

MSE - TP Clemap

Documentation de TP

Version 1

Pellandini Leo
Victor Cochet

19.12.2024

Table des matières

Introduction	2
Workflow général de l'application	2
Capteur de récolte de données Clemap	3
Script de prédiction	5
AWS	9
AWS Greengrass	9
AWS S3 Bucket - Ré-entraînement	11
AWS Stream Manager	14
AWS Sagemaker	15
Problèmes rencontrés	17
Compatibilité armv7 32 bits avec les librairies de machine learning	17
Alternative avec container	17
Conclusion	19
Références	19

Introduction

Ce projet vise à prédire la consommation électrique future à partir des données collectées par un capteur CLEMAP connecté à une prise électrique. Grâce à une combinaison de machine learning et de traitements locaux avec AWS Greengrass, le système analyse les mesures de puissance en temps réel pour anticiper les besoins énergétiques.

Les données sont centralisées dans le cloud via Amazon S3 et utilisées pour entraîner et améliorer continuellement un modèle de prédiction grâce à AWS SageMaker. Des fonctions AWS Lambda automatisent le cycle d'apprentissage et de déploiement du modèle, assurant un processus efficace et autonome.

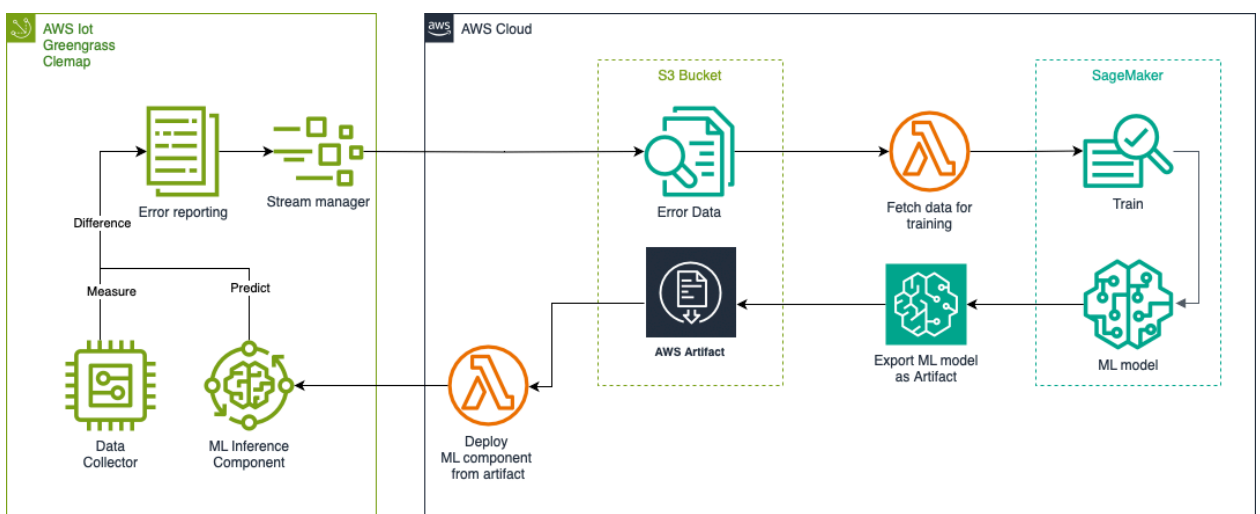
En alliant l'Edge Computing et la puissance du Cloud AWS, ce projet permet une gestion intelligente et prédictive de la consommation électrique, offrant ainsi une solution adaptable et optimisée pour anticiper les besoins énergétiques.

Workflow général de l'application

Le processus commence avec la collecte des données de consommation électrique grâce au capteur CLEMAP, qui mesure la puissance utilisée sur une prise. Ces données sont envoyées localement à AWS Greengrass, qui assure leur traitement initial avant de les transmettre vers le cloud. Une fois dans le cloud AWS, les données sont centralisées dans un bucket Amazon S3 via Stream Manager. À partir de ce stockage, une fonction AWS Lambda récupère les données pour entraîner un modèle de machine learning dans AWS SageMaker. Le modèle ainsi obtenu est ensuite exporté sous forme d'artifact pour être utilisé sur le terrain.

Le modèle mis à jour est déployé localement sur AWS Greengrass à l'aide d'une autre fonction Lambda. À ce stade, Greengrass utilise le modèle pour réaliser des prédictions locales de la consommation électrique en temps réel. Les erreurs entre les prédictions et les valeurs réelles sont mesurées et renvoyées dans le pipeline afin de réentraîner le modèle, améliorant ainsi continuellement sa précision.

Ce processus en boucle, combinant le stockage sur S3, l'entraînement sur SageMaker et l'exécution locale sur Greengrass, garantit une solution autonome et performante pour la prédiction et l'optimisation de la consommation électrique.



Capteur de récolte de données Clemap

Lors de l'intégration du capteur CLEMAP, nous avons rencontré des difficultés liées à l'architecture ARMv7 32 bits utilisée par défaut. Certaines bibliothèques Python essentielles pour notre projet, comme TensorFlow, ne parvenaient pas à s'installer en raison d'un manque de compatibilité avec cette version de l'OS. Ce problème nous a empêchés de déployer correctement les composants de machine learning nécessaires à la prédiction de la consommation électrique.

