model que s'ha entrenat amb un gran conjunt de dades per dur a terme una tasca específica, com ara el reconeixement d'imatges, el processament del llenguatge natural o el reconeixement de veu.

[3] L'aprenentatge automàtic o Machine Learning (ML) és un camp de la intel·ligència artificial que se centra en l'ús de dades i algorismes per permetre que la IA imiti la forma en què els humans aprenen, millorant gradualment la seva precisió.

[4] Un clúster és un conjunt d'ordinadors interconnectats que funcionen conjuntament com un únic sistema.

El propòsit de cada capa és el següent:

- **Ray AI Libraries:** és un conjunt de cinc llibreries destinades a facilitar l'escalament de tasques comunes d'aprenentatge automàtic.
- **Ray Core:** proporciona un petit nombre de primitives bàsiques com tasques, actors i objectes per construir i escalar aplicacions distribuïdes amb Python.
- **Ray Cluster:** facilita la creació d'un clúster que és un conjunt de worker nodes administrats per un head node.

3.1.3 Ray AI Libraries

Les llibreries de Ray pretenen simplificar la plataforma d'aprenentatge automàtic respecte a altres opcions del mercat. Proporciona les eines necessàries perquè el flux de treball sigui d'extrem a extrem. Cadascuna de les llibreries té un propòsit específic:

- **Ray Data:** aquesta llibreria està dissenyada per gestionar i manipular dades. Inclou funcions per a la càrrega, la preparació, la neteja i l'anàlisi de dades que s'utilitzaran com a entrada per entrenar models, ajustar-los o fer prediccions.
- **Ray Train:** aquesta llibreria està dissenyada per l'entrenament distribuït de models d'intel·ligència artificial. Proporcionar les eines i funcionalitats necessàries. S'integra amb llibreries d'entrenament existents com PyTorch.
- **Ray Tune:** aquesta llibreria està especialitzada a ajustar els hiperparàmetres dels models per optimitzar el seu rendiment. Els hiperparàmetres són els paràmetres que no s'aprenen durant el procés d'entrenament, sinó que s'han d'ajustar manualment.
- **Ray Serve:** aquesta llibreria està enfocada a desplegar models en línia. Facilita la creació d'una API per la inferència en línia i l'autoescalament dels recursos necessaris per servir les peticions.
- **Ray RLlib:** aquesta llibreria se centra específicament en el reforçament d'aprenentatge, en anglès Reinforcement Learning (RL). Proporciona algorismes RL, eines d'entrenament, mètriques d'avaluació i funcionalitats per a l'entrenament i el desplegament de models RL.

3.1.4 Ray Core

Ray Core [11] és el component principal del framework Ray i la base de les Ray AI Libraries. Proporciona la infraestructura fonamental i les abstraccions necessàries per a la computació paral·lela i distribuïda amb Python. Ray Core és responsable d'administrar els recursos, coordinar l'execució de tasques, gestionar la comunicació entre processos i proporcionar les interfícies de programació necessàries per a construir aplicacions escalables i eficients. Proporciona la Core API[5] que és un conjunt de primitives per crear i escalar aplicacions distribuïdes utilitzant Python.

[5] Una API (Application Programming Interface) és un conjunt de regles, protocols i eines que permeten a diferents aplicacions de programari comunicar-se entre si.

Les característiques més importants són les següents:

- **Tasques**

  Les tasques són equivalents a les funcions de Python i són *stateless*[6] (sense estat). Permeten configurar els requisits de CPU i GPU necessaris per a la seva execució, per defecte utilitzen una CPU. S'executen asincronament en un worker process[7] de qualsevol node que compleixi amb els requisits.

- **Actors**

  Els actors són equivalents a les classes de Python, però són *statefull*[8] (amb estat). També permeten configurar els requisits d'execució, és obligatori indicar la concurrència que és el nombre d'actors que s'executen simultàniament i es recomana indicar el número de CPUs. A diferència de les tasques, el worker process que executa l'actor es crea específicament i es destrueix quan finalitza.

- **Objectes**

  Els objectes són utilitzats i/o creats per tasques i actors. Es coneixen com a objectes remots perquè poden guardar-se en qualsevol lloc del clúster i s'accedeixen mitjançant una referència.

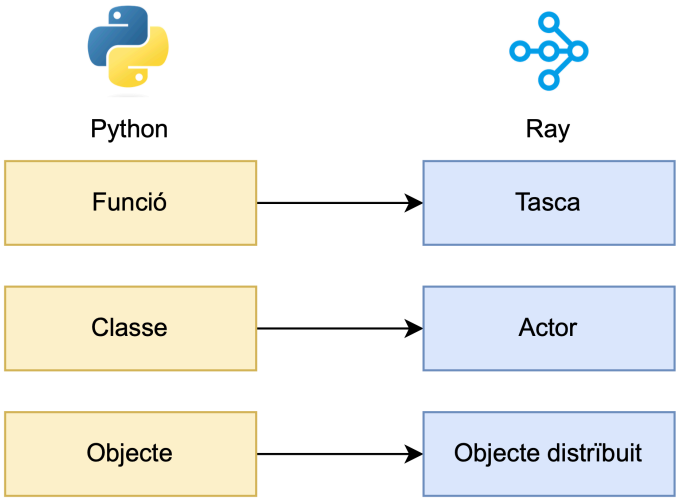


Figura 5. Relació Python amb Ray

3.1.5 Ray Cluster

Un clúster de Ray [12] és un conjunt de nodes de computació interconnectats que s'utilitzen per a executar aplicacions distribuïdes programades amb el framework Ray. Es troba format per un node principal (head node) i qualsevol nombre de nodes de treball (worker nodes). El clúster es pot configurar per escalar automàticament en funció de la demanda de recursos i la configuració establerta.

---

[6] Una tasca stateless cada vegada que s'executa no depèn de cap estat anterior.

[7] Un worker process (o ray worker) és un procés de Python encarregat d'executar tasques o actors.

[8] Un actor statefull té un estat intern persistent que es pot utilitzar per determinar la sortida.

El node principal és similar als nodes de treball, però a més s'encarrega d'executar els processos d'administració de Ray que inclouen l'autoescalador (que és opcional i permet ajustar els recursos en funció de la demanda), el Global Control Service (que és un magatzem de clau-valor amb metadades del sistema) i el driver process (que executa les funcions d'alt nivell).

En la següent imatge es pot visualitzar un exemple d'un Ray Cluster amb un head node i dos worker nodes.

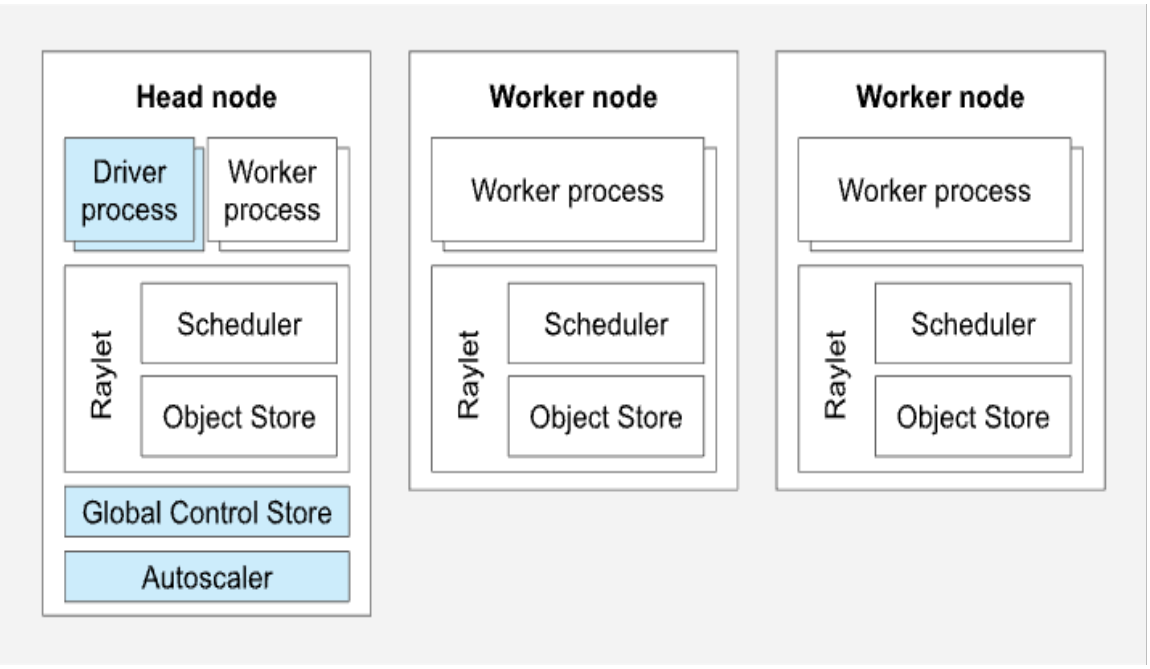


Figura 6. Esquema Ray Cluster amb un head node i dos worker nodes

A continuació es pot visualitzar com es distribueix l'execució d'una petita porció de codi en un clúster. En blau es representa el procés controlador que s'encarrega de gestionar la resta de processos representats en verd.

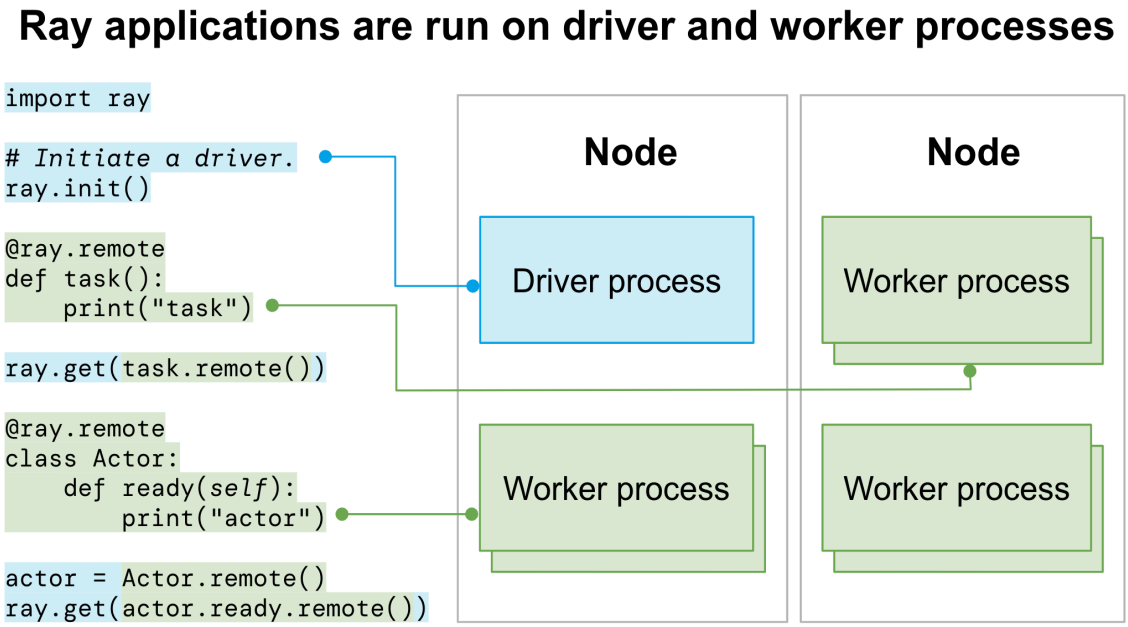


Figura 7. Exemple de la distribució de l'execució del codi en un clúster

Les opcions per desplegar un clúster de Ray és utilitzar Kubernetes [13] o VMs [14]. En el cas de Kubernetes es proporciona l'operador KubeRay que facilita la creació i administració de clústers, proporcionant 3 recursos:

- **RayCluster:** permet administrar el cicle complet d'un clúster (creació, eliminació, redimensionament i tolerància a errors)
- **RayJob:** permet crear un clúster quan s'envia un treball i l'elimina al finalitzar-lo.
- **RayService** permet servir models.

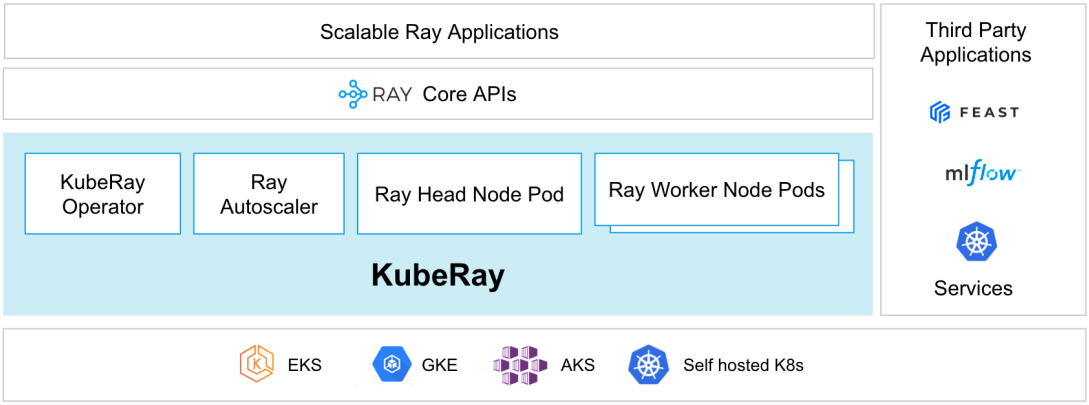


Figura 8. Representació clúster amb KubeRay

Respecte al autoescalador [15], el funcionament és el següent:

1. L'usuari envia un treball de Ray.
2. El head node calcula tots els requisits de recursos del treball i els comunica al autoescalador.
3. L'autoescalador decideix afegir els Pods (worker nodes) necessaris per satisfer els requisits d'execució.
4. L'autoescalador sol·licita els Pods de treball addicionals augmentant les rèpliques del clúster.
5. L'operador KubeRay crea els worker nodes sol·licitats.
6. El planificador col·loca la càrrega de treball als nous worker nodes.

Quan els worker nodes estan un determinat temps sense utilització, per defecte 60 segons, automàticament s'eliminen.

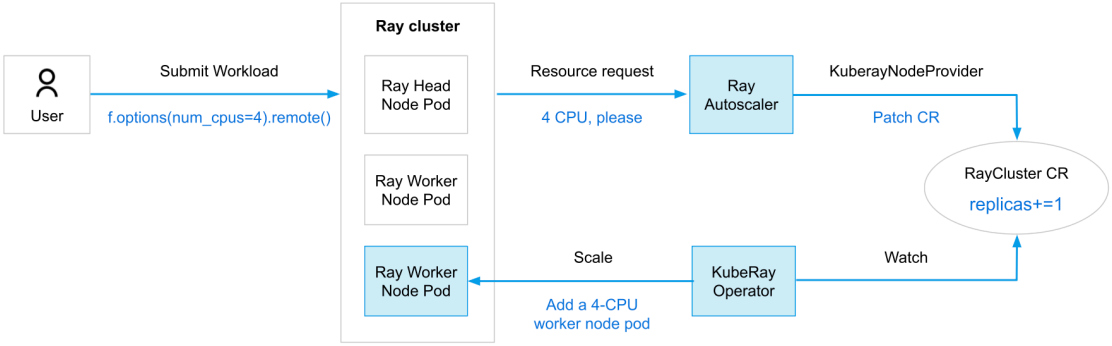


Figura 9. Diagrama procés autoescalament clúster

3.1.6 Ray Jobs

Un Ray Job [16] és el conjunt de tasques, actors i objectes que s'originen a partir del mateix codi i s'executen en un clúster. Els treballs es poden executar de dues formes diferents:

- Mitjançant l'API d'enviament de tasques (Ray Job Submit).
- Executant el codi directament en un node.

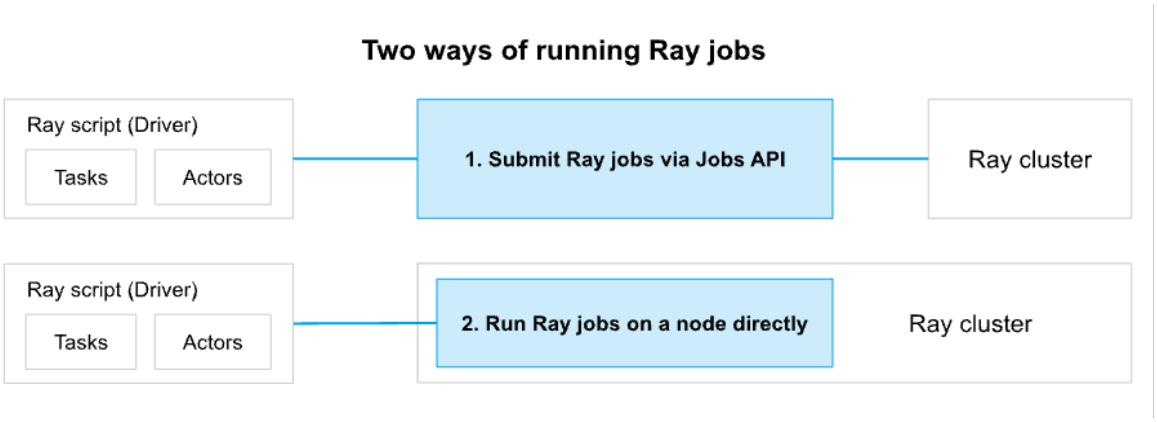


Figura 10. Diagrama mètodes d'execució de tasques

3.1.7 Ray Data

L'objectiu d'aquest treball de fi de grau és investigar la inferència per lots fora de línia, Ray Data [17] és la llibreria que proporciona els recursos necessaris per realitzar aquest procés. A continuació s'explicarà que és i quins recursos proporciona.

