

Deploy ML models SageMaker

deploy machine learning models in aws
sagemaker

Got it 👍 Let's go step by step on **deploying a machine learning model in AWS SageMaker**. I'll break it into **two workflows**:

- **Workflow 1:** Using a **built-in SageMaker algorithm / prebuilt container**
- **Workflow 2:** Deploying your own custom model (local or Hugging Face / PyTorch / TensorFlow)

Workflow 1: Deploy a Model Using SageMaker Built-In Support

This is easiest if you trained your model in SageMaker or want to use built-in algorithms.

Steps:

1. **Setup IAM Role**
 - Give your SageMaker execution role access to **S3, SageMaker, and ECR**.
2. **Upload Training Data to S3**

```
bash

aws s3 cp data.csv s3://my-bucket/data/
```

3. **Train a Model (Optional, if not already trained)**

Example: XGBoost built-in algorithm

```
python

import sagemaker
from sagemaker import Session
from sagemaker.amazon.amazon_estimator import
get_image_uri

session = sagemaker.Session()
role = sagemaker.get_execution_role()

container = get_image_uri(session.boto_region_name,
'xgboost')
```

```

estimator = sagemaker.estimator.Estimator(
    container,
    role,
    instance_count=1,
    instance_type='ml.m5.large',
    output_path='s3://my-bucket/output',
    sagemaker_session=session
)

estimator.fit({'train': 's3://my-bucket/data/train.csv'})

```

4. Deploy the Model

```

python

predictor = estimator.deploy(
    initial_instance_count=1,
    instance_type='ml.m5.large'
)

```

5. Make Predictions

```

python

result = predictor.predict("1,2,3,4")
print(result)

```



Workflow 2: Deploy Your Own Model in SageMaker

This is useful if you trained your model locally or using Hugging Face, PyTorch, TensorFlow, etc.

Example: Deploying a Hugging Face Model

1. Upload Your Model to S3

```

bash

aws s3 cp model/ s3://my-bucket/model/ --recursive

```

2. Create a Model in SageMaker

```

python

from sagemaker.huggingface import HuggingFaceModel

```

```

role = sagemaker.get_execution_role()

huggingface_model = HuggingFaceModel(
    model_data="s3://my-bucket/model.tar.gz", # tar.gz of
your model
    role=role,
    transformers_version="4.26",
    pytorch_version="1.13",
    py_version="py39",
)

```

3. Deploy Endpoint

```

python

predictor = huggingface_model.deploy(
    initial_instance_count=1,
    instance_type="ml.m5.large"
)

```

4. Invoke Endpoint

```

python

response = predictor.predict({
    "inputs": "Translate this English text to French: 'I
love machine learning'"
})
print(response)

```

✓ Key Notes

- **Training vs Inference:** You can either train inside SageMaker or just deploy a pre-trained model.
- **Endpoints cost money** → stop them if not in use:

```

python

predictor.delete_endpoint()

```

- **Instance Types:** For small tests → `ml.m5.large`, for GPU models → `ml.g4dn.xlarge` or above.

👉 Do you want me to prepare a **step-by-step full example** for:

- A **local model you trained** (say scikit-learn / PyTorch), OR
- A **Hugging Face model from the Hub**, OR
- A **completely custom Docker container**?

i have.pkl file

Perfect 👍 since you already have a `.pkl` file (probably from **scikit-learn**, XGBoost, or another Python ML library), we'll use **SageMaker's** `SKLearnModel` or a custom inference script to deploy it.

Here's a **step-by-step guide**:

Deploy `.pkl` model to SageMaker

1. Prepare your `.pkl` file

Suppose your file is `model.pkl`.