Pour résoudre cette contrainte, nous avons procédé à une mise à niveau du système d'exploitation en passant à Raspberry Pi OS Lite ARMv8 64 bits. Cette nouvelle architecture 64 bits offre une meilleure compatibilité avec les bibliothèques modernes de machine learning telles que TensorFlow et d'autres dépendances critiques.

Après l'upgrade, nous avons réinstallé toutes les dépendances Python nécessaires, ainsi que Docker, afin de faciliter l'exécution de conteneur pour collecter et traiter les données de consommation électrique.

Nous avons commencé par mettre à jour notre raspberry afin d'y installer Docker dessus.

- Mettre à jour le Legacy source avec [apt -source]
- Installation de Docker

```
sens@clemap51:~$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
828a38d1ce5c	clemap/edge_fw:latest	"./startup.sh"	12 seconds ago	Up Less than a second	0.0.0.0:4001->4000/tcp, :::4001->4000/tcp	clemap_edge_fw

Voici la commande utilisée pour run le docker qui récupère les données, il a fallu bien faire attention que le fichier config.ini soit configuré ainsi :

```
docker run -d \
  --name clemap_edge_fw \
  -p 4001:4000 \
  -v /home/sens/TP_Clemap/sens_docker/config.ini:/usr/src/app/config/config.ini \
  -v /home/sens/sens_docker/clemap_db:/usr/src/app/DB:rw \
  -e EDGE_FW_INFO_HW_SERIAL=SE05000551 \
  -e DEVICE_TYPE=cem_containerized \
  -e EDGE_FW_INFO_SENSOR_STATUS=1 \
  --device /dev/gpiomem:/dev/gpiomem \
  --device /dev/mem:/dev/mem \
  --device /dev/spidev0.0:/dev/spidev0.0 \
  --cap-add SYS_ADMIN \
  --cap-add SYS_TIME \
  --privileged \
  clemap/edge_fw:latest
```

Voici le fichier config.ini :

```
[INFO]
hw_serial = SE05000551
sensor_status = 1
sensor_id = 672a3431d8dff0014ef0283
timezone = Europe/Zurich

[NETWORKING]
wifi_country = CH

[METER_SERVICE]
controller_model = ade9000
current_sensor_rating = 42
nib_version = 3
current_sensor_type = 1
calib_cigain = 1339782
calib_avrmsos = 14
calib_cphcal0 = -56192894
calib_bvgain = -259825
calib_cvgain = -12329
calib_bigain = 1264722
calib_airmsos = 35
calib_bpgain = -330
calib_bvaros = 1
calib_avaros = 1
calib_cpgain = -3307
calib_cvaros = 1
calib_bphcal0 = -58592471
calib_bvrmsos = 14
calib_avgain = 260287
calib_cirmsos = 33
calib_aigain = 2069224
calib_apgain = -13889
calib_birmsos = 20
calib_bwattos = 2
calib_awattos = 4
calib_aphcal0 = -63482710
calib_cvrmsos = 14
calib_cwattos = 1
cha_irms = 1.5844520367023384e-06
cha_vrms = 1.3294291998883081e-05
cha_watt = 0.0028271847549624046
cha_va = 0.0028271847549624046
cha_var = 0.0028271847549624046
cha_watthr = 0.0064334159757366725
cha_varhr = 0.0064334159757366725
cha_vahr = 0.0064334159757366725
chb_irms = 1.5844520367023384e-06
chb_vrms = 1.3294291998883081e-05
chb_watt = 0.0028271847549624046
chb_va = 0.0028271847549624046
chb_var = 0.0028271847549624046
```

```
chb_watthr = 0.0064334159757366725
chb_varhr = 0.0064334159757366725
chb_vahr = 0.0064334159757366725
chc_irms = 1.5844520367023384e-06
chc_vrms = 1.3294291998883081e-05
chc_watt = 0.0028271847549624046
chc_va = 0.0028271847549624046
chc_var = 0.0028271847549624046
chc_watthr = 0.0064334159757366725
chc_varhr = 0.0064334159757366725
chc_vahr = 0.0064334159757366725
```

```
[GPIO_SERVICE]
output_pin_setup = ab
pin_out_a = 21
pin_out_b = 20
default_input_value = 0
input_pin_setup = b
pin_in_b = 16
led_pin = 25
reset_button_pin = 7
```

```
[MODULES]
data_gathering_module = true
```

```
[DATA_GATHERING_MODULE]
gathering_interval_sec = 60
persistence_interval_sec = 1209600
db_path = DB/data_gathering.db
```

```
[LOGGING]
level = info
```

Afin de voir les données minute par minute voici un moyen de vérifier si la database se remplit bel et bien?

```
sqlite3 /home/sens/sens_docker/clemap_db/data_gathering.db
SELECT * FROM meter_data LIMIT 10;
```

Script de prédiction

Nous avons déployé un script Python sur l'Edge Device pour permettre la collecte des données électriques et la prédiction de la consommation future en temps réel. Il repose sur une base SQLite, un modèle TensorFlow préalablement entraîné et exporté au format .h5, et des bibliothèques essentielles comme NumPy. Voici une explication détaillée de son fonctionnement:

Le script lit les données de consommation électrique stockées dans une base de données SQLite située à l'adresse /home/sens/sens_docker/clemap_db/data_gathering.db. Ces données sont récupérées depuis la table meter_data, où les colonnes l1_p, l2_p et l3_p représentent les puissances des trois phases. Le script extrait les 10 dernières entrées par ordre chronologique inversé et calcule la somme des puissances pour chaque enregistrement. Comme ceci :