Què és la inferència per lots fora de línia (Offline Batch Inference)?

La inferència per lots fora de línia [18] és un procés essencial en el camp de la intel·ligència artificial i l'aprenentatge automàtic. Implica la realització de prediccions o decisions sobre un conjunt de dades d'entrada mitjançant l'ús d'un model preentrenat. En lloc de processar les entrades individualment, com es fa amb la inferència en línia, s'agrupen les dades d'entrada en lots i es processen de manera conjunta.

En la següent imatge es mostra de manera simplificada el procés descrit. A partir d'un batch que són les imatges, el model de ML genera unes prediccions amb la classificació que ha determinat.

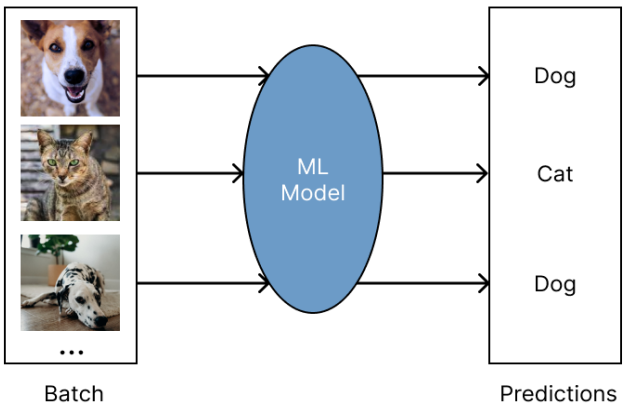


Figura 11. Concepte de processament per lots

Realment aquest procés és una mica més complex. Les fases necessàries es desglossen en:

1. **Carregar les dades:** aquesta fase implica la recopilació de les dades des de l'origen, que pot ser un directori local o remot al núvol.
2. **Preprocessar les dades:** aquesta fase és necessària per a preparar les dades per a la inferència. Inclou tasques com la normalització, el redimensionament i qualsevol altre preprocessament necessari segons les especificacions del model utilitzat.
3. **Fer la inferència amb el model:** aquesta fase es divideix en dues subfases:
    a. **Carregar el model a memòria i inicialitzar-lo:** el model s'ha de carregar a la memòria i s'ha d'inicialitzar. Normalment, és un procés costos i es realitza un únic cop.
    b. **Realitzar la inferència per a cada batch:** a continuació es realitza la inferència amb el model precarregat per a cada batch.
4. **Guardar els resultats:** finalment, els resultats de les classificacions fetes pel model es poden guardar pel seu posterior ús.

Ray Data Internals (Datasets)

La llibreria Ray Data treballa amb datasets per organitzar els objectes. Un dataset és un conjunt de referències a diversos blocs. Un bloc és la unitat mínima de processament de Ray. En la següent imatge es pot visualitzar un dataset amb tres blocs de 1000 elements cadascun.

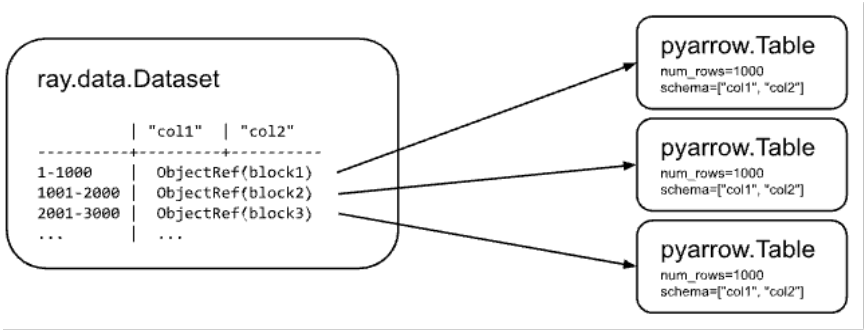


Figura 12. Dataset amb tres blocs

Execució d'operacions en un dataset

L'execució de les operacions sobre el dataset té dues característiques destacades [19]:

- **Lazy:** vol dir que les transformacions no s'executen fins que s'utilitza una operació de consum. Això permet fer optimitzacions en l'execució com l'operator fusion evitant l'execució repetida del garbage collection[9].
- **Streaming:** vol dir que les transformacions s'apliquen incrementalment quan les dades es troben disponibles, en lloc de processar-lo tot com un conjunt, permeten superposar (overlap) les posteriors fases. Això permet millorar la

[9] El garbage collection és un mecanisme automàtic que maneja la memòria dinàmica assignada a objectes que ja no són necessaris en un programa. Alliberant-la quan no són necessaris.

utilització dels recursos, reduint els temps d'execució i evitar quedar-se sense memòria quan el dataset és massa gran. A continuació hi ha dues imatges on es pot veure la diferència entre la utilització o no de l'execució en streaming.

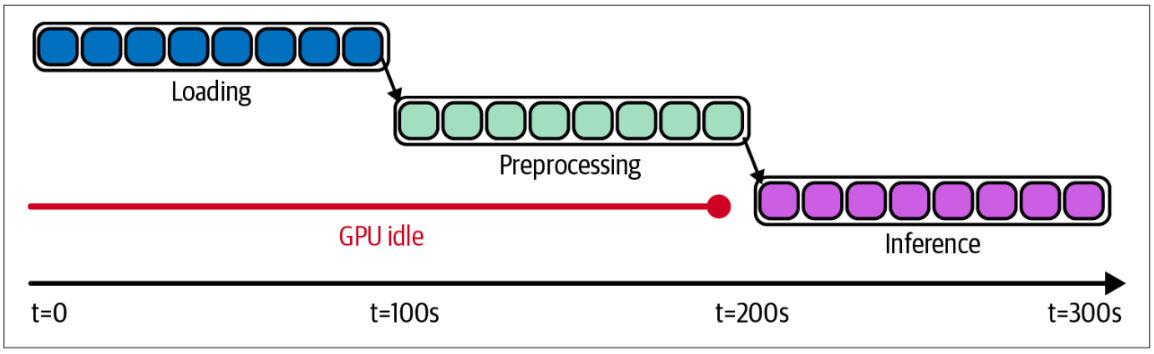


Figura 13. Seqüència d'execució estàndard

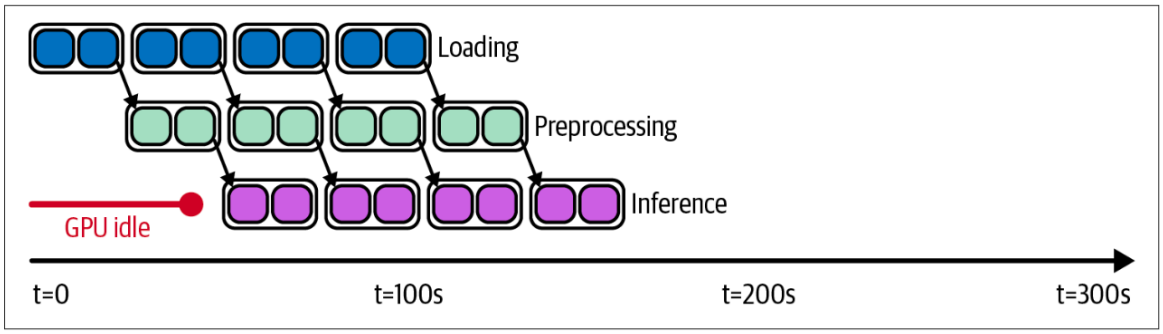


Figura 14. Seqüència d'execució en streaming

Operacions d'input/output

Per carregar la informació des de la font de les dades sobre el dataset hi ha una àmplia varietat d'APIs [20] de lectura per diversos tipus de dades depenent del format, per exemple existeixen funcions pels formats més comuns com imatges, parquet[10], CSV, JSON, etc. Aquestes funcions requereixen indicar les ubicacions *(paths)* d'on es troben emmagatzemades les dades, que pot ser una carpeta local o el núvol, per defecte és compatible amb l'API d'Amazon S3[11]. A més dels paths, es poden indicar altres paràmetres opcionals com el paral·lelisme de la lectura, les dimensions i el mode[12] resultant de les imatges carregades.

El paral·lelisme equival al nombre de tasques (worker process) que llegeixen fitxers i conseqüentment el nombre de blocs que es generen. Aquestes tasques utilitzen per defecte 1 CPU. El paral·lelisme es determina seguint la següent heurística en ordre [21]:

[10] Un parquet és un format de fitxer d'emmagatzematge columnar dissenyat per ser eficient en termes d'ús d'emmagatzematge i processament de dades en sistemes d'emmagatzematge distribuït.

[11] Amazon S3 (Simple Storage Service) és un servei d'emmagatzematge al núvol proporcionat per Amazon Web Services (AWS).

[12] El mode d'una imatge es refereix a la manera en què es representen els píxels a memòria. Un dels més comuns és el mode RGB on cada píxel es representa mitjançant tres valors que indiquen la intensitat de vermell, verd i blau.

1. S'intenta començar amb un paral·lelisme de 200 que equival a generar 200 blocs.
2. Si els blocs són menors de la mida de bloc mínima (1 MiB[13]) es redueix el paral·lelisme
3. Si els blocs són majors de la mida de bloc màxima (128 MiB) s'augmenta el paral·lelisme
4. S'intenta que el paral·lelisme sigui almenys el doble de les CPU disponibles al sistema.

Quan Ray selecciona automàticament un paral·lelisme es guarda al log[14] el motiu de l'elecció. El valor mínim de paral·lelisme a la lectura ve determinat pel nombre de fitxers. Finalment, cal destacar que si el paral·lelisme és superior al nombre de CPUs i estem a un clúster el sistema intentaria autoescalar automàticament.

En la següent imatge, es pot visualitzar un exemple de paral·lelisme. Es parteix del fet que la instància de Ray té dues CPUs, a l'esquerra no s'indica el paral·lelisme i segueix l'heurística del doble de CPUs disponibles. En canvi, a la dreta que s'indica paral·lelisme=2, es generen 2 tasques que llegeixen proporcionalment el mateix nombre de fitxers.

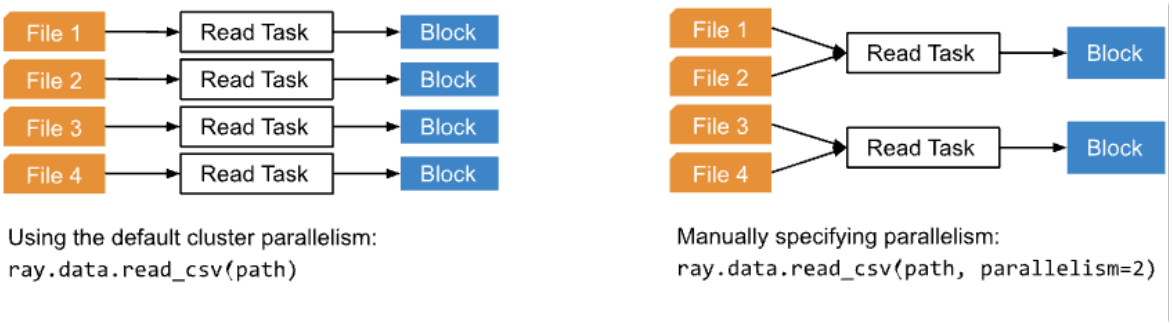


Figura 15. Lectura de fitxers

Per guardar els resultats existeix el mateix conjunt de funcions però d'escriptura. Això és útil per persistir els resultats de la inferència pel seu posterior ús.

Operacions sobre el dataset

Un cop carregades les dades sobre el dataset es poden aplicar dues operacions diferents:

- **Transformació:** pren el conjunt de dades i genera un de nou. Són del tipus lazy, un exemple de funció d'aquest tipus és map_batches(). Per defecte s'utilitzen tasques que s'inicialitzen cada cop en un worker process, això pot ser útil pel preprocessament. En el cas de la inferència no seria òptim carregar el model en cada inicialització, per aquest cas és recomanable fer servir actors que tenen estat i la càrrega del model tan sols es faria en la inicialització del worker process de l'actor.

[13] Un mebibyte és una unitat d'emmagatzemament informàtic. Equival a 2^{20} bytes.

[14] Un log o registre permet registra accions o esdeveniments del sistema.

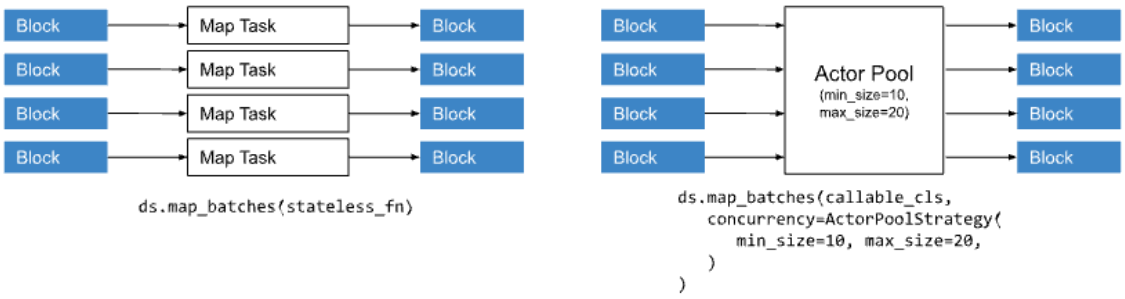


Figura 16. Exemple d'una tasca i un actor

- **Consum:** a partir del conjunt de dades i les transformacions pendents d'executar produeix valors com a sortida. Per exemple iter_batches().

Operator Fusion Optimization

Aquesta optimització consisteix a fusionar automàticament els operadors compatibles per reduir l'ús de memòria i l'overhead[15] de les tasques. Normalment, s'aplica sobre les operacions de lectura i les transformacions si segueixen el mateix patró de computació, la mateixa estratègia (tasques/actors) i si s'assignen els mateixos recursos. Es pot comprovar si alguna operació s'ha fusionat durant l'execució del codi o revisant les estadístiques del conjunt de dades amb la funció stats(), buscant-hi si les operacions es relacionen amb una fletxa (→). En la següent figura es pot veure un exemple.

```
Operator 1 ReadImage->MapBatches(preprocess): 200 tasks executed, 200 blocks produced in 7.58s
```

Figura 17. Exemple fusió d'operadors de lectura i preprocessament

Batch Optimization

Cal destacar que és més eficient realitzar operacions en batch en lloc de files quan es treballa amb operacions vectoritzades de Numpy. També és recomanable intentar augmentar al màxim possible sense esgotar la memòria disponible la mida del batch (batch_size), ja que millora el rendiment.

3.1.8 Observabilitat

Ray proporciona diverses eines d'observabilitat [22] que permeten monitorar l'estat del sistema, entendre el seu comportament, detectar errors i optimitzar-lo. Aquestes eines es divideixen en State CLI[16]/API i el Ray Dashboard.

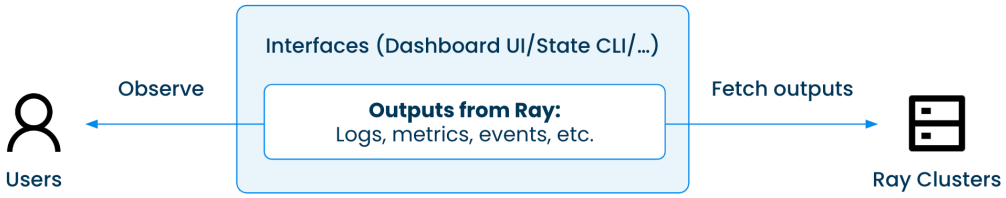


Figura 18. Esquema d'observabilitat

[15] L'overhead (sobrecarrega) és la utilització addicional de recursos per dur a terme una tasca específica.

[16] Command Line Interface (línia d'ordres) és un mecanisme per interactuar amb un programa sense interfície gràfica mitjançant l'ús del teclat.

State CLI/API

La State CLI/API permet veure l'estat del sistema mitjançant l'ús del terminal o amb l'API de Python. La informació que es pot extreure és un llistat complet o un resum de les tasques, els actors i els objectes. També es pot veure la informació completa d'un recurs a partir del seu identificador i obtenir logs del sistema.

Ray Dashboard

El dashboard de Ray està basat en la web. Funciona conjuntament amb Prometheus i Grafana, dues eines que s'han convertit en estàndards de facto per a la indústria.

- **Prometheus:** és una aplicació dissenyada per recopilar i emmagatzemar mètriques. Utilitza un model de dades de sèries temporals i es recopilen mitjançant un servidor. Les mètriques inclouen informació sobre l'ús de la CPU, la memòria, el trànsit de xarxa, i altres paràmetres del sistema.
- **Grafana:** és una aplicació que s'utilitza juntament amb Prometheus per visualitzar les mètriques. Permet crear panells interactius que representen les dades en temps real mitjançant gràfics, taules i altres elements visuals. Facilitant la interpretació de les dades recopilades, ajudant a identificar tendències i detectar problemes.