Tar it (SageMaker requires `tar.gz` format):

```
bash

tar -czvf model.tar.gz model.pkl
```

2. Upload to S3

```
bash

aws s3 cp model.tar.gz s3://my-bucket/models/
```

3. Create an inference script (`inference.py`)

This defines how **SageMaker** should load your model and how to handle requests.

```
python

import joblib
import os
import json
```

```

def model_fn(model_dir):
    """Load model from SageMaker model directory"""
    model_path = os.path.join(model_dir, "model.pkl")
    return joblib.load(model_path)

def input_fn(request_body, request_content_type):
    """Deserialize input data"""
    if request_content_type == "application/json":
        return json.loads(request_body)
    elif request_content_type == "text/csv":
        return [float(x) for x in request_body.split(",")]
    else:
        raise ValueError(f"Unsupported content type: {request_content_type}")

def predict_fn(input_data, model):
    """Prediction logic"""
    prediction = model.predict([input_data])
    return prediction.tolist()

def output_fn(prediction, response_content_type):
    """Serialize prediction output"""
    return json.dumps({"prediction": prediction})

```

4. Create SageMaker Model

Use the `SKLearnModel` class:

```

python

import sagemaker
from sagemaker.sklearn.model import SKLearnModel

role = sagemaker.get_execution_role()
session = sagemaker.Session()

sklearn_model = SKLearnModel(
    model_data="s3://my-bucket/models/model.tar.gz",
    role=role,
    entry_point="inference.py",  # your inference script
    framework_version="1.2-1",  # pick sklearn version you
used

```

```
sagemaker_session=session  
)
```

5. Deploy Endpoint

```
python  
  
predictor = sklearn_model.deploy(  
    initial_instance_count=1,  
    instance_type="ml.m5.large"  
)
```

6. Invoke Endpoint

```
python  
  
# Example with JSON  
response = predictor.predict([5.1, 3.5, 1.4, 0.2])  
print(response)
```

7. Clean Up

```
python  
  
predictor.delete_endpoint()
```

✅ Now your `.pkl` model is deployed and you can send predictions via SageMaker endpoint.

👉 Do you want me to give you the **same flow but with Docker + custom container** (in case your `.pkl` uses non-standard libs like CatBoost/LightGBM), or just stick with this **SKLearnModel** approach?

if i have pkl then why it is model.tar.gz

Great question 👍

In SageMaker, **the model artifacts must be packaged into a single `.tar.gz` file** (not just the raw `.pkl`) before uploading to S3.

Here's **why**:

- When you deploy a model, SageMaker **downloads your model from S3 → unpacks it into** `/opt/ml/model/` inside the container.
- It expects a **single archive file** (`.tar.gz`) instead of loose files.
- Inside that archive, you can have `model.pkl` , `inference.py` , and any other files needed.

✅ Example: Packaging a `.pkl` model

If you only have `model.pkl` :

```
bash

tar -czvf model.tar.gz model.pkl
```

Now upload:

```
bash

aws s3 cp model.tar.gz s3://my-bucket/models/
```

✅ If you also need a custom inference script

Put both files together:

```
model/
├─ model.pkl
└─ inference.py
```

Then compress:

```
bash

tar -czvf model.tar.gz model/
```

✅ How SageMaker sees it

When deployed, SageMaker will untar it into:

```
swift

/opt/ml/model/
├─ model.pkl
└─ inference.py
```

Then your `model_fn` inside `inference.py` will load `model.pkl`.

🔑 So:

- `.pkl` = your trained model
- `.tar.gz` = SageMaker packaging requirement for deployment

Do you want me to show you a **minimal working repo structure** (with `model.pkl` + `inference.py` + `requirements.txt`) so you can directly deploy your `.pkl` file in SageMaker?

why can't we load model directly from pickle

```
import pickle
import numpy as np
```

```
# Load model at startup
with open("model.pkl", "rb") as f:
    model = pickle.load(f)
```

```
features = np.array(request.data).reshape(1,
-1)
pred = model.predict(features)[0]
prob = model.predict_proba(features)
[0].tolist()
```

Ah I see where you're coming from 👍 — in a **normal**

Python/Flask/FastAPI app, your snippet works fine because you can just keep a `model.pkl` in the project folder and load it with `pickle.load`.