```
import sqlite3
import os

db_path = "/home/sens/sens_docker/clemap_db/data_gathering.db"

def read_data():
    print("Database exists:", os.path.exists(db_path))
    conn = sqlite3.connect(db_path)
    cursor = conn.cursor()
    query = """
    SELECT time, l1_p, l2_p, l3_p
    FROM meter_data
    ORDER BY time DESC
    LIMIT 10
    """
    cursor.execute(query)
    rows = cursor.fetchall()
    conn.close()
    data = [row[1] + row[2] + row[3] for row in rows]
    return data
```

Une fois les données collectées, elles sont ordonnées dans le sens chronologique afin de préparer une séquence temporelle. Les données sont ensuite converties en un tableau NumPy avec un format compatible pour le modèle TensorFlow :

Dimensions : (1, 10, 1) où 1 représente le batch size, 10 le nombre de pas de temps, et 1 la feature (somme des puissances). Comme ci-dessous :

```
import numpy as np

def prepare_data(data):
    data = data[::-1] # Réordonner les données
    data = np.array(data, dtype=np.float32).reshape((1, len(data), 1)) # Reshape
    return data
```

Le script vérifie ensuite la présence du fichier model.h5 à l'emplacement spécifié (/tmp/model.h5).

```

from tensorflow.keras.models import load_model
from keras.losses import MeanSquaredError
import os

model_path = "/tmp/model.h5"

def download_model():
    print(f"Chargement du modèle depuis {model_path}...")
    custom_objects = {'mse': MeanSquaredError()}
    model = load_model(model_path, custom_objects=custom_objects, compile=False)
    return model

```

Le modèle effectue une prédiction à partir des données préparées :

Le modèle effectue une prédiction à partir des données préparées. Comme ci-dessous :

```

def predict(input_data, model):
    prediction = model.predict(input_data)
    print(f"Prédiction de la prochaine puissance (sum_p) : {prediction[0][0]}")
    return prediction

```

Si l'erreur dépasse un seuil, un rapport CSV est généré pour le réentraînement :

```

def create_error_report(db_path):
    import numpy as np

    conn = sqlite3.connect(db_path)
    cursor = conn.cursor()
    sums = []
    for _ in range(100):
        cursor.execute("SELECT I1_p, I2_p, I3_p FROM meter_data ORDER BY time DESC LIMIT 10")
        rows = cursor.fetchall()
        data = [row[0] + row[1] + row[2] for row in rows]
        sums.append(sum(data))
    conn.close()

    np.savetxt('/tmp/Clemap_train.csv', np.array(sums), delimiter=',', header='target',
    comments="", fmt='%f')
    print("Rapport d'erreur généré dans /tmp/Clemap_train.csv")

```

Les données d'erreur sont exportées vers S3 :

```

from stream_manager import StreamManagerClient, S3ExportTaskDefinition, Status, Util

def send_data_to_cloud():
    client = StreamManagerClient()
    stream_name = "ErrorStream"
    bucket_name = "clemapbucket"
    key_name = "data/Clemap_train.csv"

```



```
file_url = "file:/tmp/Clemap_train.csv"