La web amb el dashboard té les següents finestres que permeten visualitzar la informació de Ray en temps real:

- **Overview:** mostra la utilització general del clúster, el nombre de nodes, els treballs recents, l'estat dels nodes, la utilització dels recursos i els esdeveniments recents.
- **Jobs:** mostra una llista de treballs executats i en execució. Cada treball es pot consultar detalladament amb informació sobre Ray Data, Ray Core, logs, una llista de tasques i una d'actors.
- **Serve:** mostra informació sobre Ray Serve, que és una de les llibreries de Ray, però no és el propòsit d'aquest treball investigar-ho.
- **Cluster:** mostra una llista de nodes del clúster, i per cada node la llista de processos (worker process) amb la utilització de recursos en temps real.
- **Actors:** mostra els actors actius i finalitzats amb detalls dels recursos utilitzats, logs i un historial d'execucions.
- **Metrics:** mostra visualment els gràfics de Grafana amb les dades recopilades per Prometheus.
- **Logs:** permet visualitzar ràpidament els logs del clúster.

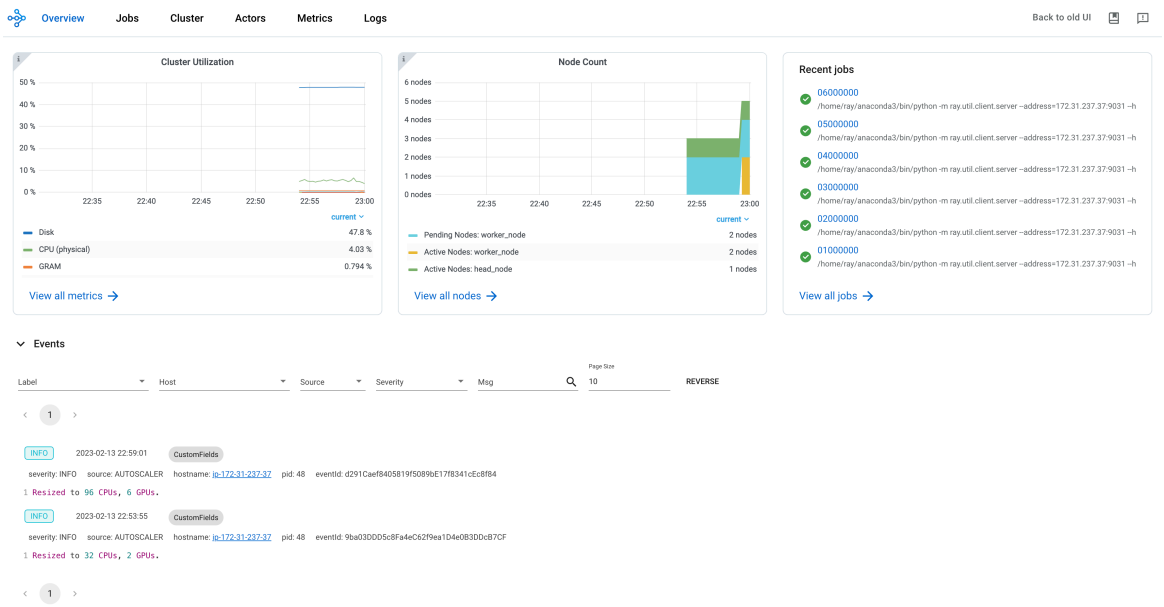


Figura 19. Ray Dashboard

3.2 Models

Per realitzar la inferència és necessari la utilització d'un model que prèviament ha estat entrenat amb un gran conjunt de dades. Ray té una llibreria específica per entrenar models (Ray Train) però l'objectiu d'aquest treball no és l'entrenament. Per aquest motiu s'utilitzarà models preentrenats.

En primer lloc, es farà servir la biblioteca d'aprenentatge automàtic PyTorch que conté la llibreria TorchVision amb models i pesos preentrenats [23]. Inicialment, s'ha experimentat amb el model ResNet50 [24] que està entrenat amb imatges d'ImageNet [25]. Utilitzar-lo permetrà familiaritzar-se amb Ray. L'estructura de ResNet50 és la següent:

- **Capes convolucionals (etapa 1):** en aquesta etapa, la imatge d'entrada es passa a través de capes que busquen patrons bàsics, com ara línies i formes. Aquestes capes convolucionals ajuden a la xarxa a comprendre l'estructura bàsica de la imatge.
- **Blocs residuals (etapes 2, 3, 4 i 5):** en aquestes etapes, la xarxa aprofundeix en l'anàlisi de la imatge. Els blocs residuals permeten a la xarxa aprendre característiques més complexes i detallades. Cada bloc residual ajusta la comprensió de la xarxa sobre la imatge, ajudant-la a reconèixer objectes específics i detalls concrets.
- **Capa completament connectada:** després de passar per les etapes anteriors, la informació es passa mitjançant una capa completament connectada. Aquesta capa pren totes les característiques apreses i les processa per produir una sortida final que indica quins objectes o característiques reconeix la xarxa a la imatge. És com l'etapa d'interpretació final on es decideix què hi ha a la imatge.

Durant l'entrenament, s'utilitzen imatges etiquetades que permeten ajustar els pesos de les capes convolucionals i dels blocs residuals per aprendre característiques.

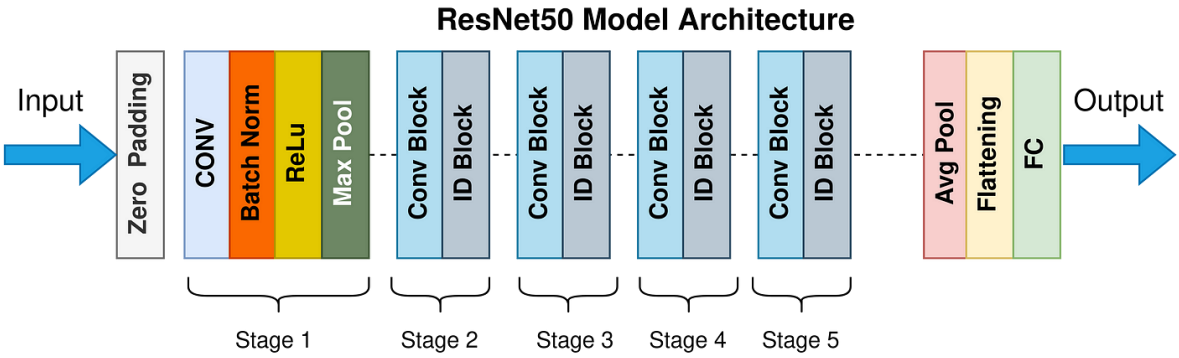


Figura 20. Estructura ResNet50

Més tard, s'utilitzarà el model Off-sample [4] desenvolupat pel Laboratori Europeu de Biologia Molecular (EMBL). Aquest model està dissenyat per identificar si una imatge d'espectrometria de masses és correcta (on-sample) o es troba fora de mostra (off-sample). Es basa en ResNet50, però està entrenat amb imatges de la llibreria MetaSpace etiquetades per experts. L'EMBL va utilitzar la biblioteca Fastai que simplifica la creació i entrenament de models de Machine Learning proporcionant abstraccions d'alt nivell que faciliten el procés. No obstant això, també presenta desavantatges, com la seva dependència de Python, limitant la portabilitat. Per aquest motiu, el grup de recerca CloudLab de la URV, ha adaptat el model perquè es pugui utilitzar amb TorchScript, una biblioteca escrita en C++ que facilita la portabilitat, millora el rendiment i elimina les dependències amb Python, permeten la compatibilitat amb altres llenguatges de programació.

3.3 Python

Com a llenguatge de programació s'ha decidit utilitzar Python perquè el framework de Ray és "Python-first", això vol dir que s'ha dissenyat i optimitzat específicament per fer servir aquest llenguatge.

Figura 21. Logotip Python

3.4 Kubernetes

Per la creació d'un clúster s'utilitzarà Kubernetes (també conegut com a K8s) [26] perquè Ray proporciona un operador compatible anomenat KubeRay que facilita l'administració i desplegament de clústers amb Ray.

Kubernetes és una eina que permet l'automatització, desplegament, escalat i gestió d'aplicacions en contenidors. A més, també permet coordinar eficientment un conjunt d'ordinadors que estan connectats per treballar com si fos un. La seva arquitectura està formada per dos components:

- **Control plane:** és el responsable de supervisar i gestionar el clúster.
- **Worker nodes:** pot ser una màquina física o virtual (depenent del clúster). Cada node és gestionat pel control plane i conté els serveis necessaris per executar Pods.

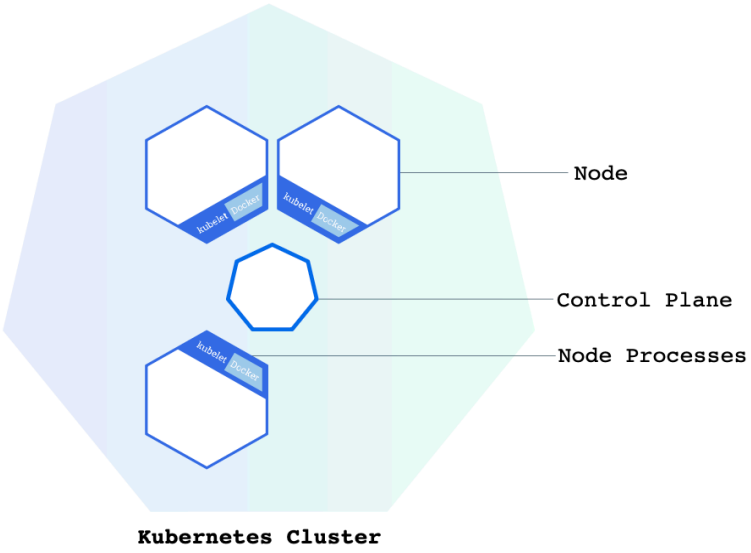


Figura 22. Representació d'un clúster de Kubernetes

A més dels components principals, hi ha altres conceptes clau:

- **Pods:** són les unitats més petites en Kubernetes, contenen les aplicacions i els seus contenidors. Un Pod pot contenir un o més contenidors, compartint el mateix entorn i recursos, com la xarxa i l'emmagatzematge.

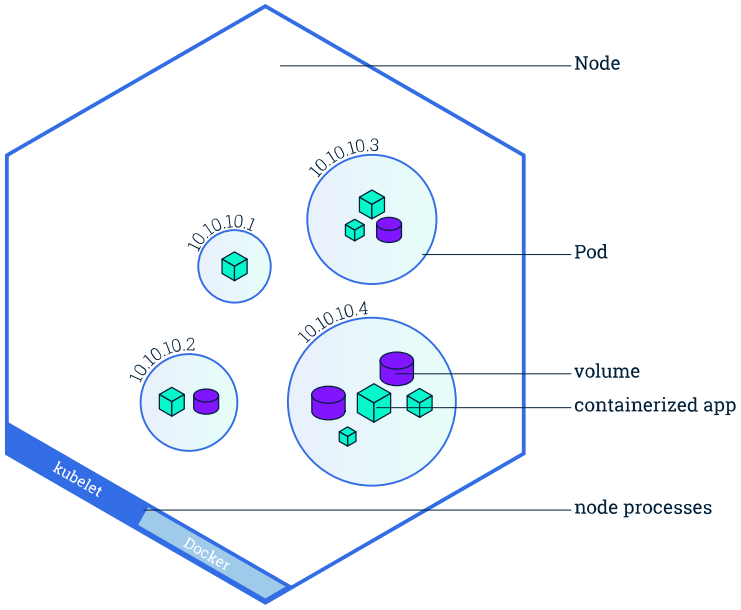


Figura 23. Representació d'un node amb quatre Pods

- **Serveis:** permeten la comunicació entre diferents aplicacions o components dins del clúster. Proporcionen una forma d'accedir a les aplicacions mitjançant una única adreça IP, independentment de la seva ubicació o nombre de rèpliques. Això facilita la connectivitat i l'escalabilitat de les aplicacions distribuïdes.

3.5 MinIO

Per portar a terme les proves amb un sistema de fitxers alternatiu al local, s'ha decidit utilitzar MinIO [27], que serà útil quan es duguin a terme proves en un clúster, ja que serà accessible des de tots els nodes.

MinIO és una solució d'emmagatzematge d'objectes compatible amb les característiques principals de l'API d'Amazon S3. Utilitza buckets per emmagatzemar els objectes. Els buckets es poden comparar amb les carpetes d'un sistema de fitxers convencional. Aquests buckets són l'espai on es guardaran les imatges que utilitzarem.

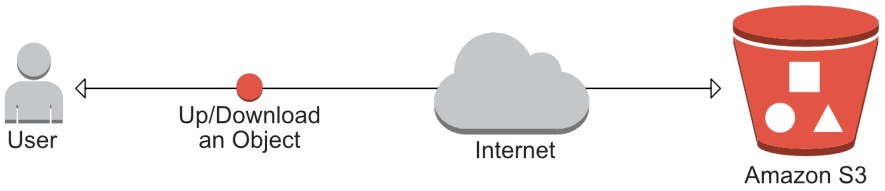


Figura 24. Representació interacció amb un Bucket

MinIO permet diverses topologies per a organitzar els seus nodes i els seus discos:

- **Single-Node Single-Drive (SNSD):** En aquesta configuració, només hi ha un node i un sol disc. És adequada per a casos d'ús senzills i petits, com a proves de concepte o desenvolupament local.
- **Single-Node Multi-Drive (SNMD):** En aquesta configuració, encara només hi ha un node, però amb diversos discos connectats. És útil per a augmentar l'emmagatzematge i la redundància en un entorn senzill.
- **Multi-Node Multi-Drive (MNMD):** Aquesta és una configuració distribuïda amb múltiples nodes i múltiples discos. És l'opció més escalable i resilient, ja que permet una distribució de càrrega i redundància entre múltiples nodes i discos. És ideal per a entorns empresarials i empreses amb necessitats d'emmagatzematge importants.

En aquest treball, es farà servir la configuració SNSD per fer les proves en local i MNMD en el cas de l'execució al clúster. La decisió d'emprar MinIO es deu a la compatibilitat per defecte de Ray amb l'API de S3, perquè en cas de ser necessari, el codi implementat seria fàcilment modificable per fer la transició als servidors d'Amazon Web Services.

3.6 Rack URV

Per dur a terme els experiments amb Ray s'utilitzarà el rack de la URV [28]. Aquest rack forma un clúster amb dos grups diferents:

- **Grup de computació**

  Aquest grup està format per 9 nodes (compute nodes), tots ells amb Docker i Kubernetes configurats amb la següent distribució:

  - Hi ha 1 node coordinador (conegut com a K8s control plane) que s'encarrega de gestionar i coordinar el funcionament del clúster.

El coordinador disposa de 6 Cores amb 12 Threads, 32 GB de RAM i 200 GB de disc.

- Hi ha 8 nodes de computació (coneguts com a K8s worker nodes) que s'encarreguen d'executar les aplicacions als contenidors.

  Entre tots els nodes de computació es proporciona una capacitat de càlcul de 228 Cores amb 420 Threads, 564 GB de RAM i 2440 GB de disc.

- **Grup d'emmagatzematge**

  Aquest grup està format per 6 nodes (storage nodes), cadascun amb 453 GB que se subdivideixen en:

  - Hi ha 3 nodes que actuen com un servidor de MinIO.
  - Hi ha 3 nodes que actualment no s'utilitzen.

En la següent figura es representa gràficament el clúster.

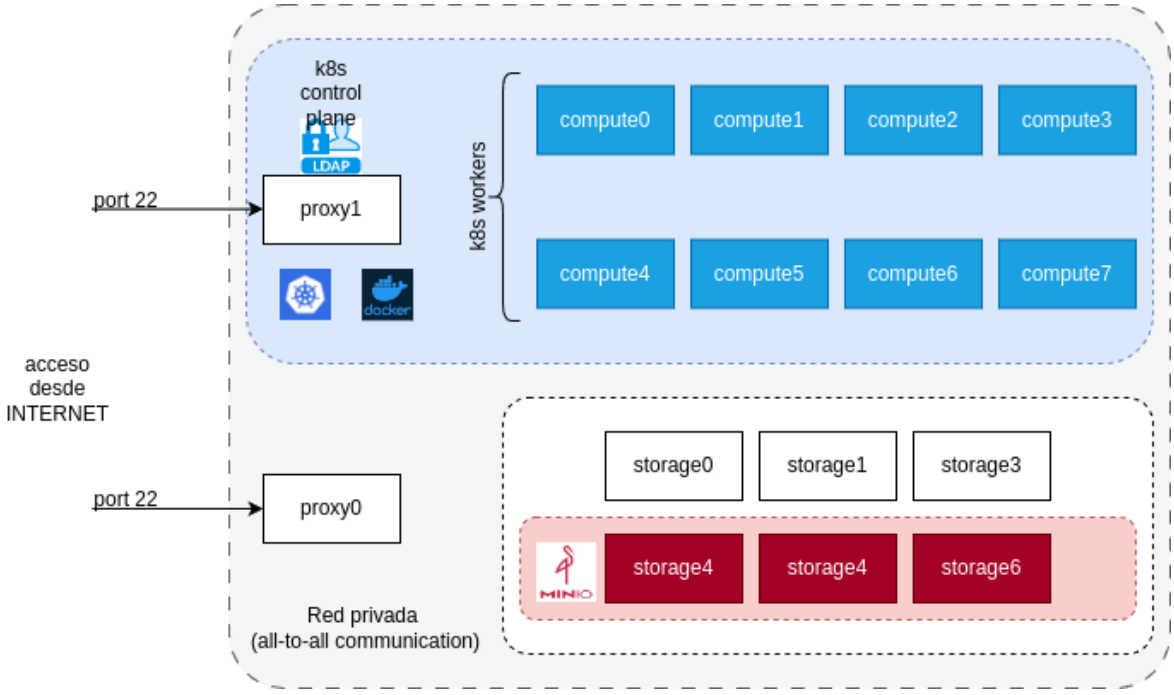


Figura 25. Representació gràfica del clúster

4 Fases del desenvolupament

Després d'haver portat a terme la recerca d'informació documentada a la secció anterior, s'ha iniciat el desenvolupament amb Ray. El procediment seguit ha consistit inicialment en el desplegament de Ray en un entorn local amb un únic node que ha permès familiaritzar-se amb el framework i comprendre amb més detall les seves característiques i el seu funcionament, més tard es va desplegar un clúster local fent ús de Kubernetes amb la possibilitat de tenir múltiples nodes per distribuir la càrrega. Finalment, es va escalar al rack de la universitat, una màquina amb capacitats molt superiors que permet simular un entorn real de treball i és on s'executaran els experiments finals.