But in **AWS SageMaker**, things work a bit differently:

🔑 Why we can't just load `model.pkl` directly

1. SageMaker stores models in S3

- When you call `model.deploy(...)`, SageMaker expects `model_data="s3://.../model.tar.gz"`.
- It downloads that file into the container at `/opt/ml/model/`.

2. The container expects an archive

- SageMaker pulls **one file** (`model.tar.gz`) from S3 and extracts it.
- It does **not** fetch individual `.pkl` files directly.

3. Consistency for all frameworks

- Same mechanism works for TensorFlow `.pb`, PyTorch `.pt`, Hugging Face models, scikit-learn `.pkl`, etc.
- That's why AWS enforces `.tar.gz` packaging.

✅ How it works inside SageMaker

Once SageMaker extracts your tarball into `/opt/ml/model/`, your code *can* look exactly like your snippet:

```
python

import joblib
import numpy as np
import os

def model_fn(model_dir):
    # SageMaker puts model files here after untar
    model_path = os.path.join(model_dir, "model.pkl")
    with open(model_path, "rb") as f:
        model = joblib.load(f)
    return model

def predict_fn(input_data, model):
    features = np.array(input_data).reshape(1, -1)
    pred = model.predict(features)[0]
    prob = model.predict_proba(features)[0].tolist()
    return {"prediction": int(pred), "probability": prob}
```

So — you **still load with pickle/joblib**, but SageMaker makes you wrap it inside `model_fn` and put it in a `.tar.gz`.

📌 **Summary:**

- You *can* use your `pickle.load` approach inside SageMaker.
- The only difference is that SageMaker requires you to package `model.pkl` into a `.tar.gz` and place it in S3 so it knows how to fetch and extract it into `/opt/ml/model/`.

👉 Do you want me to give you a **full working minimal FastAPI-style inference script** for SageMaker that loads your `.pkl` and responds to requests, so it looks exactly like your local code but SageMaker-compatible?

yes

Perfect ✅ Let's make a **minimal working SageMaker inference setup** that looks like your local `pickle` code but is fully SageMaker-compatible.

📁 Project Structure

```
perl
```

```
my-model/  
├─ model.pkl  
└─ inference.py
```

📄 `inference.py` (SageMaker entry point)

```
python
```

```
import os  
import pickle  
import numpy as np  
import json
```

```

# SageMaker will call this to load your model
def model_fn(model_dir):
    model_path = os.path.join(model_dir, "model.pkl")
    with open(model_path, "rb") as f:
        model = pickle.load(f)
    return model

# Handles input from the request
def input_fn(request_body, request_content_type):
    if request_content_type == "application/json":
        data = json.loads(request_body)
        return np.array(data).reshape(1, -1)
    elif request_content_type == "text/csv":
        return np.array([float(x) for x in
request_body.split(",")]).reshape(1, -1)
    else:
        raise ValueError(f"Unsupported content type:
{request_content_type}")

# Runs prediction
def predict_fn(input_data, model):
    pred = model.predict(input_data)[0]
    prob = model.predict_proba(input_data)[0].tolist()
    return {"prediction": int(pred), "probability": prob}

# Converts output to response
def output_fn(prediction, response_content_type):
    return json.dumps(prediction)

```



Package Model

1. Tar the folder:

```

bash

tar -czvf model.tar.gz model.pkl inference.py

```

2. Upload to S3:

```

bash

aws s3 cp model.tar.gz s3://my-bucket/models/

```



Deploy in SageMaker

```
python

import sagemaker
from sagemaker.sklearn.model import SKLearnModel

role = sagemaker.get_execution_role()
session = sagemaker.Session()

# Point to your packaged tar.gz
sklearn_model = SKLearnModel(
    model_data="s3://my-bucket/models/model.tar.gz",
    role=role,
    entry_point="inference.py",  # must match the script
    # inside tar
    framework_version="1.2-1",
    sagemaker_session=session
)

# Deploy endpoint
predictor = sklearn_model.deploy(
    initial_instance_count=1,
    instance_type="ml.m5.large"
)

# Test prediction
print(predictor.predict([5.1, 3.5, 1.4, 0.2]))
```



Cleanup

```
python

predictor.delete_endpoint()
```

✅ Now this behaves like your local `pickle` snippet but runs in SageMaker.