client.create_message_stream(stream_name)
task_definition = S3ExportTaskDefinition(input_url=file_url, bucket=bucket_name,
key=key_name)
client.append_message(stream_name,
Util.validate_and_serialize_to_json_bytes(task_definition))
print("Données envoyées vers S3.")
```

Le processus tourne en continu pour prédire, détecter les erreurs et déclencher des actions si nécessaire. L'Edge Device agit ainsi comme un nœud intelligent capable de lire les données, préparer des séquences temporelles et exécuter des prédictions avec le modèle h5 fourni par le cloud. Ce code permet un déploiement sur l'Edge Device avec des fonctionnalités telles que la prédiction en temps réel, la gestion des erreurs dynamiques, et l'intégration avec AWS IoT Greengrass pour assurer la remontée des données vers le Cloud et la mise à jour des modèles.

AWS

L'objectif de l'utilisation de AWS est de concevoir un workflow automatisé dans lequel le capteur CLEMAP collecte les données de consommation électrique et les envoie à AWS Greengrass. Ces données serviront à entraîner un modèle de machine learning capable de prédire la consommation électrique future. Une fois les prédictions réalisées, elles seront transmises à une application pour affichage des résultats en temps réel.

Pour répondre à ces besoins, nous avons choisi AWS SageMaker comme outil central pour l'entraînement et l'optimisation continue du modèle de prédiction. AWS SageMaker offre une infrastructure scalable et performante pour entraîner, déployer et mettre à jour les modèles de machine learning, tout en intégrant facilement les composants AWS Greengrass pour l'inférence locale.

Ce workflow complet, du collecte des données jusqu'à l'affichage des prédictions, permet d'allier traitement local et puissance du cloud AWS, assurant ainsi une solution rapide, fiable et évolutive pour l'anticipation des besoins énergétiques.

AWS Greengrass

Notre infrastructure repose sur AWS IoT Greengrass pour exécuter les composants locaux nécessaires à la collecte, au traitement et à la prédiction des données de consommation électrique. Le dispositif principal, nommé RealClemap, fonctionne sur une plateforme linux/aarch64 avec une version de Greengrass Core : `aws_nucleus_classic 2.13.0`. Cette configuration permet une intégration robuste avec les services AWS tout en tirant parti de l'exécution locale pour minimiser les latences.

[AWS IoT](#) > [Greengrass](#) > [Appareils principaux](#) > RealClemap

RealClemap

Supprimer

Présentation↻

Les appareils principaux Greengrass sont des objets AWS IoT qui exécutent le logiciel Greengrass Core.

Objet RealClemap	Logiciel d'exécution Greengrass Core et version <code>aws_nucleus_classic 2.13.0</code>	Statut ✓ Sain
Plateforme linux/aarch64	Journaux Afficher dans CloudWatch	Statut signalé il y a 8 heures

Le déploiement de notre solution est géré via AWS Greengrass Groups, ici représenté par le groupe ClemapDummyGroup. Ce groupe permet de définir et de gérer les déploiements de plusieurs appareils, facilitant ainsi l'exécution continue des tâches sur l'ensemble de notre parc d'équipements. Actuellement, trois appareils sont déployés avec succès, chacun ayant un statut sain comme en témoigne leur signalement actif dans AWS. La raison est qu'avant de régler les problèmes d'architecture 32 bits sur le clemap nous avons fait des déploiement sur un autre raspberry pi qui lui était déjà avec une architecture 64 bits.

Présentation

AWS IoT Greengrass utilise des tâches AWS IoT continues pour déployer des groupes d'objets.

Cible

[ClemapDummyGroup](#)

Type de cible

Groupe d'objets

Déploiement créé

il y a 16 heures

Tâche IoT

[bb27e055-81ac-47ef-afab-05bf013bce0a](#)

Statut du déploiement

Actif

Exécutions

Appareils

Composants

Configurations

Sous-déploiements

Balises

Appareils déployés (3) [Infos](#)

Appareils principaux ayant correctement déployé une révision de ce déploiement.

Statut de l'appareil

Tout statut de l'appareil

	Nom de l'appareil principal	Exécution	Statut de l'appareil	Statut signalé
<input type="checkbox"/>	RealClemap	aws_nucleus_classic	Sain	il y a 8 heures

DeployMLfromArtifact

Limiter
Copier l'ARN
Actions ▼

▼ Présentation de la fonction Info

Diagramme

Modèle

DeployMLfromArtifact

Layers (0)

S3

+ Ajouter une destination

+ Ajouter un déclencheur

Description

-

Dernière modification
il y a 17 heures

ARN de la fonction
 arn:aws:lambda:us-east-1:060795903373:function:DeployMLfromArtifact

URL de fonction Info

-

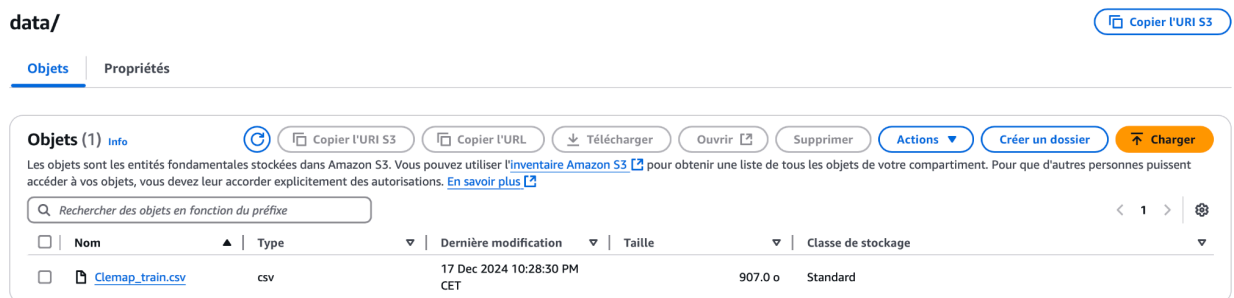
AWS S3 Bucket - Ré-entraînement

Notre bucket Amazon S3 constitue un point central pour le stockage des données et l'automatisation du réentraînement des modèles SageMaker. Deux répertoires principaux sont utilisés dans ce bucket :

1. /data :

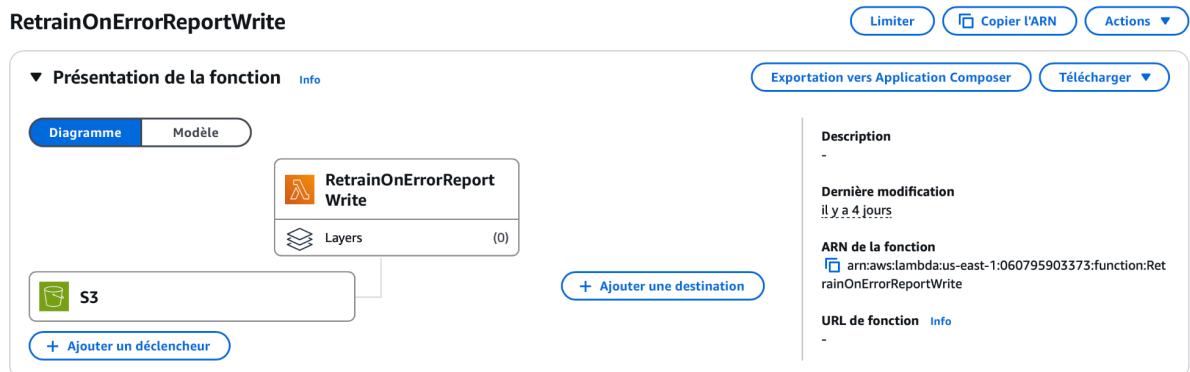
Ce répertoire stocke les données d'entraînement du modèle de machine learning. Les fichiers, comme Clemap_train.csv, contiennent les mesures collectées par le capteur CLEMAP.

Lorsqu'un décalage est identifié entre les prédictions et les données réelles, ces nouvelles données sont ajoutées pour ajuster et améliorer les performances du modèle.



2. /scripts :

Cet emplacement contient les scripts nécessaires pour exécuter le processus d'entraînement sur AWS SageMaker, comme le fichier train_lstm.py. Ce script définit l'architecture du modèle et la logique de l'entraînement.



Code qui gère le ré-entraînement :