4.1 Ray en local

Per començar, es van explorar els exemples oficials de Ray per realitzar la inferència per lots fora de línia. Utilitzant aquests exemples es va desenvolupar la càrrega i el preprocessament de les imatges, utilitzant imatges d'ImageNet. Primer es va provar d'utilitzar una font de dades local i més tard es va adaptar per obtenir-les des de MinIO que permet simular un emmagatzematge al núvol. A continuació es va adaptar el codi per realitzar la inferència amb el model ResNet50 utilitzant els pesos predeterminats.

Un cop es va obtenir un codi funcional, es van fixar els requisits del projecte per evitar incompatibilitats entre paquets i sobretot, pel fet que Ray és un framework molt recent que s'actualitza de versió múltiples vegades cada mes, la qual cosa, pot implicar canvis a la seva API causant la incompatibilitat del codi desenvolupat. En la següent taula es detallen les versions dels paquets utilitzats i perquè serveixen.

Paquet	**Versió**	**Descripció**
ray[default, data]	2.9.3	Framework base i llibreria data enfocada al ML.
pandas	1.5.2	Llibreria d'anàlisi i manipulació de dades.
tqdm	4.65.0	Paquet que permet la visualització de la barra de progrés de Ray.
s3fs	2023.10.0	Llibreria que permet connectar-se a S3.
pyarrow	13.0.0	Plataforma de desenvolupament per a dades en memòria, centrada en el processament de grans dades.
numpy	1.26.4	Paquet que proporciona suport per a matrius grans i multidimensionals.
torch	2.1.2	Llibreria per treballar amb tensors optimitzada per a l'aprenentatge profund mitjançant CPU i/o GPU.
torchvision	0.16.2	Llibreria per a PyTorch, que conté conjunts de dades populars, arquitectures de models i transformacions d'imatges.

Taula 1. Requisits del projecte

```
--extra-index-url https://download.pytorch.org/whl/cpu
ray[default]==2.9.3
ray[data]==2.9.3
pandas==1.5.2
tqdm==4.65.0
s3fs==2023.10.0
pyarrow==13.0.0
numpy==1.26.4
torch==2.1.2
torchvision==0.16.2
```

Codi 1. Fitxer requirements.txt

Cal destacar que Torch i TorchVison s'han utilitzat amb la versió enfocada a utilitzar-se en CPUs, ja que es pretén estudiar el comportament de la inferència amb Ray sense la utilització de GPUs, ja que no es disposen d'elles.

Més tard aquest codi es va adaptar per processar imatges del Laboratori Europeu de Biologia Molecular utilitzant el model preentrenat "Off-sample". Aquest canvi ha implicat la modificació de la funció de preprocessament i de la classe de l'actor que realitza la inferència per adaptar-se als requisits d'aquest model.

El desenvolupament d'aquests codis ha estat fonamental per identificar els paràmetres configurables de cada funció. Mitjançant aquests paràmetres, s'ha creat un nou codi que realitza una cerca en graella[17] que permet automatitzar l'execució del codi amb múltiples combinacions de paràmetres per trobar-ne els òptims. En cada execució es recopila en un fitxer CSV els paràmetres de configuració i les mètriques de rendiment del codi. Addicionalment, també es permet configurar el nombre de repeticions de cada configuració per calcular la mitjana dels resultants, assegurant-ne una mostra major que proporciona una major fiabilitat dels resultats.

4.2 Ray clúster en local

En aquest moment, es va decidir provar Ray amb múltiples nodes, per això es va instal·lar Kubernets localment en un portàtil amb 4 CPUs cadascuna amb dos fils d'execució i 16 GB de RAM.

Per desplegar el clúster ha sigut necessari instal·lar l'operador KuberRay i seleccionar una imatge predefinida de docker amb Ray 2.9.3 i Python 3.11, concretament s'ha seleccionat la imatge rayproject/ray:2.9.3-py311 [29]. A continuació s'ha investigat com crear el fitxer de configuració del clúster. Aquest fitxer es troba en format YAML, que és un format de serialització de dades generalment utilitzat per les configuracions. En el cas de KubeRay permet definir els següents paràmetres:

- La versió de Ray que ha de coincidir amb la inclosa en la imatge de Docker.
- La configuració del head node i dels workers nodes. En els dos es pot indicar la imatge de docker a utilitzar i els requisits/límits de recursos de CPU i RAM. En el cas del head node a més, es pot configurar si també s'encarrega d'executar tasques i actors o tan sols s'encarrega d'executar el procés de control general. En el cas dels workers es pot configurar el nombre de rèpliques desitjades, i si està l'autoescalador habilitat s'han de definir el mínim i el màxim de rèpliques.

[17] Una cerca en graella o grid search és una tècnica utilitza per trobar la millor combinació de paràmetres.

- La utilització o no de l'autoescalador i els requisits/límits de recursos de CPU i RAM.
- Els ports exposats dels serveis del head node que es volen exposar per exemple el dashboard.

S'han creat dos fitxers. El primer fitxer defineix un clúster amb 1 head node i 2 worker nodes cadascun amb 2 CPUs i 3 GB de RAM. El segon, integra l'autoescalador juntament amb Prometheus i Grafana per poder recopilar i visualitzar tota la informació del dashboard.

Un cop creat el clúster, que es troba en l'espai de Kubernetes (aïllat respecte a la resta de l'ordinador) és necessari connectar-se d'alguna forma per executar el codi. Això es pot aconseguir connectant-se per SSH o exposant amb algun dels ports que utilitza Ray amb port forwarding, el número del port depèn del mètode desitjat. Les formes de fer-lo són les següents:

- Executar el codi directament en el clúster fent una connexió SSH al head node, aquesta opció es va descartar perquè ens interessa un desenvolupament més interactiu.
- Utilitzar Ray Client, que permet connectar-se a un clúster remot mitjançant el port 10001. Requereix que el clúster i l'entorn local utilitzin la mateixa versió de Ray i Python, un cop es va configurar tot, a l'hora d'executar el codi va aparèixer l'error *"Global node is not initialized."*, investigant pels fòrums de Ray [30] es va descobrir que el client de Ray no és directament compatible amb els fluxos de treball de ML com potser Ray Data, es proposava una solució que consistia a encapsular el codi dins d'una tasca, però es va descartar la seva utilització.
- Utilitzar Ray Jobs, que mitjançant una comanda en el terminal permet encuar un Job al clúster per la seva execució utilitzant el port 8265. Aquesta opció és finalment la seleccionada.

Per utilitzar Ray Jobs és necessari el reenviament del port 8265 des del clúster de kubernetes cap a la màquina local, ja que és el port utilitzat per encuar els treballs i que coincideix amb el port que es fa servir per a dashboard permeten alhora el seu accés des de localhost:8265. Al començament es va provar de declarar en la comanda del job els requisits d'entorn, a continuació es pot veure un exemple:

```
kubectl port-forward --address 0.0.0.0 svc/raycluster-head-svc 8265:8265

ray job submit --address http://localhost:8265 --runtime-env-json='{"pip": [
        "pandas==1.5.2",
        "tqdm==4.65.0",
        "s3fs==2023.10.0",
        "pyarrow==13.0.0",
        "numpy==1.26.4",
        "torch==2.1.2",
        "torchvision==0.16.2"
    ]}' --working-dir . -- python  file_name
```

Codi 2. Comanda port forwarding i Ray Job Submit

Durant les proves es va detectar que la instal·lació de tots els requisits a cada node era molt lenta, aquesta opció és més adient per requisits puntuals i execucions esporàdiques. Com no és el nostre cas, ja que volem fer proves amb codis que comparteixen els mateixos requisits s'ha creat una imatge personalitzada de Docker nova partint de la imatge oficial utilitzada anteriorment i afegint:

- Els requisits de l'aplicació descrits anteriorment.
- El model Off-sample per realitzar la inferència. S'inclou pel fet que és més eficient i ràpid que estigui emmagatzemant en cada node en lloc de descarregar-lo en cada execució des de MinIO.

La imatge creada es troba disponible en el següent repositori públic de Docker Hub.

- javierqt26/ray_py311_requirements [31]

S'ha creat a partir d'aquest Dockerfile:

```
FROM rayproject/ray:2.9.3-py311

RUN pip install pandas==1.5.2 \
tqdm==4.65.0 \
s3fs==2023.10.0 \
pyarrow==13.0.0 \
numpy==1.26.4

ARG BINS_DIR="/bin"
RUN mkdir -p ${BINS_DIR}
COPY torchscript_model_2.1.2.pt ${BINS_DIR}/model.pt

RUN pip install torch==2.1.2 torchvision==0.16.2 --index-url
https://download.pytorch.org/whl/cpu
```

Codi 3. Dockerfile creació imatge clúster

A continuació es pot veure un exemple de fitxer YAML per la configuració del clúster:

```
apiVersion: ray.io/v1
kind: RayCluster
metadata:
  name: raycluster-mini
spec:
  rayVersion: '2.9.3'
  headGroupSpec:
    rayStartParams:
      dashboard-host: '0.0.0.0'
    template:
      spec:
        containers:
        - name: ray-head
          image: javierqt26/ray_py311_requirements
          ports:
          - containerPort: 6379
            name: gcs
          - containerPort: 8265
            name: dashboard
          - containerPort: 10001
            name: client
          lifecycle:
            preStop:
              exec:
                command: ["/bin/sh","-c","ray stop"]
          resources:
            limits:
              cpu: "2"
              memory: "4G"
            requests:
              cpu: "2"
              memory: "4G"
  workerGroupSpecs:
  - replicas: 2
    groupName: ray-worker
    rayStartParams: {}
    template:
      spec:
        containers:
        - name: ray-worker
          image: javierqt26/ray_py311_requirements
          lifecycle:
```

```
            preStop:
              exec:
                command: ["/bin/sh","-c","ray stop"]
          resources:
            limits:
              cpu: "2"
              memory: "4G"
            requests:
              cpu: "2"
              memory: "4G"
```

Codi 4. YAML configuració clúster

Un cop configurat el clúster es va dur a terme diverses proves per comprovar el seu correcte funcionament, donat que alhora s'executava Kubernetes amb el clúster (format per un head node i dos workers) i un procés amb el servidor MinIO van aparèixer múltiples errors quan s'executava el codi. Entre ells, els dos més freqüents eren la finalització inesperada d'algun node i la falta de memòria *(out of memory error)* [32], tot provocat per la falta de recursos de l'ordinador. En la majoria dels casos provocava la interrupció de l'execució del codi i en algun cas més concret la destrucció del head node sense possibilitat d'una autoregeneració del clúster.

A pesar de tots aquests problemes, s'ha aconseguit el propòsit de familiaritzar-se amb Kubernetes creant un clúster amb una imatge de Docker personalitzada, desenvolupar un codi funcional i aconseguir executar-lo.

4.3 Ray clúster amb el rack del CloudLab

Finalment, en aquesta fase es va replicar el clúster de KubeRay al rack del CloudLab. Primer, es va configurar l'ordinador per poder connectar-se al rack. Per a això, es va utilitzar l'aplicació Termius per establir una connexió SSH amb el proxy1, que és la porta d'entrada al rack. Aquesta aplicació també facilita el pas de fitxers i la creació dels ports forwarding necessaris per accedir als serveis que ho requereixen des de l'ordinador, com el dashboard de Ray i el de MinIO.

A més, s'ha utilitzat Lens, un IDE que permet monitorear i administrar visualment l'estat de Kubernetes, una tasca que fins ara s'havia realitzat a través del terminal. També es van crear variacions dels fitxers de configuració per assignar recursos similars als que es podrien assignar en el desplegament d'un clúster per a una aplicació real.

Per a la creació del clúster, s'han aprofitat els nodes de computació que tenen Kubernetes preinstal·lat, aplicant l'operador KubeRay per desplegar el clúster de Ray. Pel que fa a l'emmagatzematge compartit, s'han utilitzat els nodes amb un servidor MinIO que ja estava configurat. Només ha estat necessari crear un nou bucket per emmagatzemar les mètriques recopilades.

En la següent imatge es poden visualitzar els Pods d'un clúster amb Ray desplegat:

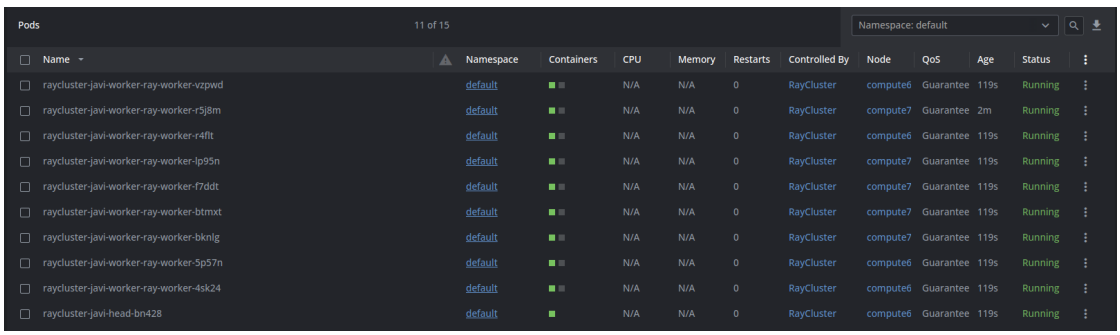


Figura 26. Captura Pods clúster Ray (Lens IDE)

5 Disseny del codi

En aquesta secció, es detalla l'estructura principal del codi desenvolupat. Com s'ha comentat anteriorment, es va crear un codi que utilitza imatges de la base de dades ImageNet juntament amb el model ResNet50. Posteriorment, es va adaptar i millorar aquest codi per processar imatges del Laboratori Europeu de Biologia Molecular (EMBL) utilitzant el model Off-sample. També es va modificar per incorporar una cerca en graella (grid search) i finalment, es va adaptar per ser executat en el clúster.

5.1 Codi batch inference offline amb el model ResNet50

Primer, s'importen els paquets necessaris per a l'execució del codi:

```
import os
import ray
import torch
from torchvision import transforms
from torchvision.models import resnet50, ResNet50_Weights
import time
import s3fs
```

Codi 5. Importacions de paquets

Es declara el 'BATCH_SIZE', s'inicialitza Ray i es carrega el model ResNet50 amb els pesos preentrenats. Aquest model s'emmagatzema amb ray.put() en l'object storage compartit i es guarda la seva referència.

```
DATASET_SIZE = 14
BATCH_SIZE = 2
cwd = os.getcwd()
ray.init()
model = resnet50(weights=ResNet50_Weights.DEFAULT)
model_ref = ray.put(model)
```

Codi 6. Inicialització Ray

A continuació es declara l'operació lazy read_imatges() que permet carregar les imatges des d'un directori local en un dataset. Les imatges es redimensionen a la mida de 224x224 píxels, que és comunament utilitzat pels models i es transformen al mode RGB, un tipus comunament utilitzat perquè totes les imatges tinguin tres canals (Red/Green/Blue).

```
image_directory = f"{cwd}/../../imagenet/ImageNet-Datasets-Downloader/imagenet_images"
image_files = [os.path.join(image_directory, file) for file in os.listdir(image_directory)]
image_files = image_files[:DATASET_SIZE]
ds = ray.data.read_images(paths=image_files, size=(224, 224), mode='RGB',
ray_remote_args={"num_cpus": 1})
```

Codi 7. Càrrega de fitxers locals

Alternativament, s'ha modificat la càrrega d'imatges perquè es pugui realitzar des d'un bucket de MinIO.

```
fs = s3fs.S3FileSystem(anon=False, key='minioadmin', secret='minioadmin',
client_kwargs={'endpoint_url': 'http://localhost:9000'})
ds = ray.data.read_images(filesystem=fs, paths="s3://imagenet/imagenet_images/", size=(224,
224), mode='RGB', ray_remote_args={"num_cpus": 1})
```

Codi 8. Càrrega de fitxers S3

Respecte al preprocessament de les imatges s'ha creat la següent funció que Ray tracta com una tasca sense estat i s'encarrega d'aplicar el preprocessament al batch complet.