Would you like me to also show you how to **invoke this endpoint from outside SageMaker** (e.g., using Python `boto3` or `curl`), so you can hit it like a normal REST API?

```
predictor.predict how it will call to
inference.py
```

Excellent question 🚀 — let's peel back the curtain on what happens when you call

```
python

predictor.predict([5.1, 3.5, 1.4, 0.2])
```

in SageMaker.

Flow of a Prediction in SageMaker

1. Your call

```
python

predictor.predict([5.1, 3.5, 1.4, 0.2])
```

→ This sends the request to the deployed **SageMaker HTTPS endpoint** (Amazon API Gateway + a container running inside SageMaker Hosting).

2. Inside the container

SageMaker automatically sets up a **model server** (based on your framework, e.g., `sagemaker-scikit-learn-container`).

That server looks for four functions in your `inference.py` (entry point script):

- `model_fn(model_dir)` → loads your model once at startup
- `input_fn(request_body, request_content_type)` → preprocesses input
- `predict_fn(input_data, model)` → runs the model
- `output_fn(prediction, response_content_type)` → formats response

3. How the call maps

- SageMaker serializes your input (`[5.1, 3.5, 1.4, 0.2]`) into JSON (`"[5.1, 3.5, 1.4, 0.2]"`)
- Sends it to the endpoint with header `Content-Type: application/json`

- Container calls:

```
python

# (done once at container startup)
model = model_fn("/opt/ml/model") # Loads model.pkl

# for each request
data = input_fn(request_body, "application/json")
prediction = predict_fn(data, model)
response = output_fn(prediction, "application/json")
```

- The `response` is then sent back to your notebook/Python client.

4. You get the result

Example:

```
json

{"prediction": 0, "probability": [0.9, 0.1]}
```


Key Insight

`predictor.predict(...)` **does not directly call** `inference.py` like a normal Python import.


Instead:

- It sends an HTTPS request → SageMaker hosting container
- The container runs your script's functions (`model_fn`, `input_fn`, `predict_fn`, `output_fn`) in the right order.

That's why SageMaker forces you to wrap your logic inside those four functions rather than just `pickle.load()` and `model.predict()` directly.

 Do you want me to diagram this request → container → inference script flow (step-by-step arrows) so it's crystal clear how `predictor.predict` hits your `inference.py` ?

`model_fn`, `input_fn`, `predict_fn`, `output_fn` are predefined functions?

Yes  — great catch.

In **AWS SageMaker**, those functions are **not built into Python itself**, but they are **predefined entry points** that the **SageMaker model server** looks for inside your `inference.py`.

Think of them as **hooks (callbacks)** that SageMaker calls automatically when requests come into your endpoint.

What each function does

Function	When it's called	Purpose
<code>model_fn(model_dir)</code>	Once, when the container starts	Load your model from <code>/opt/ml/model/</code> . This is where you <code>pickle.load</code> or <code>joblib.load</code> .
<code>input_fn(request_body, request_content_type)</code>	For every request	Convert raw request (<code>JSON</code> , <code>CSV</code> , etc.) into a NumPy array / tensor your model can use.
<code>predict_fn(input_data, model)</code>	For every request	Run inference (<code>model.predict(...)</code>).
<code>output_fn(prediction, response_content_type)</code>	For every request	Convert your prediction into JSON/CSV/whatever format for the HTTP response.

Example flow (SageMaker calling these functions)

```
python


# at container startup
model = model_fn("/opt/ml/model")

# when a request comes in
raw_body = "[5.1, 3.5, 1.4, 0.2]" # from
predictor.predict(...)
input_data = input_fn(raw_body, "application/json")
```


```
prediction = predict_fn(input_data, model)
response = output_fn(prediction, "application/json")
```

Important notes


- You **don't have to define all four functions**.
 - If you only provide `model_fn`, SageMaker uses **default implementations** for the others (for JSON/NumPy).
 - But if your model needs custom preprocessing or postprocessing, you override them.
- They **exist only inside your** `inference.py`.
They aren't part of Python or scikit-learn — SageMaker just looks for these names when loading the script.