```
import json
import boto3
import os
import uuid
import time

# Client S3 et SageMaker
s3_client = boto3.client('s3')
```

```

sagemaker_client = boto3.client('sagemaker')

# Paramètres de ton job SageMaker
ROLE =
'arn:aws:iam::060795903373:role/service-role/AmazonSageMaker-ExecutionRole-20241114T
100363' # Remplace par ton rôle SageMaker
BUCKET_NAME = 'clemapbucket'
MODEL_OUTPUT_PATH = 's3://clemapbucket/output/' # Emplacement de sortie pour les
résultats du modèle
SOURCE_INPUT_PATH = 's3://clemapbucket/scripts/train_lstm.py'
TRAINING_IMAGE =
'763104351884.dkr.ecr.us-east-1.amazonaws.com/tensorflow-training:2.16.2-cpu-py310'
# Image Docker pour l'entraînement
KEY = 'data/Clemap_train.csv' # Corrected key to point to the file directly

def lambda_handler(event, context):
    # Créer un job d'entraînement SageMaker
    timestamp = str(int(time.time()))
    unique_suffix = str(uuid.uuid4())[:8] # Limite à 8 caractères
    job_name = f"lstm-training-job-{timestamp}-{unique_suffix}"

    # Configurer le job d'entraînement
    response = sagemaker_client.create_training_job(
        TrainingJobName=job_name,
        AlgorithmSpecification={
            'TrainingImage': TRAINING_IMAGE,
            'TrainingInputMode': 'File',
            "ContainerEntrypoint": ["/usr/local/bin/python3.10"],
            "ContainerArguments": ["/opt/ml/input/data/scripts/train_lstm.py"]
        },
        RoleArn=ROLE,
        InputDataConfig=[
            {
                'ChannelName': 'training',
                'DataSource': {
                    'S3DataSource': {
                        'S3DataType': 'S3Prefix',
                        'S3Uri': f's3://{BUCKET_NAME}/data', # Use dynamic bucket
and key from event
                        'S3DataDistributionType': 'FullyReplicated'
                    }
                },
                'ContentType': 'csv',
                'ChannelName': 'scripts',

```

```

        'DataSource': {
            'S3DataSource': {
                'S3DataType': 'S3Prefix',
                'S3Uri': f's3://{BUCKET_NAME}/scripts', # Use dynamic bucket
and key from event
                'S3DataDistributionType': 'FullyReplicated'
            }
        },
        'ContentType': 'py',
    }
],
OutputDataConfig={
    'S3OutputPath': MODEL_OUTPUT_PATH
},
ResourceConfig={
    'InstanceType' : 'ml.m5.large',
    'InstanceCount': 1,
    'VolumeSizeInGB': 1
},
StoppingCondition={
    'MaxRuntimeInSeconds': 3600 # Limiter le temps d'exécution
}
)

return {
    'statusCode': 200,
    'body': json.dumps(f'Job started: {job_name}')
}