En primer lloc, es canvia l'ordre dels eixos de les imatges de (batch_size, height, width, channels) a (batch_size, channels, height, width) que és el format esperat per PyTorch. i es converteixen les imatges en un tensor que permet realitzar les transformacions més ràpidament.

A continuació s'aplica la seqüència de transformacions definida a preprocess seguint les especificacions [33] del model ResNet50 al tensor d'imatges i aquest es converteix a un array de NumPy.

```
def preprocess(image_batch):
    preprocess = transforms.Compose(
        [
            transforms.Resize(256, antialias=None),
            transforms.CenterCrop(224),
            transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
        ]
    )
    torch_tensor = torch.Tensor(image_batch["image"].transpose(0, 3, 1, 2))
    preprocessed_images = preprocess(torch_tensor).numpy()
    return {"image": preprocessed_images}
```

Codi 9. Funció preprocessament ResNet50

També es defineix la classe que s'encarrega de la inferència. Es defineix com una classe perquè Ray la tracta com un Actor amb estat. Hi ha dos mètodes

- __init__ en la inicialització de l'Actor, que tan sols es realitza una vegada, carrega el model des de l'object storage de Ray, s'estableix en el mode d'avaluació (desactivant certes característiques utilitzades exclusivament durant l'entrenament) i es mou a la CPU.
- __call__ permet invocar a l'actor. A partir d'un batch realitza la inferència utilitzant el model precarregat i retorna els resultats.

```
class Actor:
    def __init__(self, model):
        self.model = ray.get(model)
        self.model.eval()
        self.model.to("cpu")

    def __call__(self, batch):
        with torch.inference_mode():
            output = self.model(torch.as_tensor(batch["image"], device="cpu"))
            return {"class": output.cpu().numpy()}
```

Codi 10. Classe inferència ResNet50

Finalment, amb la funció de map_batches() es declaren sobre el dataset l'operació de preprocessament que es convertirà en una tasca de Ray i la d'inferència en un actor. Com map_batches() és del tipus lazy, tan sols acumula que fer.

```
ds = ds.map_batches(preprocess, batch_format="numpy", num_cpus=1)
ds = ds.map_batches(Actor, batch_format="numpy", batch_size=BATCH_SIZE, num_cpus=1,
concurrency=2, fn_constructor_kwargs={"model": model_ref})
```

Codi 11. Declaració paràmetres preprocessament i inferència

Un cop declarat que fer, cal utilitzar una funció de consum perquè s'executin les funcions lazy. Per això s'utilitza un bucle que amb la funció iter_batches() permet iterar sobretot tots els batchs del dataset.

```
for _ in ds.iter_batches(batch_format="numpy", batch_size=None):
    pass
```

Codi 12. Funció de consum que executa les operacions declarades

5.2 Codi batch inference offline amb el model EMBL

Les fases del codi són les mateixes que en l'anterior codi però adaptat per utilitzar el model de EMBL. En primer lloc, s'importen els paquets necessaris per a l'execució del codi:

```
import os
import ray
import torch
from torchvision import transforms
import time
from PIL import Image
import torch.nn.functional as F
import s3fs
```

Codi 13. Importació de paquets

A continuació es declara el DATASET_SIZE, el BATCH_SIZE i s'inicialitza Ray.

```
DATASET_SIZE = 14
BATCH_SIZE = 2
cwd = os.getcwd()
ray.init()
```

Codi 14. Inicialització de Ray

Després es defineix el directori de les imatges, es crea una llista amb tots els paths de les imatges i es limita en funció del dataset_size. Es declara l'operació lazy read_imatges() que carrega les imatges a partir de la llista de paths indicada. En aquest cas totes les imatges utilitzades tenen la mateixa mida i no cal indicar-la, es guarden els paths de les imatges per identificar-les, es transformen al mode RGB i es permet la càrrega de qualsevol classe d'imatge amb file_extensions=None, ja que per defecte tan sols s'accepten png, jpg, jpeg, tif, tiff, bmp i gif. Això és necessari perquè les imatges del EMBL s'obtenen a partir d'URLs sense extensió.

```
image_directory = f"{cwd}/../../embl/datasets/datasets_images/2024-01-08_11h01m57s/"
image_files = [os.path.join(image_directory, file) for file in os.listdir(image_directory)]
image_files = image_files[:DATASET_SIZE]
ds = ray.data.read_images(paths=image_files, include_paths=True, mode="RGB",
file_extensions=None, ray_remote_args={"num_cpus": 1})
```

Codi 15. Carregar imatges EMBL

Respecte al preprocessament de les imatges, com el model Off-sample originalment es va entrenar amb Fastai i aquesta biblioteca proporciona un conjunt de transformacions estàndards, s'ha investigat en el codi font en què consistien i s'han replicat. Concretament, les transformacions es defineixen a funció apply_tfms() [34]. Aquestes transformacions s'han adaptat a composed_transforms i s'apliquen al batch amb un bucle imatge a imatge.

```
def preprocess(image_batch):
    composed_transforms = transforms.Compose([
        transforms.ToTensor(),
        transforms.Lambda(lambda x: F.grid_sample(x.unsqueeze(0),
torch.nn.functional.affine_grid(torch.eye(2, 3, dtype=torch.float32).unsqueeze(0), [1, 3,
224, 224], True), mode='bilinear', padding_mode='reflection', align_corners=True)),
        transforms.Lambda(lambda x: x.squeeze(0)),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
    ])
    preprocessed_images = []
    for img in image_batch["image"]:
        img_pil = Image.fromarray(img)
        preprocessed_img = composed_transforms(img_pil)
        preprocessed_images.append(preprocessed_img.numpy())
    return {"image": preprocessed_images, "path": image_batch["path"]}
```

Codi 16. Funció preprocessament model Off-sample

Respecte a la inferència de les imatges

- __init__ carrega el model Off-sample des d'un directori local.
- __call__ permet invocar a l'actor. A partir d'un batch realitza la inferència i un postprocessament per extraure la probabilitat del tensor, afegint una etiqueta on/off i la seva probabilitat, juntament amb el path que identifica la imatge.

```
class Actor:
    def __init__(self):
        self.model =
torch.jit.load(f"{cwd}/../../embl/commons/torchscript_model_2.1.2.pt",torch.device('cpu'))
    def __call__(self, batch):
        inputs = torch.as_tensor(batch["image"], device="cpu")
        with torch.no_grad():
            output_batch = self.model.forward(inputs)
            predictions_batch = torch.softmax(output_batch, dim=1)
            pred_probs = predictions_batch.numpy()
            preds = pred_probs.argmax(axis=1)
            labels = []
            probabilities = []
            for i in range(len(pred_probs)):
                probabilities.append(pred_probs[i][0])
                if preds[i] == 0:
                    labels.append('off')
                else:
                    labels.append('on')
            return {"path": batch["path"], "label": labels, "prob": probabilities}
```

Codi 17. Classe inferència model EMBL

Finalment, amb la funció de map_batches() i iter_batches() es realitza el preprocessament i la inferència.

```
ds = ds.map_batches(preprocess, batch_format="numpy", num_cpus=1)
ds = ds.map_batches(Actor, batch_format="numpy", batch_size=BATCH_SIZE, num_cpus=1,
concurrency=2)

for _ in ds.iter_batches(batch_format="numpy", batch_size=None):
    pass
```

Codi 18. Execució preprocessament i inferència

5.3 Codi Grid search

El codi de grid search, és una evolució del codi anterior. Incorporant un bucle per explorar totes les combinacions d'un conjunt d'opcions indicades com a paràmetres de configuració de Ray. Aquesta tècnica permet examinar diferents combinacions de paràmetres per trobar la configuració més bona per la inferència per lots fora de línia. Per obtenir uns resultats més precisos i fiables, cada combinació es pot repetir n vegades en funció de l'especificat a n_repeticions i permet la configuració dels paràmetres que es veuen a continuació:

```
ds_name = '2024-01-08_11h01m57s'
n_repetitions = 3
param_grid_inference = {
    'dataset_size': [2954],
    'parallelism_read': [-1, 25, 50],
    'num_cpus': [1, 2],
    'batch_size_map_batches': [64, 128, 256],
    'concurrency': [1, 2, 3],
    'preserve_order': [False]
}
configurations = list(itertools.product(*param_grid_inference.values()))
```

Codi 19. Declaració paràmetres grid search

A més, s'ha creat dues funcions per emmagatzemar els resultats obtinguts en un fitxer CSV local o a MinIO. Aquest fitxer inclou els paràmetres de configuració utilitzats en cada execució, els resultats de temps d'execució i el troughput (imatges per segon). Facilitant l'anàlisi posterior i la comparació de les diferents configuracions.

```
def save_metrics_locally(metrics_to_csv, file_name):
    try:
        # Try to load existing CSV file and concatenate new metrics
        existing_df = pd.read_csv(file_name)
        new_df = pd.concat([existing_df, pd.DataFrame(metrics_to_csv, index=[0])])
        new_df.to_csv(file_name, index=False)
        print("Updated metrics have been saved to", file_name)
    except FileNotFoundError:
        # If the file doesn't exist, create a new one with the metrics
        df = pd.DataFrame(metrics_to_csv, index=[0])
        df.to_csv(file_name, index=False)
        print("A new file", file_name, "has been created with metrics")
```

Codi 20. Funció per emmagatzemar resultats localment

```
def save_metrics_to_s3(metrics_to_csv, file_name, s3_bucket, fs):
    try:
        # Try to load existing CSV file
        with fs.open(f"{s3_bucket}/{file_name}", 'rb') as f:
            existing_df = pd.read_csv(f)
        # Concatenate new metrics with existing DataFrame
        new_df = pd.concat([existing_df, pd.DataFrame(metrics_to_csv, index=[0])])
        # Save the updated DataFrame to the CSV file
        with fs.open(f"{s3_bucket}/{file_name}", 'wb') as f:
            new_df.to_csv(f, index=False)
        print("Updated metrics have been saved to", f"{s3_bucket}/{file_name}")
    except FileNotFoundError:
        # If the file doesn't exist, create a new one with the metrics
        df = pd.DataFrame(metrics_to_csv, index=[0])
        with fs.open(f"{s3_bucket}/{file_name}", 'wb') as f:
            df.to_csv(f, index=False)
        print("A new file", f"{s3_bucket}/{file_name}", "has been created with metrics")
```

Codi 21. Funció per emmagatzemar resultats a S3

La funció evaluate() encapsula la lògica d'inicialitzar Ray, carregar les imatges, realitzar el preprocessamet i la inferència a partir d'uns paràmetres i guardant les mètriques amb save_metrics_locally() o save_metrics_to_s3().

```
def evaluate(params):
    ds_name, num_cpus, batch_size_map_batches, concurrency, preserve_order = params

    ctx = ray.data.DataContext.get_current()
    ctx.execution_options.verbose_progress = True
    ctx.execution_options.preserve_order = preserve_order

    job_id = ray.get_runtime_context().get_job_id()

    start_time = time.time()

    ds = ray.data.read_images(filesystem=fs, paths=f"s3://tfg/{ds_name}/",
include_paths=True, mode='RGB', file_extensions=None, ray_remote_args={"num_cpus":
num_cpus})

    start_time_without_metadata_fetching = time.time()

    ds = ds.map_batches(preprocess, batch_format="numpy",
batch_size=batch_size_map_batches, num_cpus=num_cpus)
    ds = ds.map_batches(Actor, batch_format="numpy", batch_size=batch_size_map_batches,
num_cpus=num_cpus, concurrency=concurrency)

    for _ in ds.iter_batches(batch_format="numpy", batch_size=None):
        pass

    end_time = time.time()

    num_records = ds.count()
```

```
    total_time = end_time - start_time
    throughput = num_records / total_time
    total_time_without_metadata_fetching = end_time - start_time_without_metadata_fetching
    throughput_without_metadata_fetching = num_records /
total_time_without_metadata_fetching

    metrics = {
        "Job ID": job_id,
        "Dataset": ds_name,
        "Preserve order": preserve_order,
        "Num cpus": num_cpus,
        "Concurrency": concurrency,
        "Batch size map_batches()": batch_size_map_batches,
        "Dataset size (bytes)": ds.size_bytes(),
        "Num records dataset": num_records,
        "Num blocks dataset": ds.num_blocks(),
        "Total time": total_time,
        "Throughput (img/sec)": throughput,
        "Total time w/o metadata fetching": total_time_without_metadata_fetching,
        "Throughput w/o metadata fetching (img/sec)": throughput_without_metadata_fetching
    }
    return metrics
```

Codi 22. Funció per provar una configuració determinada

Finalment, es troba el bucle que prova totes les configuracions amb n_repeticions, calculant la mitjana i quan finalitza l'execució mostra la millor combinació de paràmetres en funció del throughput.

```
best_throughput = -1
best_params = None
for config in configurations:
    iteration_metrics = []
    job_ids = []
    sum_metrics = {
        "Total time": 0,
        "Throughput (img/sec)": 0,
        "Total time w/o metadata fetching": 0,
        "Throughput w/o metadata fetching (img/sec)": 0
    }

    for _ in range(n_repetitions):
        iteration_metrics = evaluate(config)

        job_ids.append(iteration_metrics["Job ID"])

        for key in sum_metrics.keys():
            if key in iteration_metrics:
                sum_metrics[key] += iteration_metrics[key]

    avg_metrics = iteration_metrics

    avg_metrics["Job ID"] = ', '.join(map(str, job_ids))

    avg_metrics.update({key: value / n_repetitions for key, value in sum_metrics.items()})

    save_metrics_to_s3(avg_metrics, "metrics_embl_inference_cluster_exp2.csv", "tfg", fs)
    if avg_metrics["Throughput (img/sec)"] > best_throughput:
        best_throughput = avg_metrics["Throughput (img/sec)"]
        best_params = config

print("Best throughput:", best_throughput)
print("Best parameters:", best_params)
```

Codi 23. Bucle grid search

Aquesta millora del codi permet una exploració exhaustiva de l'espai de paràmetres facilitant trobar la millor combinació per optimitzar el pipeline.

6 Disseny dels experiments

En primer lloc, s'han plantejat les següents qüestions a resoldre:

- Quins paràmetres de configuració tenen un impacte significatiu en el rendiment de les aplicacions?
- Com afecta la quantitat de recursos disponibles (CPUs, memòria i nodes) al rendiment de les aplicacions?
- La utilització d'un clúster amb autoescalador impacta en el rendiment de les aplicacions?

A continuació, s'han definit tres tipus de configuracions de nodes per fer els experiments:

Configuració	CPUs	RAM
1	2	3
2	3	5
3	6	10

Taula 2. Configuracions de recursos dels nodes

A partir d'aquestes preguntes i les configuracions de nodes s'han definit els següents experiments.

6.1 Experiment 1

Aquest experiment pretén respondre a la primera pregunta.

- Quins paràmetres de configuració tenen un impacte significatiu en el rendiment de les aplicacions?

Per abordar aquesta qüestió, s'ha decidit utilitzar un grid search, una tècnica que permet explorar exhaustivament un espai de paràmetres predefinit i identificar la millor configuració. Aquest mètode és especialment útil perquè prova totes les combinacions possibles dins del conjunt de paràmetres definit, facilita una comparació directa entre les diferents configuracions, i ajuda a identificar tant els paràmetres amb un major impacte en el rendiment com la millor combinació.