 So in short:

- **Yes, they are “predefined hooks” by SageMaker**, not built-in Python functions.
- **You implement them** in your script.
- SageMaker Hosting calls them automatically in the right order whenever you hit your endpoint.

 Do you want me to show you what the **default SageMaker implementations** of these functions look like (so you can see what happens if you don't override them)?

give all predefined hooks in sagemaker to write inference.py

Great question  — let's go beyond just `model_fn`, `input_fn`, `predict_fn`, and `output_fn`.

When you write a **custom** `inference.py` **script** in SageMaker, the framework containers (scikit-learn, PyTorch, TensorFlow, Hugging Face, etc.) look for certain **predefined hooks** that you *may* implement.

Here's the full list of hooks you can use:

Predefined Inference Script Hooks in SageMaker

1. `model_fn(model_dir)`

- **Purpose:** Load the model from `/opt/ml/model` (where SageMaker unpacks `model.tar.gz`).
- **Called:** Once, when the container starts.
- **Example:**

```
python

def model_fn(model_dir):
    import pickle, os
    with open(os.path.join(model_dir, "model.pkl"), "rb")
    as f:
        return pickle.load(f)
```

2. `input_fn(request_body, request_content_type)`

- **Purpose:** Deserialize and preprocess the request.
- **Called:** Every request, right after the raw HTTP body is received.
- **Example:**

```
python

def input_fn(request_body, request_content_type):
    if request_content_type == "application/json":
        import json, numpy as np
        data = json.loads(request_body)
        return np.array(data).reshape(1, -1)
    elif request_content_type == "text/csv":
        return [float(x) for x in request_body.split(",")]
    else:
        raise ValueError(f"Unsupported content type:
{request_content_type}")
```

3. `predict_fn(input_data, model)`

- **Purpose:** Run inference logic.
- **Called:** After preprocessing, with the model already loaded.

- **Example:**

python

```
def predict_fn(input_data, model):
    pred = model.predict(input_data)[0]
    prob = model.predict_proba(input_data)[0].tolist()
    return {"prediction": int(pred), "probability": prob}
```

4. `output_fn(prediction, response_content_type)`

- **Purpose:** Serialize the response.
- **Called:** Every request, right before sending HTTP response.
- **Example:**

python

```
def output_fn(prediction, response_content_type):
    import json
    return json.dumps(prediction)
```

5. `transform_fn(model, request_body, request_content_type, response_content_type)`

◆ *(Optional, advanced hook)*

- Lets you **combine all steps into one function** instead of implementing `input_fn`, `predict_fn`, and `output_fn` separately.
- If `transform_fn` is present, SageMaker **ignores** the other three.
- **Example:**

python

```
def transform_fn(model, request_body, request_content_type, response_content_type):
    import json
    data = json.loads(request_body)
    pred = model.predict([data])[0]
    return json.dumps({"prediction": int(pred)})
```

Optional Training-Time Hooks (only if you use same script for training + inference)

If you use a **script mode container** (PyTorch, TensorFlow, Hugging Face, SKLearn), you can also define:


- `train(args)` → Training loop
- `model_fn(model_dir)` → Load trained model (used in inference)
- `save_model(model, model_dir)` → Save model after training

But if you're only serving `.pkl` for inference → you only care about the **inference hooks** above.

Summary

For **inference.py**, SageMaker recognizes these hooks:

1. `model_fn(model_dir)` → load model
 2. `input_fn(request_body, request_content_type)` → preprocess request
 3. `predict_fn(input_data, model)` → run inference
 4. `output_fn(prediction, response_content_type)` → postprocess response
 5. `transform_fn(model, request_body, request_content_type, response_content_type)` → (optional) replaces 2–4
-

 Do you want me to build you a **template inference.py** that includes **all hooks** (with defaults), so you can just plug in your `.pkl` model?

same now `.onnx` model

Got it  You now want the **same SageMaker inference flow**, but for a `.onnx` **model** instead of `.pkl`.