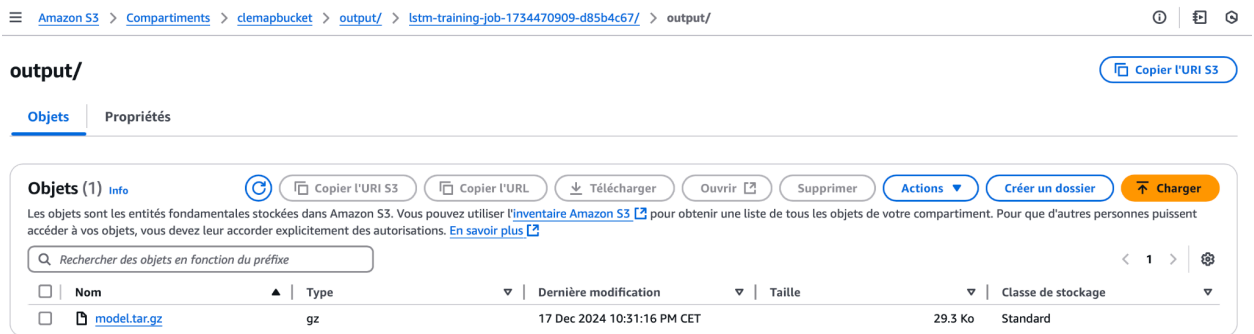
```

Pour optimiser le processus de réentraînement, nous avons mis en place une fonction AWS Lambda nommée `RetrainOnErrorReportWrite`. Cette fonction s'exécute lorsqu'une erreur de prédiction est détectée ou lorsqu'un événement S3 déclenche une mise à jour des données dans `/data`.

La fonction Lambda lance un job SageMaker de manière automatisée. Voici son fonctionnement :

- Elle utilise un rôle IAM pour autoriser l'exécution du job d'entraînement.
- Elle spécifie l'image Docker TensorFlow pour configurer l'environnement d'exécution.
- Les données sont récupérées depuis `/data` dans le bucket S3.
- Le script d'entraînement (`train_lstm.py`) est extrait depuis `/scripts`.
- Le modèle entraîné est sauvegardé dans le répertoire `/output` du bucket S3, ce qui permet un déploiement continu vers AWS Greengrass.

Une fois le model ré-entraîner SageMaker stocke le nouveau modèle dans le S3 :



Amazon S3 > Compartiments > demapbucket > output/ > lstm-training-job-1734470909-d85b4c67/ > output/

output/ Copier l'URI S3

Objets (1) Info

Les objets sont les entités fondamentales stockées dans Amazon S3. Vous pouvez utiliser l'[inventaire Amazon S3](#) pour obtenir une liste de tous les objets de votre compartiment. Pour que d'autres personnes puissent accéder à vos objets, vous devez leur accorder explicitement des autorisations. [En savoir plus](#)

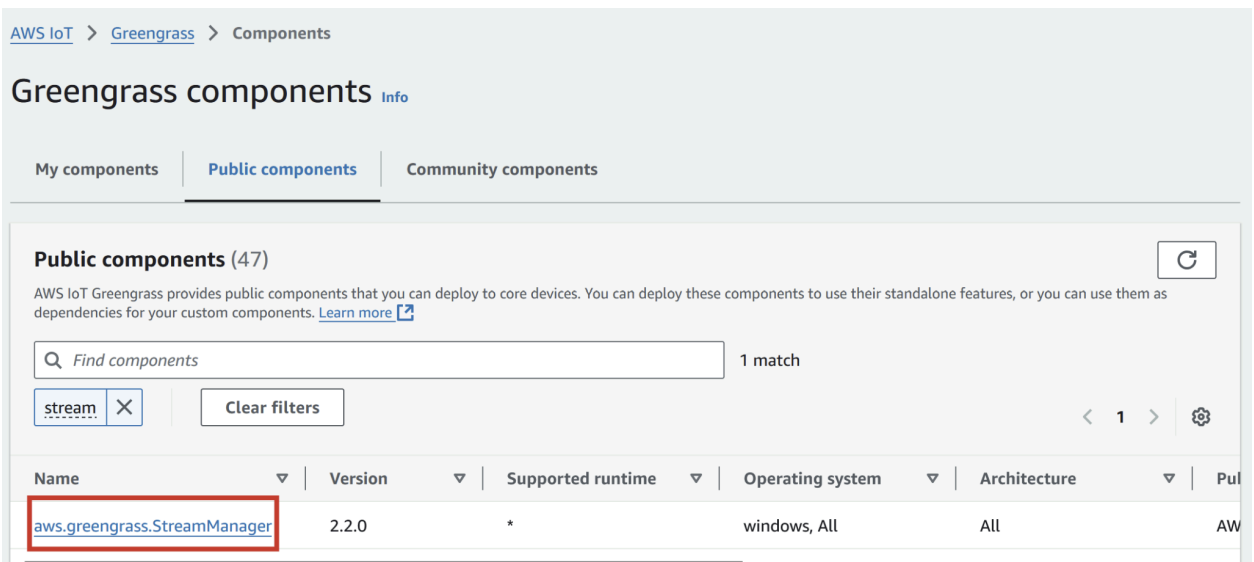
Rechercher des objets en fonction du préfixe

Nom	Type	Dernière modification	Taille	Classe de stockage
model.tar.gz	gz	17 Dec 2024 10:31:16 PM CET	29.3 Ko	Standard

AWS Stream Manager

Lorsqu'une prédiction de l'instance du modèle sur notre edge device est erronée nous devons remonter des données d'entraînements pour le prochain modèle sur le cloud. Nous utilisons le stream manager qui est un composant qui permet de stocker et exporter les données à partir du edge device dans l'environnement greengrass. Pour l'utiliser nous avons besoin de deux éléments :

1. Le composant générique "aws.greengrass.StreamManager" doit être déployé
2. Le composant utilisant le StreamManager doit avoir le composant générique en dépendance dans sa recette.
3. Le composant utilisant le StreamManager doit également avoir le module python "stream_manager" pour interagir avec le composant public.



AWS IoT > Greengrass > Components

Greengrass components Info

My components Public components Community components

Public components (47)

AWS IoT Greengrass provides public components that you can deploy to core devices. You can deploy these components to use their standalone features, or you can use them as dependencies for your custom components. [Learn more](#)

Find components 1 match

stream X Clear filters

Name	Version	Supported runtime	Operating system	Architecture	Pub
aws.greengrass.StreamManager	2.2.0	*	windows, All	All	AW

Il se trouve sous les composants publiques dans la console AWS lot -> Greengrass et pour l'ajouter en dépendance de notre composant, on doit rajouter la ligne :