S'han definit tres configuracions de clústers diferents a partir dels nodes definits anteriorment on es realitzarà el grid search:

Configuració	CPUs	RAM	Nodes	Total CPUS	Total RAM
1	2	3	30	60	90
2	3	5	20	60	100
3	6	10	10	60	100

Taula 3. Configuracions clúster experiment 1

També s'ha definit un diccionari amb els paràmetres a provar pel grid search adaptat als recursos de cada clúster. Uns pel preprocessament:

```
# 2CPU 3RAM
param_grid_preprocess = {
    'ds_name': ['1kds'],
    'parallelism_read': [-1, 50, 100, 200, 300, 400],
    'num_cpus': [2],
    'batch_size_map_batches': [8, 16, 32, 64, 128],
    'preserve_order': [False]
}

# 3CPU 5RAM
param_grid_preprocess = {
    'ds_name': ['1kds', '10kds'],
    'parallelism_read': [-1, 50, 100, 200, 300, 400],
    'num_cpus': [3],
    'batch_size_map_batches': [8, 16, 32, 64, 128],
    'preserve_order': [False]
}

# 6CPU 10RAM
param_grid_preprocess = {
    'ds_name': ['1kds', '10kds'],
    'parallelism_read': [-1, 1, 50, 100, 200, 300, 400],
    'num_cpus': [6],
    'batch_size_map_batches': [8, 16, 32, 64, 128, 256],
    'preserve_order': [False]
}
```

Codi 24. Definició paràmetres de l'experiment 1 amb preprocessament

Uns altres per la inferència:

```
# 2CPU 3RAM
param_grid_inference = {
    'ds_name': ['1kds'],
    'parallelism_read': [-1, 50, 100, 200, 300, 400],
    'num_cpus': [2],
    'batch_size_map_batches': [8, 16, 32, 64, 128],
    'concurrency': [30],
    'preserve_order': [False]
}

# 3CPU 5RAM
param_grid_inference = {
    'ds_name': ['1kds', '10kds'],
    'parallelism_read': [-1, 50, 100, 200, 300, 400],
    'num_cpus': [3],
    'batch_size_map_batches': [8, 16, 32, 64, 128],
    'concurrency': [20],
    'preserve_order': [False]
}

# 6CPU 10RAM
param_grid_inference = {
    'ds_name': ['1kds', '10kds'],
    'parallelism_read': [-1, 1, 50, 100, 200, 300, 400],
    'num_cpus': [6],
    'batch_size_map_batches': [8, 16, 32, 64, 128, 256],
    'concurrency': [10],
    'preserve_order': [False]
}
```

Codi 25. Definició paràmetres de l'experiment 1 amb inferència

Per a l'execució, com tot el codi es troba dins d'un fitxer Python es pot encuar com un únic Job de Ray. Per facilitar l'execució de l'experiment s'ha creat el següent script de Bash que crea el clúster, espera que els Pods estiguin en estat "Running" i encua el treball:

```
#!/bin/bash

################################################################################
# Preprocess experiment
PYTHON_SCRIPT="1-exp1_ray_embl_preprocess_cluster_gs.py"

# Inference experiment
#PYTHON_SCRIPT="2-exp1_ray_embl_inference_cluster_gs.py"
################################################################################

################################################################################
# Cluster with 2 CPUs and 3 GB of RAM per node (1 head + 30 workers)
YAML_PATH="../../config/cluster_config/ray-cluster-2cpu-3ram.yaml"
replicas=30

# Cluster with 3 CPUs and 5 GB of RAM per node (1 head/worker + 19 workers)
#YAML_PATH="../../config/cluster_config/ray-cluster-3cpu-5ram.yaml"
#replicas=19

# Cluster with 6 CPUs and 10 GB of RAM per node (1 head/worker + 9 workers)
#YAML_PATH="../../config/cluster_config/ray-cluster-6cpu-10ram.yaml"
#replicas=9
################################################################################

wait_for_pods() {
  while true; do
    STATUS=$(kubectl get pods -l ray.io/is-ray-node=yes -o
jsonpath='{.items[*].status.phase}')
    if [[ "$STATUS" == *"Pending"* ]] || [[ "$STATUS" == *"ContainerCreating"* ]]; then
      echo "Waiting for the pods to be in Running state..."
      sleep 10
    else
      echo "All pods are in Running state."
      break
    fi
  done
}


# Create cluster
sed -i "s/replicas: [0-9]\+/replicas: $replicas/" $YAML_PATH
kubectl apply -f $YAML_PATH
kubectl delete pods -l ray.io/is-ray-node=yes
wait_for_pods

# Port forwarding
kubectl port-forward --address 0.0.0.0 svc/raycluster-javi-head-svc 8265:8265 &
PORT_FORWARD_PID=$!
sleep 10

# Job submit
ray job submit --address http://localhost:8265 --working-dir . -- python $PYTHON_SCRIPT

# Finish port forwarding
kill $PORT_FORWARD_PID
```

Codi 26. Script execució automatitzada experiment 1

6.2 Experiment 2

Aquest experiment pretén respondre a la segona pregunta.

- Com afecta la quantitat de recursos disponibles (CPUs, memòria i nodes) al rendiment de les aplicacions?

Per abordar aquesta qüestió s'executarà el codi de la inferència en clústers amb diferent nombre de nodes:

Configuració	CPUs	RAM	Nodes al clúster
1	2	3	1 a 60
2	3	5	1 a 40
3	6	10	1 a 20

Taula 4. Configuracions clúster de l'experiment 2

Com a l'experiment anterior, els paràmetres de l'execució també es troben adaptats als recursos de cada clúster. Donat que el nombre de nodes varia a cada execució, s'ha decidit deixar el paral·lelisme automàtic perquè s'adapti dinàmicament. La concurrència es defineix en funció del nombre de nodes de l'execució en curs.

```
# Cluster with 2 CPUs and 3 GB of RAM per node
param_grid_inference = {
    'ds_name': ['1kds'],
    'num_cpus': [2],
    'batch_size_map_batches': [8],
    'concurrency': [int(sys.argv[1])],
    'preserve_order': [False]
}

# Cluster with 3 CPUs and 5 GB of RAM per node
param_grid_inference = {
    'ds_name': ['1kds', '10kds'],
    'num_cpus': [3],
    'batch_size_map_batches': [8],
    'concurrency': [int(sys.argv[1])],
    'preserve_order': [False]
}

# Cluster with 6 CPUs and 10 GB of RAM per node
param_grid_inference = {
    'ds_name': ['1kds', '10kds'],
    'num_cpus': [6],
    'batch_size_map_batches': [8],
    'concurrency': [int(sys.argv[1])],
    'preserve_order': [False]
}
```

Codi 27. Definició paràmetres de l'experiment 2

En aquest cas també s'ha creat un script per a l'execució. Aquest script automatitza el procés de modificar-ne el fitxer YAML per la configuració del clúster, l'aplica i esperar que es redimensioni, és a dir que tots els Pods de Kubernetes estiguin en estat "Running", a més modifica els paràmetres del codi de Python en funció del clúster actual i encua el treball. A continuació es pot veure l'script.

```
#!/bin/bash

PYTHON_SCRIPT="1-exp2_ray_embl_inference_cluster_gs.py"

##############################################################################
# Cluster with 2 CPUs and 3 GB of RAM per node
YAML_PATH="../../config/cluster_config/ray-cluster-2cpu-3ram.yaml"
min_replicas=1
max_replicas=60
concurrency=$min_replicas

# Cluster with 3 CPUs and 5 GB of RAM per node
#YAML_PATH="../../config/cluster_config/ray-cluster-3cpu-5ram.yaml"
#min_replicas=0
#max_replicas=39
#concurrency=$((min_replicas + 1))


# Cluster with 6 CPUs and 10 GB of RAM per node
#YAML_PATH="../../config/cluster_config/ray-cluster-6cpu-10ram.yaml"
#min_replicas=0
#max_replicas=19
#concurrency=$((min_replicas + 1))
##############################################################################

wait_for_pods() {
  while true; do
    STATUS=$(kubectl get pods -l ray.io/is-ray-node=yes -o
jsonpath='{.items[*].status.phase}')
    if [[ "$STATUS" == *"Pending"* ]] || [[ "$STATUS" == *"ContainerCreating"* ]]; then
      echo "Waiting for the pods to be in Running state..."
      sleep 10
    else
      echo "All pods are in Running state."
      break
    fi
  done
}

# Create cluster
sed -i "s/replicas: [0-9]\+/replicas: $min_replicas/" $YAML_PATH
kubectl apply -f $YAML_PATH
kubectl delete pods -l ray.io/is-ray-node=yes
wait_for_pods

# Port forwarding
kubectl port-forward --address 0.0.0.0 svc/raycluster-javi-head-svc 8265:8265 &
PORT_FORWARD_PID=$!
sleep 10

for ((replicas=$min_replicas; replicas<max_replicas; replicas++))
do
  sed -i "s/replicas: [0-9]\+/replicas: $replicas/" $YAML_PATH
  kubectl apply -f $YAML_PATH
  wait_for_pods
  ray job submit --address http://localhost:8265 --working-dir . -- python $PYTHON_SCRIPT
"$concurrency"
  ((concurrency++))
done

# Finish port forwarding
kill $PORT_FORWARD_PID
```

Codi 28. Script execució automatitzada experiment 2

6.3 Experiment 3

Aquest experiment pretén respondre a la tercera pregunta.

- La utilització d'un clúster amb autoescalador impacta en el rendiment de les aplicacions?

Per resoldre aquesta qüestió s'estudiarà el comportament amb l'autoescalador i sense. Per aquest motiu s'han definit 3 configuracions de clústers per cada cas, que utilitzen el mateix tipus de nodes (6CPU i 10RAM).

	Clúster variable (amb autoescalador)			**Clúster fix**
Configuració	**Mínim nombre de nodes**	**Màxim nombre de nodes**	**Temps d'espera abans de desescalar**	**Nombre de nodes**
1	1	10	60s	10
2	1	20	60s	20
3	1	30	60s	20

Taula 5. Configuracions clústers de l'experiment 3

Ha estat necessari crear una variació del fitxer de configuració del clúster amb l'autoescalador habilitat utilitzant com a referència l'exemple oficial [35].

Es pretén executar el mateix codi en el clúster variable i el fix mesurant el rendiment i la latència. A més en el clúster variable es recopilarà el temps de creació dels Pods. Per facilitar aquest procés s'ha creat el següent programa amb Python que s'ha d'executar un cop redimensionat el clúster i guarda les mètriques dels Pods en un CSV.

```
import subprocess
import json
import csv

csv_file = "pods_times.csv"

pods = subprocess.run(["kubectl", "get", "pods", "-l=ray.io/is-ray-node=yes", "-o",
"json"], capture_output=True, text=True)
pods_json = json.loads(pods.stdout)

with open(csv_file, mode='w') as file:
    writer = csv.writer(file)
    writer.writerow(["Pod", "Initialized", "Ready"])

for pod in pods_json['items']:
    pod_name = pod['metadata']['name']
    print(pod)
    initialized_time = None
    ready_time = None

    for condition in pod['status']['conditions']:
        if condition['type'] == 'PodScheduled':
            initialized_time = condition['lastTransitionTime']
        elif condition['type'] == 'Ready':
            ready_time = condition['lastTransitionTime']

    with open(csv_file, mode='a') as file:
        writer = csv.writer(file)
        writer.writerow([pod_name, initialized_time, ready_time])

print("The data has been saved in", csv_file)
```

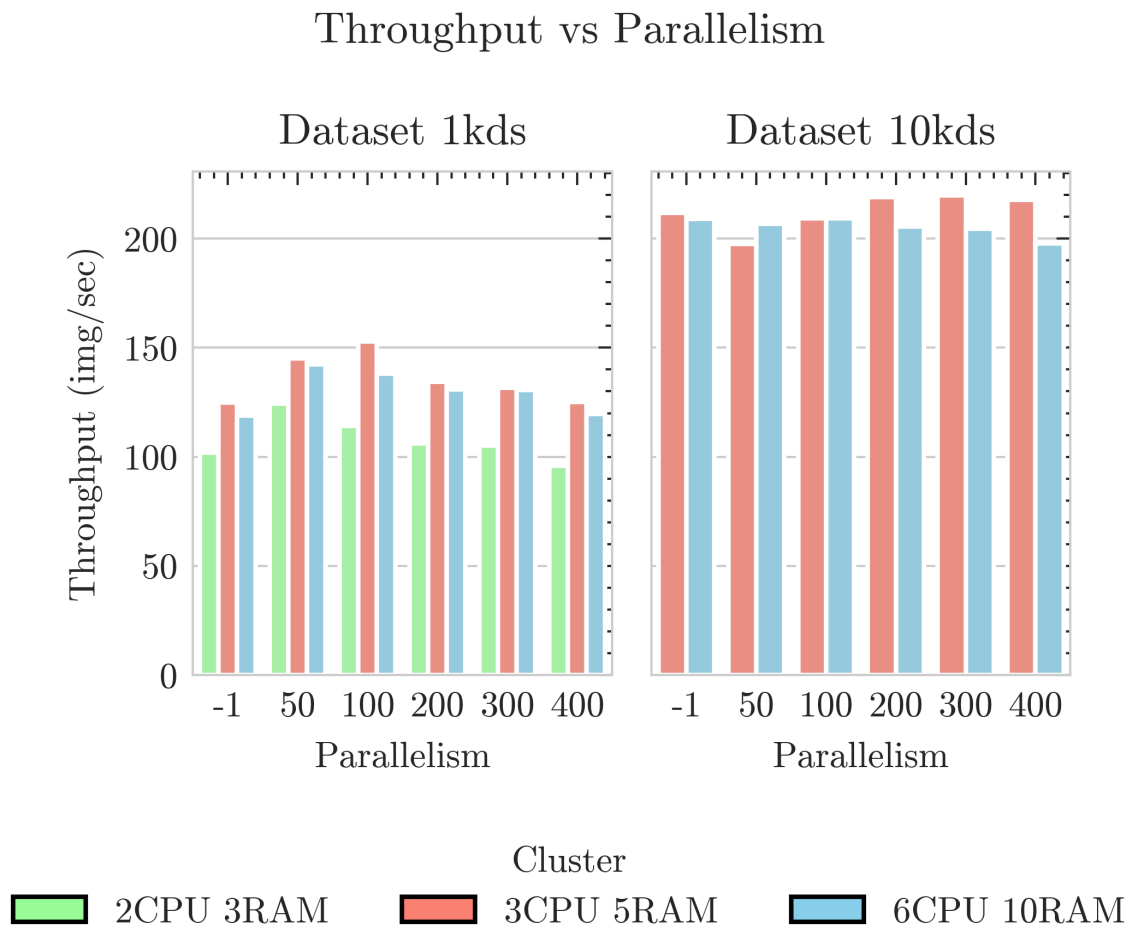
Codi 29. Codi per guardar el temps de creació dels Pods

7 Resultats dels experiments

7.1 Experiment 1

7.1.1 Resultats preprocessament

En primer lloc, s'ha volgut estudiar el comportament de modificar el paral·lelisme de la lectura d'imatges i veure com afecta en el rendiment del preprocessament. Els resultats obtinguts són els següents:



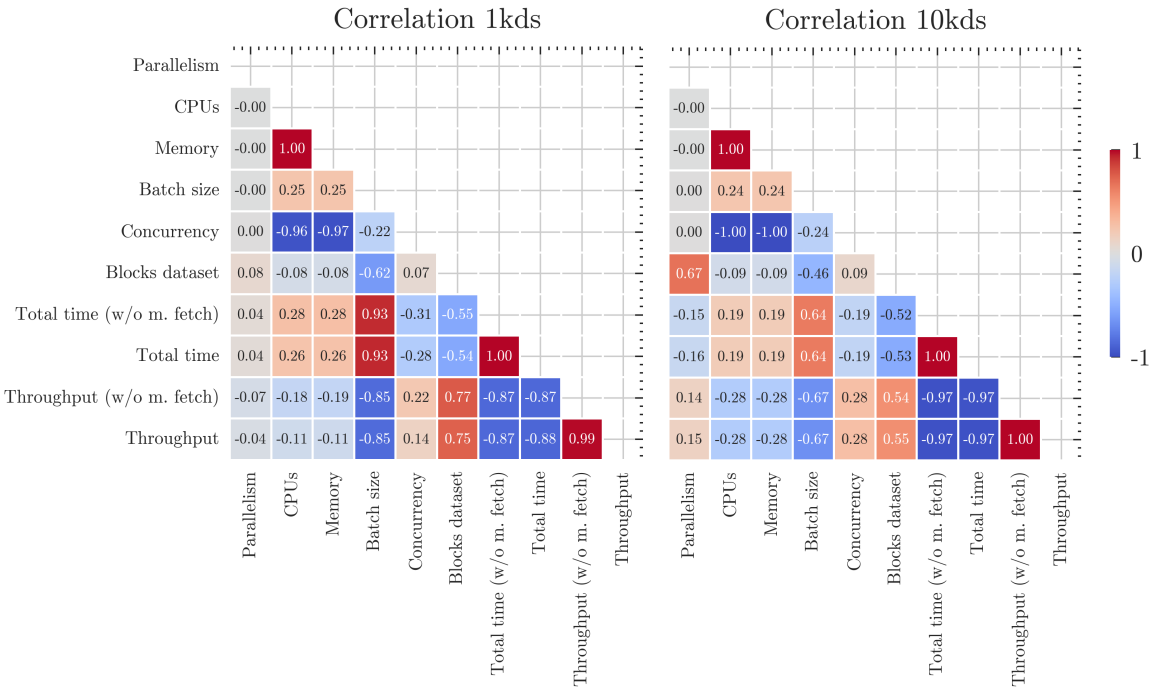
Gràfic 1. Rendiment vs Paral·lelisme (experiment 1)

En aquests gràfics de barres es pot veure que amb el dataset de 1.000 imatges, el millor paral·lelisme es troba entre 50 i 100. En canvi, amb el dataset de 10.000 el rendiment amb qualsevol configuració és molt major destacant lleugerament els paral·lelismes de 200 i 300. Per tant, **a mesura que augmenta la mida del dataset, seria convenient incrementar lleugerament el paral·lelisme**.

El paral·lelisme -1 és l'opció predeterminada que s'ajusta dinàmicament en funció de la quantitat d'imatges seguint l'heurística de Ray [21]. Es pot apreciar que ajustant-lo manualment es pot aconseguir una lleugera millora de rendiment.

7.1.2 Resultats inferència

A continuació s'ha representat un gràfic de correlació per determinar quins paràmetres afecten els resultats.