With ONNX models, you typically use **ONNX Runtime** (`onnxruntime`) inside `inference.py`.

The hooks are the same (`model_fn`, `input_fn`, `predict_fn`, `output_fn`), only the **loading and inference code changes**.

Project Structure

perl

```
my-onnx-model/  
├─ model.onnx  
└─ inference.py
```



inference.py for ONNX

python

```
import os  
import json  
import numpy as np  
import onnxruntime as ort  
  
# Called once when the container starts  
def model_fn(model_dir):  
    model_path = os.path.join(model_dir, "model.onnx")  
    # Load ONNX model with ONNX Runtime  
    session = ort.InferenceSession(model_path, providers=  
["CPUExecutionProvider"])  
    return session  
  
# Parse incoming request  
def input_fn(request_body, request_content_type):  
    if request_content_type == "application/json":  
        data = json.loads(request_body)  
        return np.array(data, dtype=np.float32).reshape(1, -1)  
    elif request_content_type == "text/csv":  
        return np.array([float(x) for x in  
request_body.split(",")], dtype=np.float32).reshape(1, -1)  
    else:  
        raise ValueError(f"Unsupported content type:  
{request_content_type}")  
  
# Run prediction  
def predict_fn(input_data, model):  
    input_name = model.get_inputs()[0].name  
    outputs = model.run(None, {input_name: input_data})  
    # Assume first output is prediction scores  
    prediction = outputs[0]  
    return prediction.tolist()
```

```
# Format output for response
def output_fn(prediction, response_content_type):
    return json.dumps({"prediction": prediction})
```



Package Model

```
bash

tar -czvf model.tar.gz model.onnx inference.py
aws s3 cp model.tar.gz s3://my-bucket/models/
```



Deploy in SageMaker

```
python

import sagemaker
from sagemaker.model import Model

role = sagemaker.get_execution_role()
session = sagemaker.Session()

onnx_model = Model(
    model_data="s3://my-bucket/models/model.tar.gz",
    role=role,
    entry_point="inference.py",
    image_uri=sagemaker.image_uris.retrieve(
        framework="sklearn", # or "pytorch" if you prefer
        region=session.boto_region_name,
        version="1.2-1", # version only matters for base
        container
        py_version="py3"
    ),
    sagemaker_session=session
)

predictor = onnx_model.deploy(
    initial_instance_count=1,
    instance_type="ml.m5.large"
)
```

```
print(predictor.predict([5.1, 3.5, 1.4, 0.2]))
```

✓ Key Notes

- `onnxruntime` must be available in the container.
 - You can install it via `requirements.txt` or extend the base container.
 - Example `requirements.txt` :

```
nginx  
  
onnxruntime  
numpy
```

- If you're using **deep learning models** (PyTorch/TensorFlow exported to ONNX), the same code works — just make sure to pass the correct input shape.

👉 Do you want me to also show you a **custom Dockerfile** with `onnxruntime` pre-installed (instead of relying on SKLearn/PyTorch base images)? That's usually cleaner for `.onnx` deployments.

`.tf` model is there now?

Perfect 👍 now let's do the same flow for a **TensorFlow SavedModel** (`.tf` model / **SavedModel** folder) in SageMaker.

TensorFlow models are usually exported in **SavedModel** format (a directory with `saved_model.pb` and `variables/`). SageMaker has **native support** for TensorFlow models, so it's even easier than `.pkl` or `.onnx`.

📁 Project Structure

```
kotlin  
  
my-tf-model/  
└─ saved_model.pb
```

```
└─ variables/
   └─ variables.index
      └─ variables.data-00000-of-00001
```



Package Model

SageMaker requires a `.tar.gz`:

```
bash

tar -czvf model.tar.gz saved_model.pb variables/
aws s3 cp model.tar.gz s3://my-bucket/models/
```



Inference Script (`inference.py`)

If you only want **raw TensorFlow inference** (no custom pre/post-processing), you don't even need an `inference.py` — SageMaker's TensorFlow container can serve the model directly.