```
"ComponentDependencies": {
  "aws.greengrass.StreamManager": {
    "VersionRequirement": ">=2.0.0 <3.0.0",
    "DependencyType": "HARD"
  }
},
```

Dans la recette de notre composant qui utilise le Stream Manager.

Une fois que cela est fait, on doit utiliser le module “stream_manager” dans notre script python pour créer un stream et interagir avec le composant pour envoyer des données dans le cloud. Il est compatible avec plusieurs autres services, notamment le S3 que nous utilisons. C’est lui qui va se charger de remonter les données vers le S3 lorsque l’on veut ré-entraîner le modèle.

AWS Sagemaker

Notre infrastructure utilise Amazon SageMaker pour l'entraînement et le déploiement continu du modèle de prédiction de la consommation électrique. Voici les principaux éléments de configuration que nous avons mis en place :

Nous avons créé un domaine SageMaker nommé QuickSetupDomain-20241114T100362 qui assure la gestion centralisée des environnements SageMaker, des profils utilisateur et des configurations d'applications. Le domaine est opérationnel et fonctionne dans un VPC dédié pour garantir la sécurité des communications réseau.

Amazon SageMaker AI

>

Domaines

>

Domaine : QuickSetupDomain-20241114T100362

QuickSetupDomain-20241114T100362

Détails du domaine

Configurez et gérez le domaine.

Paramètres de domaine

Profils utilisateur

Gestion de l'espace

Configurations des applications

Environnement

Ressources

Paramètres généraux

Infos

Nom	Statut	ID de domaine
QuickSetupDomain-20241114T100362	Ready	d-4hpdqyukaamb
Créé	Dernière modification	VPC
Thu Nov 14 2024 10:03:32 GMT+0100 (heure normale d'Europe centrale)	Thu Nov 14 2024 10:09:15 GMT+0100 (heure normale d'Europe centrale)	vpc-03e7585ae2a245f36

Les tâches d'entraînement sont exécutées à partir des données stockées sur S3. Chaque tâche est nommée dynamiquement, comme les exemples lstm-training-job-1734470909 et autres variantes.

Les principales caractéristiques des tâches d'entraînement sont leurs durées, chaque job dure environ 3 minutes, ce qui garantit un réentraînement rapide. Nous utilisons aussi une instance ml.m5.large pour exécuter l'entraînement avec les ressources suffisantes pour notre charge de travail.

Amazon SageMaker AI

>

Tâches d'entraînement

Tâches d'entraînement

Infos

Actions

Créer une tâche d'entraînement

Rechercher tâches d'entraînement

Statut : Completed

Heure de création après : Dec 17, 2024 14:59 UTC

Clear Filters

< 1 >

	Nom	Heure de création	Durée	Statut de la tâche	Statut secondaire du travail	Statut du groupe d'instances pré-initialisées	Temps restant
<input type="radio"/>	lstm-training-job-1734470909-d85b4c67	17/12/2024 22:28:30	3 minutes	Completed	Completed	-	-
<input type="radio"/>	lstm-training-job-1734470684-786f0cbd	17/12/2024 22:24:44	3 minutes	Completed	Completed	-	-
<input type="radio"/>	lstm-training-job-1734470455-f6ab06a5	17/12/2024 22:20:56	3 minutes	Completed	Completed	-	-
<input type="radio"/>	lstm-training-job-1734470259-6ac9227e	17/12/2024 22:17:40	3 minutes	Completed	Completed	-	-
<input type="radio"/>	lstm-training-job-1734470050-02797e9a	17/12/2024 22:14:11	3 minutes	Completed	Completed	-	-
<input type="radio"/>	lstm-training-job-1734468868-fac6117e	17/12/2024 21:54:29	3 minutes	Completed	Completed	-	-
<input type="radio"/>	lstm-training-job-1734459557-8df2de6c	17/12/2024 19:19:18	3 minutes	Completed	Completed	-	-

Les modèles entraînés sont exportés sous forme d'artefacts dans le répertoire /output du bucket S3. Chaque tâche produit un fichier compressé model.tar.gz qui contient le modèle prêt à être déployé. Par exemple :

Sortie
Artefact de modèle S3 s3://clemapbucket/output/lstm-training-job-1734470909-d85b4c67/output/model.tar.gz

Cet artefact est ensuite utilisé pour redéployer le modèle sur AWS Greengrass, assurant ainsi un cycle d'amélioration continue.

La configuration SageMaker permet une orchestration automatisée des tâches d'entraînement, un déploiement rapide des modèles grâce aux artefacts S3 et une gestion centralisée via le domaine SageMaker. Ce workflow garantit une scalabilité, une fiabilité et une amélioration continue des performances prédictives.

Problèmes rencontrés

Compatibilité armv7 32 bits avec les librairies de machine learning

Lors du déploiement initial sur l'Edge Device, nous avons rencontré des problèmes de compatibilité liés à l'architecture ARMv7 32 bits utilisée par le système d'exploitation par défaut. Cette architecture limitée a posé des contraintes majeures pour l'installation des librairies de machine learning modernes, notamment TensorFlow. En effet, TensorFlow ne propose pas de version stable et optimisée pour les systèmes 32 bits, ce qui a empêché le chargement et l'exécution du modèle .h5 sur l'appareil.

Pour résoudre ce problème, nous avons effectué une mise à niveau vers un OS en 64 bits en installant Raspberry Pi OS Lite ARMv8 64 bits. Cette nouvelle architecture a permis de prendre en charge les librairies nécessaires, notamment TensorFlow dans sa version compatible ARM64.







Après cette migration, nous avons réinstallé les dépendances et configuré l'environnement pour permettre l'exécution fluide des tâches de prédiction en temps réel. Cette solution a rendu l'Edge Device capable d'exécuter des modèles de machine learning tout en restant performant.

Alternative avec container