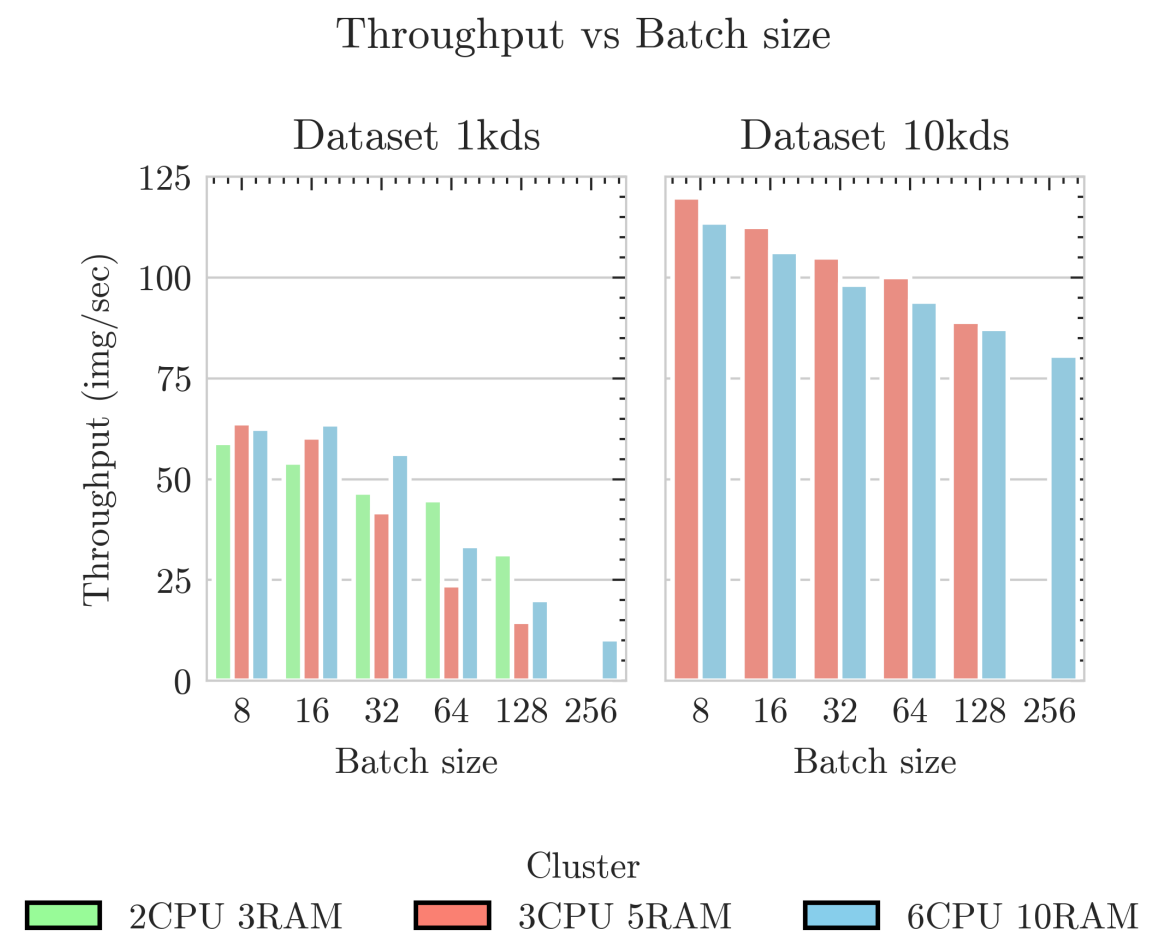


Gràfic 2. Correlacions (experiment 1)

Als dos gràfics de correlacions, un per 1.000 imatges i l'altre per 10.000, es poden apreciar que les tendències són similars. Es fan evidents aspectes com que a mesura que el rendiment augmenta, també disminueix el temps total, i que una major concurrència (que equival a un major nombre de nodes al clúster) implica un menor nombre de CPUs i menys memòria per node, d'acord amb les configuracions preestablertes pels clústers. A més, també es pot observar la relació entre el nombre de blocs del dataset i el batch size, a menys d'un, més de l'altre. Les tendències més rellevants són:

- El rendiment no està gaire correlacionat amb el paral·lelisme.
- A major paral·lelisme major nombre de blocs, tan sols en el cas del dataset gran. Això és així perquè Ray determina dinàmicament si és viable utilitzar el paral·lelisme sol·licitat, que en el cas del conjunt de 1.000 imatges, els paral·lelismes grans no són viables i Ray els ajusta automàticament.
- A menor batch size, major nombre de blocs, el temps total és menor i el rendiment augmenta.

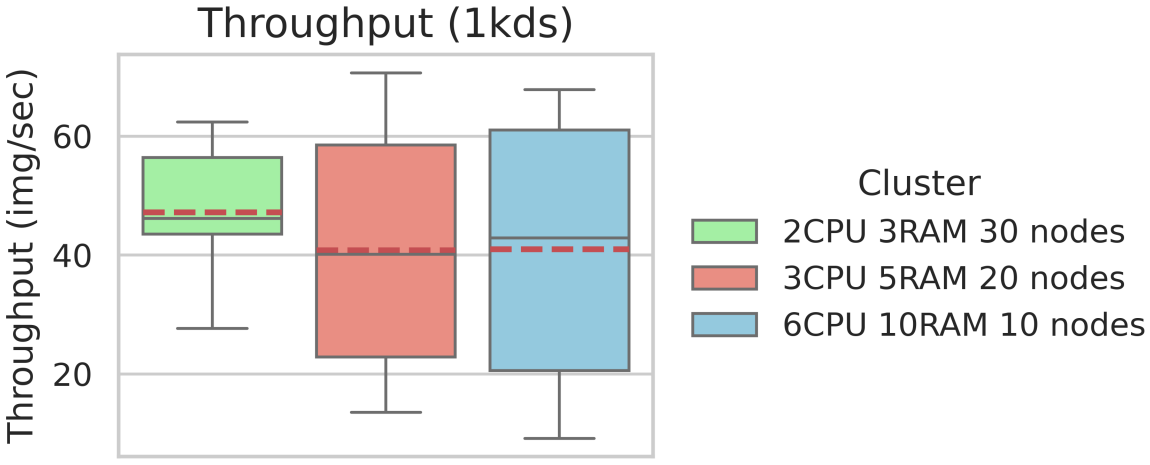
A continuació s'ha comparat el rendiment respecte al batch size.



Gràfic 3. Rendiment vs Batch size (experiment 1)

En aquests gràfics de barres es demostra l'observació anterior del gràfic de correlacions. Tant en el dataset de 1.000 com en el de 10.000 imatges, es pot observar que a mesura que la mida del batch augmenta, el rendiment empitjora. Per tant, és millor treballar una mida de batch petita.

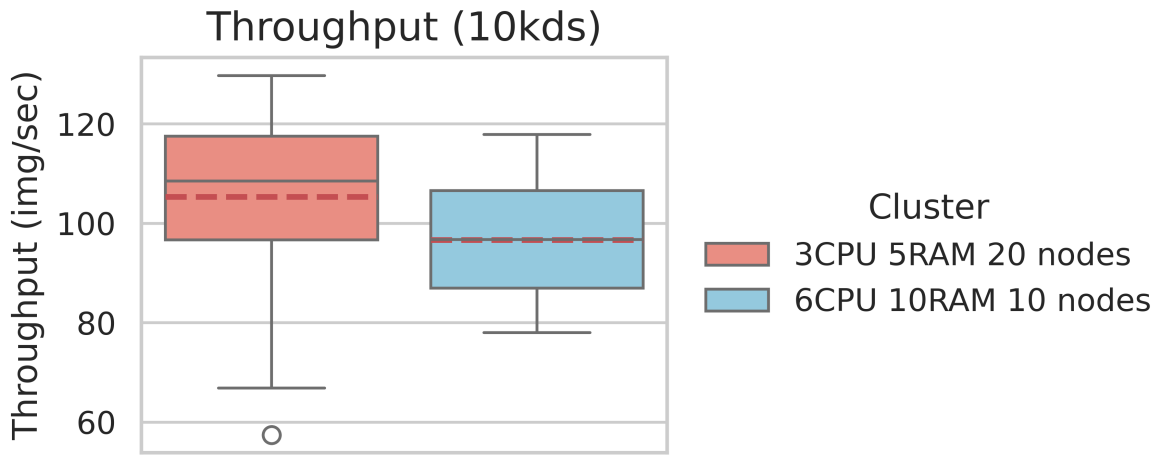
Finalment, s'ha volgut comparar el rendiment general de cada clúster.



Gràfic 4. Diagrama de caixes rendiment dataset de 1.000 imatges (experiment 1)

En aquest cas s'ha decidit fer la representació mitjançant un diagrama de caixes (box plot) perquè permet visualitzar ràpidament la distribució de les dades i compara-les entre els clústers. Amb el dataset de 1.000 imatges es pot veure que en la configuració:

- **2CPU 3RAM 30 nodes:** El rendiment va de 30 img/sec fins a 60 img/sec, amb una mitjana al voltant de 50 img/sec.
- **3CPU 5RAM 20 nodes:** El rendiment en aquesta configuració és més variable, amb un rang des de 15 img/sec fins a 65 img/sec, i una mitjana al voltant de 40 img/sec.
- **6CPU 10RAM 10 nodes:** El rendiment en aquesta configuració varia entre 10 img/sec i 65 img/sec, amb una mitjana també al voltant de 40 img/sec.



Gràfic 5. Diagrama de caixes rendiment dataset de 10.000 imatges (experiment 1)

Al box plot del dataset de 10.000 imatges es pot veure que en la configuració:

- **3CPU 5RAM 20 nodes:** El rendiment varia d'aproximadament 65 img/sec fins a 130 img/sec, amb una mitjana de 110 img/sec. Es veu que hi ha una major variabilitat, i s'observa un valor atípic (outlier) pel rang inferior.
- **6CPU 10RAM 10 nodes:** El rendiment en aquesta configuració varia entre 80 img/sec i 110 img/sec, amb una mitjana al voltant de 100 img/sec. Aquesta configuració mostra una menor variabilitat comparada amb el clúster de 3CPU 5RAM 20 nodes.

Comparant les dues gràfiques es pot apreciar que el rendiment de la inferència també ve directament determinat per la quantitat de càrrega de treball, a més imatges major throughput. També es pot veure que quan els recursos globals del clúster són iguals (és a dir, s'utilitzen el mateix nombre de CPUs i RAM distribuït en nodes amb diferents configuracions), **el clúster que ofereix un major pic de rendiment és el de 3CPU 5RAM 20 nodes**.

7.1.3 Resposta a la qüestió 1

Quins paràmetres de configuració tenen un impacte significatiu en el rendiment de les aplicacions?

Com s'ha vist en els resultats anteriors el principal factor que afecta el rendiment és la mida del batch que com menor sigui millor rendiment s'aconsegueix. A més, també cal buscar un equilibri en els recursos del clúster, com s'ha visualitzat al diagrama de caixes la configuració amb 3CPU/5RAM és la que millors resultats ha proporcionat.

7.1.4 Problemes durant l'execució

Durant l'execució d'aquest experiment s'ha detectat que incloure el head node com a worker node (és a dir, que executi codi) en el clúster amb 2 CPUs i 3 de RAM no és viable donats els limitats recursos d'aquesta configuració. La majoria són utilitzats per les tasques d'administració que realitza el head node, provocant que, quan s'intenta executar codi, s'esgotin fàcilment i apareguin errors de falta de memòria [32] que aturen l'execució del grid search. Per aquest motiu, s'ha hagut d'excloure el head node del grup de workers modificant el fitxer de configuració del clúster i per contrarestar això s'ha afegit un worker node addicional. A més, amb el clúster amb 2 CPUs i 3 GB de RAM no s'ha pogut provar el dataset amb 10.000 imatges perquè la majoria de configuracions provocaven també errors de falta de memòria.

També s'ha volgut provar amb paral·lelisme 1, però amb els clústers de 2CPU/3RAM i 3CPU/5RAM no ha estat possible per falta de memòria. El resultat obtingut amb el clúster de 6CPU/10RAM ha sigut en el millor dels casos 14 img/sec pel dataset de 1.000 imatges i 18 img/sec pel dataset de 10.000 imatges. Aquests resultats són molt més dolents que la resta i donat que tan sols es tenen en un clúster s'ha descartat utilitzar-los en les representacions.

Un altre problema detectat és que augmentar molt el paral·lelisme, concretament a partir de 400, provocava la saturació del head node en qualsevol de les configuracions, apareixent múltiples errors quan els workers s'intenten comunicar, però no obtenen resposta, això es coneix com a timeout.

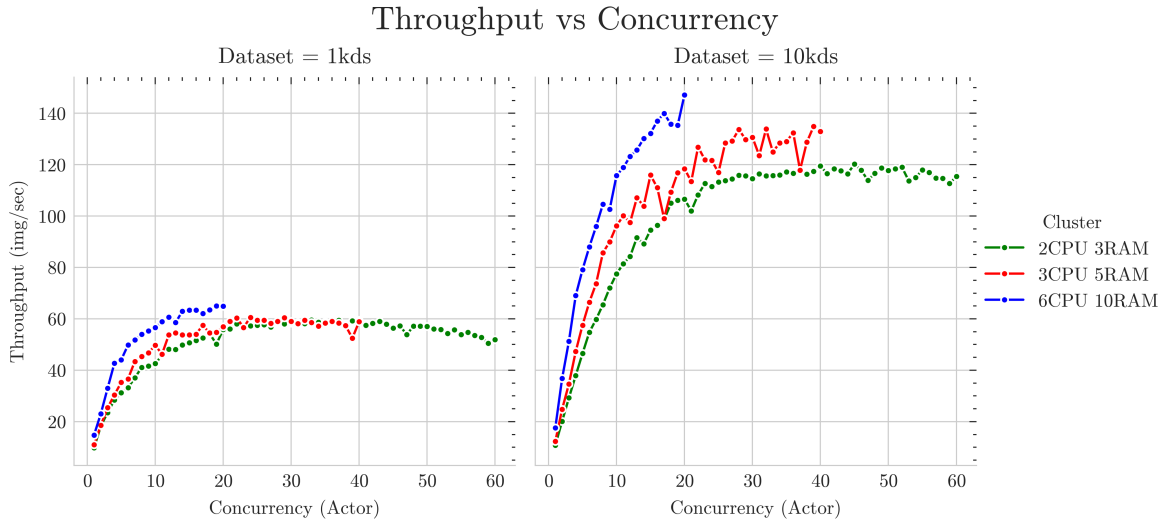
7.2 Experiment 2

7.2.1 Resultats

En aquest estudi s'ha analitzat el creixement del throughput en un clúster utilitzant tres configuracions de nodes diferents. Les configuracions de nodes segons els recursos són:

- 2 CPU i 3 GB de RAM Nodes amb menys recursos (color verd).
- 3 CPU i 5 GB de RAM Nodes intermedis (color vermell).
- 6 CPU i 10 GB de RAM Nodes amb més recursos (color blau).

S'ha escalat el nombre de nodes a cada clúster fins a assolir el mateix límit total de recursos i s'ha avaluat el rendiment amb dos datasets, un amb 1000 imatges i un altre amb 10.000 imatges. Els resultats obtinguts es presenten en els gràfics següents:



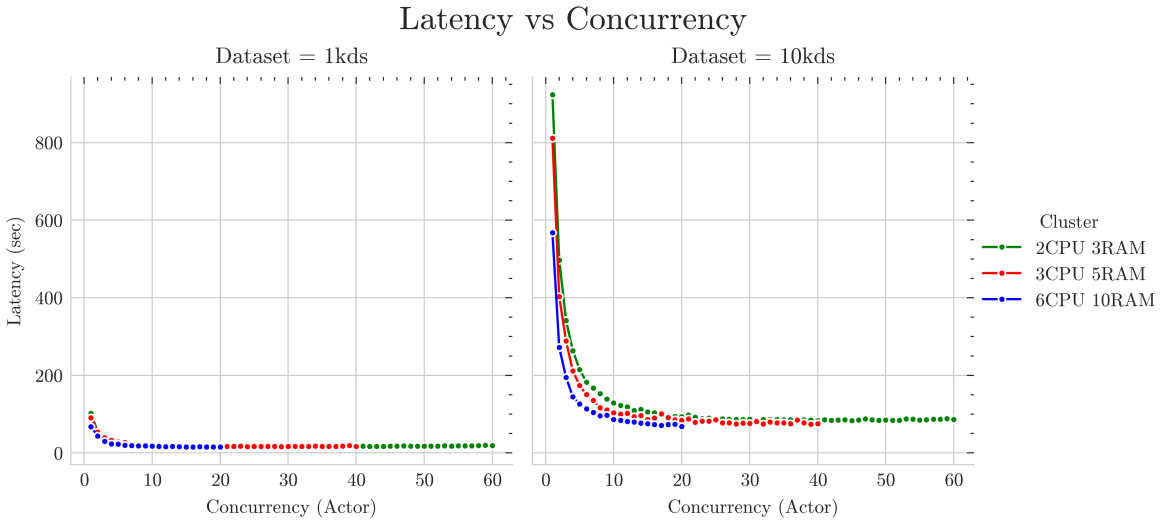
Gràfic 6. Rendiment vs Concurrència (experiment 2)

Al gràfic del dataset amb 1.000 imatges, es pot apreciar que, en general, augmentar el nombre de nodes incrementa el throughput. Inicialment, aquest creixement és lineal, duplicar el nombre de nodes gairebé duplica el throughput. Tot i això, a partir dels 4-5 nodes, el creixement es torna sublineal a causa de la sobrecàrrega de coordinació i comunicació entre els nodes. Finalment, al voltant dels 12-13 nodes, s'assoleix un punt de saturació on afegir més nodes no proporciona cap millora en el rendiment. Aquest comportament és visible als tres tipus de clústers.