But if you want custom preprocessing (like with `.pkl` or `.onnx`), then add an `inference.py` with hooks.

```
python

import os
import json
import numpy as np
import tensorflow as tf

# Load TensorFlow SavedModel
def model_fn(model_dir):
    model = tf.keras.models.load_model(model_dir)
    return model

# Preprocess input
def input_fn(request_body, request_content_type):
    if request_content_type == "application/json":
        data = json.loads(request_body)
        return np.array(data, dtype=np.float32)
    elif request_content_type == "text/csv":
        return np.array([float(x) for x in
            request_body.split(",")], dtype=np.float32).reshape(1, -1)
```

```

        else:
            raise ValueError(f"Unsupported content type:
{request_content_type}")

# Run prediction
def predict_fn(input_data, model):
    prediction = model.predict(input_data)
    return prediction.tolist()

# Format response
def output_fn(prediction, response_content_type):
    return json.dumps({"prediction": prediction})

```

Deploy in SageMaker

```

python

import sagemaker
from sagemaker.tensorflow import TensorFlowModel

role = sagemaker.get_execution_role()
session = sagemaker.Session()

tf_model = TensorFlowModel(
    model_data="s3://my-bucket/models/model.tar.gz",
    role=role,
    entry_point="inference.py", # optional; omit if no
preprocessing needed
    framework_version="2.13", # pick your TF version
    sagemaker_session=session
)

predictor = tf_model.deploy(
    initial_instance_count=1,
    instance_type="ml.m5.large"
)

print(predictor.predict([[5.1, 3.5, 1.4, 0.2]]))

```



Key Notes

- If your model is **pure TensorFlow SavedModel**, you can skip `inference.py` and SageMaker will still serve it.
- If you want **custom preprocessing/postprocessing**, add `inference.py` with the hooks (just like `.pkl` and `.onnx`).
- For **Keras .h5 models**, you must first convert them to **SavedModel** (`model.save("my-tf-model")`).

👉 Do you want me to also show you the **training + deployment flow** in SageMaker for TensorFlow (train inside SageMaker → auto-save → deploy), or just stick to inference from a pre-saved `.tf` model?

give examples for all below types separately for all

.pkl
 .joblib
 .h5, .hdf5
 .onnx
 .pt, .pth, .pth.tar
 .ckpt, .data, .index
 .tflite
 .mlmodel
 .caffemodel, .prototxt
 .pb
 .safetensors

Great question 👉 — you want **separate SageMaker inference examples** for each model format (`.pkl`, `.joblib`, `.h5`, `.onnx`, `.pt`, `.ckpt`, `.tflite`, `.mlmodel`, `.caffemodel`, `.pb`, `.safetensors`).

👉 In SageMaker, we usually create a `model.tar.gz` (containing the model + inference script). The inference script (`inference.py`) uses the **predefined hooks**:

- `model_fn(model_dir)` → load model from file
- `input_fn(request_body, request_content_type)` → parse input
- `predict_fn(input_data, model)` → run inference
- `output_fn(prediction, content_type)` → format response

I'll now give **examples for each type**.

1. Scikit-learn / XGBoost → .pkl

```
python

# inference.py
import pickle
import numpy as np

def model_fn(model_dir):
    with open(f"{model_dir}/model.pkl", "rb") as f:
        return pickle.load(f)

def input_fn(request_body, request_content_type):
    data = np.array(eval(request_body)).reshape(1, -1)
    return data

def predict_fn(input_data, model):
    return {"prediction": model.predict(input_data).tolist(),
            "proba": model.predict_proba(input_data).tolist()}

def output_fn(prediction, response_content_type):
    return str(prediction)
```

2. Scikit-learn → .joblib

```
python

import joblib
import numpy as np

def model_fn(model_dir):
    return joblib.load(f"{model_dir}/model.joblib")
```