Pour contourner ce problème avec l'infrastructure 32 bits, nous avons tenté de déployer un conteneur Docker préconfiguré avec les librairies nécessaires. Cependant, cette approche a également échoué en raison des restrictions inhérentes à l'infrastructure 32 bits, qui limitent la prise en charge des environnements conteneurisés complexes.

Ces difficultés techniques ont eu des répercussions directes sur notre capacité à déployer l'ensemble des composants sur AWS Greengrass dans les délais prévus. L'absence d'un environnement fonctionnel a empêché l'exécution locale des modèles de machine learning et perturbé plusieurs itérations de tests et d'améliorations du workflow. Finalement, pour résoudre ces limitations, nous avons procédé à une mise à niveau vers un OS ARMv8 64 bits (Raspberry Pi OS Lite), ce qui a permis de restaurer la compatibilité avec les librairies TensorFlow et d'assurer une intégration complète avec AWS Greengrass.


Nous avons donc aussi utilisé le stockage ERC d'AWS pour stocké des containers :

Images (4)							 Supprimer Détails Scanner Afficher les commandes Push	
<input type="text" value="Rechercher des artefacts"/>							< 1 > 	
<input type="checkbox"/>	Balise d'image ▼	Type d'artefact	Transmis à ▼	Taille (Mo) ▼	URI de l'image	Digest		
<input type="checkbox"/>	latest	Image	12 décembre 2024, 13:54:17 (UTC+01)	603.86	 Copier l'URI	sha256:d3a13fc5f2cb1de...		
<input type="checkbox"/>	-	Image	12 décembre 2024, 09:56:42 (UTC+01)	104.71	 Copier l'URI	sha256:7fb3b8aa68941c...		
<input type="checkbox"/>	-	Image	12 décembre 2024, 09:20:03 (UTC+01)	104.71	 Copier l'URI	sha256:ee01824cb12bd7...		
<input type="checkbox"/>	-	Image	12 décembre 2024, 09:08:42 (UTC+01)	104.71	 Copier l'URI	sha256:789655339e1e24...		

[AWS IoT](#) > [Greengrass](#) > [Déploiements](#) > Docker_deployment


Docker_deployment

Dernière révision: 33 ▼ Annuler Actions ▼

 Ce déploiement cible un groupe d'objets AWS IoT. Ajoutez un appareil principal au groupe d'objets pour lui appliquer ce déploiement.

Présentation

AWS IoT Greengrass utilise des tâches AWS IoT continues pour déployer des groupes d'objets.

Cible GreengrassQuickStartGroup	Type de cible Groupe d'objets	Déploiement créé il y a 10 jours
Tâche IoT d1623699-a18d-43e9-a839-b361fcec70b4	Statut du déploiement  Actif	

Suite à la mise à niveau du système d'exploitation vers ARMv8 64 bits, les limitations liées à l'architecture ont été levées. Désormais, il est possible d'utiliser des images Docker optimisées pour ARM64 incluant les bibliothèques nécessaires telles que TensorFlow. Cette solution présente plusieurs avantages :

Portabilité : Le modèle et ses dépendances peuvent être encapsulés dans une image unique, simplifiant le déploiement sur plusieurs appareils Edge.

Facilité de maintenance : Les mises à jour du modèle ou des bibliothèques peuvent être effectuées en reconstruisant et poussant une nouvelle image vers le registre.

Compatibilité avec greengrass : AWS Greengrass peut orchestrer les déploiements des conteneurs depuis le registre Docker directement vers les Edge Devices.

Ainsi, avec l'infrastructure mise à jour, le déploiement des modèles de machine learning via des conteneurs Docker devient une alternative viable et scalable pour notre solution, permettant une meilleure automatisation et gestion des déploiements sur les appareils connectés.

Conclusion

Ce projet avait pour objectif de prédire la consommation électrique en temps réel grâce à un capteur CLEMAP et un modèle de machine learning déployé localement sur un Edge Device. Tout au long du développement, nous avons rencontré plusieurs défis techniques, notamment des problèmes de compatibilité avec l'architecture ARMv7 32 bits qui a empêché l'installation des bibliothèques nécessaires comme TensorFlow. Ces limitations ont également rendu impossible le déploiement de conteneurs Docker au départ, ce qui a ralenti l'intégration avec AWS IoT Greengrass.

Cependant, en migrant vers un système d'exploitation ARMv8 64 bits, nous avons réussi à résoudre ces contraintes. Cela a permis non seulement de faire fonctionner le modèle TensorFlow en local, mais également d'intégrer un workflow complet et fiable. Les données du capteur CLEMAP sont collectées, traitées et analysées en temps réel, tandis que les erreurs de prédiction déclenchent un réentraînement automatique du modèle grâce à AWS SageMaker. Les rapports sont ensuite envoyés vers Amazon S3 via AWS IoT Greengrass Stream Manager, assurant une boucle d'amélioration continue.

AWS Greengrass s'est révélé être une solution idéale pour ce type de cas d'usage, offrant à la fois la puissance du cloud et des capacités de traitement local. Il garantit une faible latence, une résilience aux interruptions réseau et une grande flexibilité pour gérer les déploiements à l'échelle. Grâce à cette infrastructure, nous avons pu mettre en place un système intelligent et autonome, capable de prédire efficacement la consommation électrique tout en s'adaptant aux conditions réelles du terrain.

Les scripts déployés sont disponibles sur ce git : https://github.com/pleopleo6/IoT_AWS_Clemap

Références

Documentation Clemap : <https://fr.clemap.com/>

AWS workshop : <https://catalog.workshops.aws/aws-iot-immersionday-workshop/en-US>

Raspberry Pi : <https://www.raspberrypi.com/software/operating-systems/>