Al gràfic del dataset amb 10.000 imatges, s'observa un fenomen similar. No obstant això, el punt de saturació s'assoleix amb aproximadament el doble de nodes als clústers amb nodes petits i mitjans. En el cas del clúster amb nodes grans, no s'arriba al punt on el throughput deixa de millorar dins del rang avaluat. Això indica que els nodes amb més recursos poden manejar una major càrrega abans que la sobrecàrrega de coordinació impacti significativament en el rendiment.

El comportament visible als dos gràfics es coneix com la **llei de rendiments decreixents** (Diminishing returns) [36]. Aquesta llei indica que **cada vegada s'obté un menor rendiment addicional encara que s'afegeixin més recursos**. Això es veu demostrat clarament als gràfics, on l'augment de nodes no es tradueix en una millora proporcional del rendiment, sinó que aquest augment és cada cop menor.

Comparant el rendiment entre els diferents clústers, es pot veure que està directament determinat pel nombre d'imatges del dataset. **A major nombre d'imatges, s'observa un throughput més gran, atès que la càrrega de treball és suficient per aprofitar els recursos addicionals**. Els nodes amb més recursos (clúster blau) sempre presenten un millor rendiment en comparació dels nodes menys potents (clúster verd i vermell), la qual cosa subratlla la importància d'una configuració de maquinari potent per a tasques intensives.



Gràfic 7. Latència vs Concurrència (experiment 2)

Inversament, s'observa que la latència disminueix a mesura que augmenta el nombre de nodes. Això és perquè la càrrega de treball es distribueix entre més nodes, reduint el temps d'espera per processar cada bloc. Igual que amb el throughput, la reducció de la latència és més significativa a les primeres etapes de l'escalament. A partir d'un cert punt afegir més nodes té una menor reducció de la latència, o inclús l'augmenta.

7.2.2 Resposta a la qüestió 2

Com afecta la quantitat de recursos disponibles (CPUs, memòria i nodes) al rendiment de les aplicacions?

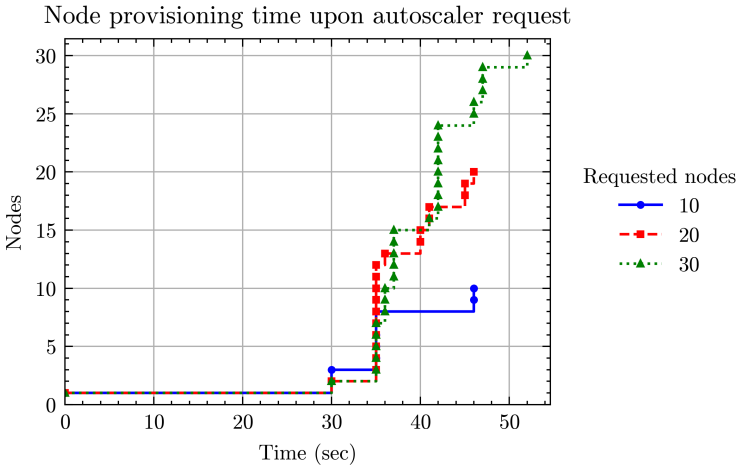
Com s'ha observat en els gràfics anteriors, l'augment de recursos inicialment proporciona un creixement lineal del rendiment, més tard sublineal i finalment s'estabilitza sense cap millora. Per aquest motiu cal analitzar en cada cas quina concurrència és l'òptima.

7.3 Experiment 3

7.3.1 Resultats

En aquest experiment s'han realitzat proves amb un clúster de mida variant per estudiar el comportament de la redimensió del clúster.

Per a la prova, es va partir d'un únic node, el head node, amb el clúster configurat per escalar fins a un màxim de 10/20/30 nodes. A continuació s'executava el codi de l'EMBL amb el dataset de 10.000 imatges, un batch size de 8 i 10/20/30 actors. Mitjançant el codi de Python creat per aquest experiment es guarda la informació de creació dels Pods que a continuació es representen:



Gràfic 8. Temps creació workers (experiment 3)

En aquest gràfic es pot veure com ha escalat el clúster (t = 0 equival a la primera sol·licitud de l'autoescalador). S'aprecia com el clúster s'ha redimensionat fins a arribar als 10/20/30 nodes necessaris per a l'execució dels 10/20/30 actors. També es pot veure que el temps total per inicialitzar els nous nodes ha estat de 46 segons per 10/20 nodes i 52 segons per 30 nodes.

A continuació s'ha comparat a executar el mateix codi i les mateixes configuracions de nodes en un clúster de mida fixa (sense autoescalador) i un amb mida variable (amb autoescalador) guardant-ne les dades de la latència i el rendiment.

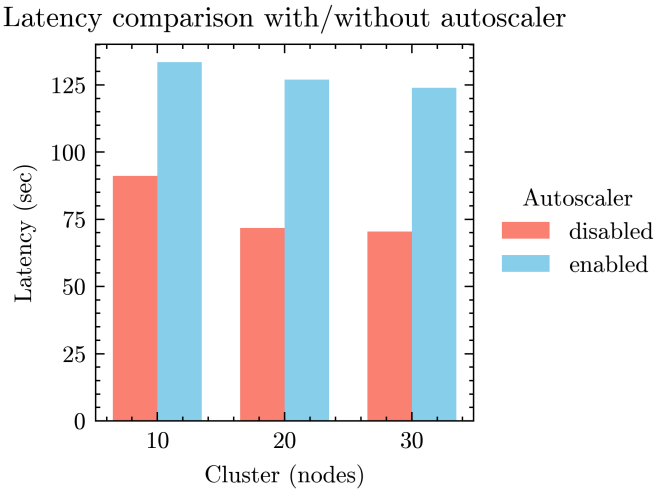
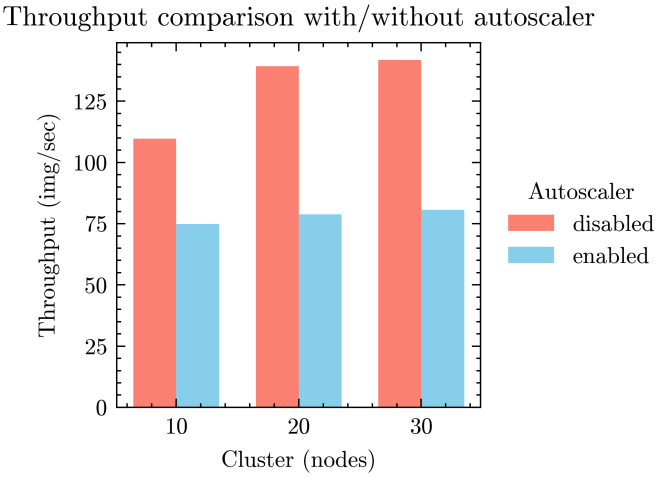


Figura 27. Latència vs Autoescalador desactivat/activat (experiment 3)

Com es pot veure en la gràfica la latència es veu afectada en funció de si s'utilitza o no l'autoescalador. La diferència de segons ve directament determinada pel temps que ha tardat el clúster a redimensionar els recursos. **La diferència entre la latència és directament proporcional al temps de redimensionament del clúster pel fet que l'execució del codi de Ray no s'inicia fins que el clúster està completament redimensionat**, ja que Ray coneix la demanda de recursos per avançat a partir dels paràmetres declarats en les funcions utilitzades. Cal destacar que Ray inicialitza els Actors, carregant el model, però no comença l'execució fins que estan tots creats.



Gràfic 9. Rendiment vs Autoescalador desactivat/activat (experiment 3)

Conseqüentment, la latència afecta el rendiment de l'execució. En el cas del clúster amb mida variable, el rendiment és entre un 30 i 40% menor en la primera execució en comparació amb el clúster de mida fixa. No obstant això, en les execucions subsegüents, el rendiment no es veu afectat, ja que el clúster ja està redimensionat.

7.3.2 Resposta a la qüestió 3

La utilització d'un clúster amb autoescalador impacta en el rendiment de les aplicacions?

Com s'ha demostrat en les gràfiques anteriors, la utilització de l'autoescalador afecta negativament el rendiment. Encara que a primera vista pugui semblar que és un factor negatiu a causa de l'impacte inicial en la latència, els seus avantatges superen aquest inconvenient. L'autoescalador permet una variabilitat del clúster, adaptant-se dinàmicament a les necessitats concretes del codi executat. En un entorn real, on es treballa amb datasets de mides variables i codis amb diferents demandes de recursos, aquesta capacitat de redimensionament és imprescindible, fent que el clúster s'expandeixi o es contragui segons la càrrega de treball actual, optimitzant l'ús dels recursos disponibles i satisfent els pics de demanda.

També cal destacar que implementar un clúster sense autoescalador en un entorn del cloud pot ser econòmicament inviable, ja que una configuració fixa de recursos implica un cost continu, independentment de la càrrega de treball real. A més, durant períodes de baixa demanda, gran part dels recursos no es fan servir. En canvi, un clúster amb autoescalador ajusta automàticament el nombre de nodes segons la demanda, permeten maximitzar l'ús de recursos i reduir els costos, ja que els recursos només s'aprovisionen quan són necessaris i s'alliberen quan la càrrega disminueix, garantint així que ens facturin únicament els utilitzats.

7.4 Recomanacions

Mitjançant les proves realitzades es pot concloure que és millor treballar amb datasets amb una major quantitat d'imatges com el de 10.000 perquè permet augmentar el paral·lelisme de la lectura a 200. La mida del batch ha de ser petita, una molt bona opció és 8. Respecte a la concurrència que està vinculada amb els nodes disponibles al clúster depèn dels recursos totals disponibles, els nodes mitjans (3CPU/5RAM) en general dona bons resultats a l'experiment 1, per com s'ha vist a l'experiment 2, si s'augmenten per sobre de 20 el rendiment no millora a causa de la llei de rendiments decreixents, per tant, és millor una configuració més potent (6CPU/10RAM), ja que en comparació proporciona un pic de rendiment major. Finalment, cal destacar que l'autoescalador afecta directament el rendiment donat el funcionament de Ray, però en entorns del cloud és imprescindible per la reducció de costos.

En un futur seria interessant dissenyar una tècnica capaç d'ajustar els requisits del codi a executar en funció de la mida del dataset i els recursos màxim del clúster.

8 Conclusions

Durant el desenvolupament d'aquest projecte, s'han assolit els objectius plantejats amb èxit. Inicialment, es va realitzar una recerca prèvia per comprendre l'abast i el funcionament del framework de Ray, centrant-se especialment en la llibreria Ray Data que proporciona les eines necessàries per a la inferència per lots fora de línia. Aquesta investigació inicial ha estat molt important per establir una base sòlida de coneixements sobre el framework que han sigut necessaris per al desenvolupament posterior.

A continuació, es va crear una versió inicial del codi utilitzant el model preentrenat ResNet50. Més tard, aquest codi es va adaptar per utilitzar el model Off-sample, dissenyat per detectar si una imatge d'espectrometria de masses es troba fora de mostra.

Finalment, es va crear un clúster local de Ray utilitzant Kubernetes, que ha permès familiaritzar-se amb la seva utilització. Posteriorment, s'ha pogut desplegar satisfactòriament al rack del CloudLab, un entorn amb molts més recursos on s'han fet els experiments finals.

Els resultats obtinguts demostren la necessitat de buscar la configuració òptima dels paràmetres per millorar el rendiment i l'assignació eficient dels recursos del clúster per maximitzar l'ús del maquinari sense sobreaprovisionar el clúster. A més, s'ha analitzat el comportament de l'autoescalador, valorant la importància de la seva utilització.

Personalment, considero que ha estat molt interessant realitzar aquest projecte d'investigació com a treball de fi de grau que va més enllà del típic desenvolupament d'una aplicació mòbil o d'una pàgina web. He tingut l'oportunitat de descobrir i utilitzar tecnologies innovadores per la computació paral·lela que no s'han tractat durant el grau. És important destacar que l'aplicació desenvolupada amb el model Off-sample té una utilitat real. M'hauria agradat poder aprofundir encara més en el framework de Ray, perquè ofereix moltes altres eines i funcionalitats, però donada la limitació dels objectius, les constants actualitzacions del framework i el límit de temps, no he pogut explorar-lo en més profunditat. No obstant això, aquest treball m'ha proporcionat una base sòlida que serà de gran utilitat en el meu futur professional.

9 Bibliografía

[1] «EMBL Website,» [En línia]. Disponible: https://www.embl.org

[2] «OffsampleAI paper,» [En línia]. Disponible: https://bmcbioinformatics.biomedcentral.com/articles/10.1186/s12859-020-3425-x

[3] EMBL, «Metaspace,» [En línia]. Disponible: https://metaspace2020.eu

[4] EMBL, «Off-sample model,» [En línia]. Disponible: https://github.com/metaspace2020/offsample

[5] «CloudButton Serverless Data Analytics Platform,» [En línia]. Disponible: https://cloudbutton.eu/docs/deliverables/CloudButton_D2.5.pdf

[6] «Amazon SageMaker,» [En línia]. Disponible: https://docs.aws.amazon.com/es_es/sagemaker/latest/dg/whatis.html

[7] «Apache Spark,» [En línia]. Disponible: https://spark.apache.org/docs/3.5.1/

[8] «Ray,» [En línia]. Disponible: https://docs.ray.io/en/releases-2.9.3/

[9] «Article Anyscale,» [En línia]. Disponible: https://www.anyscale.com/blog/offline-batch-inference-comparing-ray-apache-spark-and-sagemaker

[10] «Ray Overview,» [En línia]. Disponible: https://docs.ray.io/en/releases-2.9.3/ray-overview/index.html

[11] «Ray Core,» [En línia]. Disponible: https://docs.ray.io/en/releases-2.9.3/ray-core/walkthrough.html

[12] «Ray Cluster,» [En línia]. Disponible: https://docs.ray.io/en/releases-2.9.3/cluster/getting-started.html

[13] «Ray on Kubernetes,» [En línia]. Disponible: https://docs.ray.io/en/releases-2.9.3/cluster/kubernetes/index.html

[14] «Ray on Cloud VMs,» [En línia]. Disponible: https://docs.ray.io/en/releases-2.9.3/cluster/vms/index.html

[15] «Ray Autoscaler,» [En línia]. Disponible: https://docs.ray.io/en/releases-2.9.3/cluster/kubernetes/user-guides/configuring-autoscaling.html#overview

[16] «Ray Job,» [En línia]. Disponible: https://docs.ray.io/en/releases-2.9.3/cluster/running-applications/job-submission/index.html

[17] «Ray Data,» [En línia]. Disponible: https://docs.ray.io/en/releases-2.9.3/data/key-concepts.html

[18] «Offline Batch Inference,» [En línia]. Disponible: https://docs.ray.io/en/releases-2.9.3/data/batch_inference.html

[19] M. Pumperla, E. Oakes y R. Liaw, Learning Ray, O'Reilly

[20] «Ray Read APIs,» [En línia]. Disponible: https://docs.ray.io/en/releases-2.9.3/data/api/input_output.html

[21] «Tuning read parallelism,» [En línia]. Disponible: https://docs.ray.io/en/releases-2.9.3/data/performance-tips.html#tuning-read-parallelism

[22] «Ray Monitoring and Debugging,» [En línia]. Disponible: https://docs.ray.io/en/releases-2.9.3/ray-observability/index.html

[23] «Models and pre-trained weights,» [En línia]. Disponible: https://pytorch.org/vision/main/models.html

[24] «ResNet50 model,» [En línia]. Disponible: https://pytorch.org/vision/main/models/generated/torchvision.models.resnet50.html

[25] «ImageNet,» [En línia]. Disponible: https://www.image-net.org

[26] «Kubernetes,» [En línia]. Disponible: https://kubernetes.io/docs/tutorials/kubernetes-basics/

[27] «MinIO Core Operational Concepts,» [En línia]. Disponible: https://min.io/docs/minio/kubernetes/upstream/operations/concepts.html

[28] CloudLab, «Manual de primera conexión al rack»

[29] «rayproject/ray:2.9.3-py311,» [En línia]. Disponible: https://hub.docker.com/layers/rayproject/ray/2.9.3-py311/images/sha256-6baf62ec97ab5e94719246aaf23544f4740d623373a65ccaa3e038c710b9548c?context=explore

[30] «Ray Client error "Global node is not initialized.",» [En línia]. Disponible: https://github.com/ray-project/ray/issues/41333

[31] «javierqt26/ray_py311_requirements,» [En línia]. Disponible: https://hub.docker.com/r/javierqt26/ray_py311_requirements

[32] «Head node out-of-Memory error,» [En línia]. Disponible: https://docs.ray.io/en/releases-2.9.3/ray-observability/user-guides/debug-apps/debug-memory.html

[33] «Especificacions ResNet50,» [En línia]. Disponible: https://pytorch.org/vision/main/models/generated/torchvision.models.resnet50.html#torchvision.models.ResNet50_Weights

[34] «Transformacions Fastai,» [En línia]. Disponible: https://github.com/fastai/fastai1/blob/master/fastai/vision/image.py#L96

[35] «GitHub Ray Autoscaler (YAML),» [En línia]. Disponible: https://github.com/ray-project/kuberay/blob/master/ray-operator/config/samples/ray-cluster.autoscaler.yaml

[36] «Wikipedia Diminishing returns,» [En línia]. Disponible: https://en.wikipedia.org/wiki/Diminishing_returns

Annexos

Disponibilitat del codi

El projecte desenvolupat es troba disponible en el meu GitHub personal. L'enllaç al respositori és el següent:

https://github.com/javierqt26/ray-benchmarks