3. Keras / TensorFlow → .h5, .hdf5

```
python

from tensorflow.keras.models import load_model
import numpy as np

def model_fn(model_dir):
```

```

    return load_model(f"{model_dir}/model.h5")

def predict_fn(input_data, model):
    return model.predict(input_data).tolist()

```

4. ONNX → .onnx

```

python

import onnxruntime as ort
import numpy as np

def model_fn(model_dir):
    return ort.InferenceSession(f"{model_dir}/model.onnx")

def predict_fn(input_data, session):
    inputs = {session.get_inputs()[0].name:
input_data.astype(np.float32)}
    return session.run(None, inputs)

```

5. PyTorch → .pt, .pth, .pth.tar

```

python

import torch

def model_fn(model_dir):
    model = torch.load(f"{model_dir}/model.pth",
map_location="cpu")
    model.eval()
    return model

def predict_fn(input_data, model):
    with torch.no_grad():
        return
model(torch.tensor(input_data)).numpy().tolist()

```

6. TensorFlow Checkpoints → .ckpt, .data, .index

```

python

```

```
import tensorflow as tf

def model_fn(model_dir):
    model = tf.train.Checkpoint()
    ckpt = tf.train.Checkpoint(model=model)

    ckpt.restore(tf.train.latest_checkpoint(model_dir)).expect_partial()
    return model
```

7. TensorFlow Lite → .tflite

python

```
import tensorflow as tf
import numpy as np

def model_fn(model_dir):
    interpreter = tf.lite.Interpreter(model_path=f"{model_dir}/model.tflite")
    interpreter.allocate_tensors()
    return interpreter

def predict_fn(input_data, interpreter):
    input_details = interpreter.get_input_details()
    output_details = interpreter.get_output_details()
    interpreter.set_tensor(input_details[0]['index'],
                           input_data.astype(np.float32))
    interpreter.invoke()
    return interpreter.get_tensor(output_details[0]
                                  ['index']).tolist()
```

8. Apple CoreML → .mlmodel

python

```
import coremltools as ct

def model_fn(model_dir):
    return ct.models.MLModel(f"{model_dir}/model.mlmodel")
```

```
def predict_fn(input_data, model):
    return model.predict({"input": input_data})
```

9. Caffe → .caffemodel, .prototxt

```
python

import caffe
import numpy as np

def model_fn(model_dir):
    net = caffe.Net(f"{model_dir}/deploy.prototxt",
                    f"{model_dir}/model.caffemodel",
                    caffe.TEST)

    return net

def predict_fn(input_data, net):
    net.blobs['data'].data[...] = input_data
    output = net.forward()
    return output
```

10. TensorFlow Frozen Graph → .pb

```
python

import tensorflow as tf

def model_fn(model_dir):
    model = tf.saved_model.load(model_dir) # expects
    saved_model.pb
    return model

def predict_fn(input_data, model):
    infer = model.signatures["serving_default"]
    return infer(tf.constant(input_data)).numpy().tolist()
```

11. Hugging Face / PyTorch → .safetensors

```
python
```

```

from transformers import AutoModel, AutoTokenizer
import torch

def model_fn(model_dir):
    tokenizer = AutoTokenizer.from_pretrained(model_dir)
    model = AutoModel.from_pretrained(model_dir,
    trust_remote_code=True)
    return (tokenizer, model)

def predict_fn(input_data, model_tuple):
    tokenizer, model = model_tuple
    inputs = tokenizer(input_data, return_tensors="pt")
    with torch.no_grad():
        outputs = model(**inputs)
    return outputs.last_hidden_state.tolist()

```

✓ Each of these scripts would be inside `inference.py` and bundled with the model in `model.tar.gz`.

✓ You deploy with `sagemaker.Model(..., entry_point="inference.py", ...)`.

Do you want me to also give you a **step-by-step packaging + deployment example** (upload `.pkl` → create `model.tar.gz` → deploy in SageMaker → run prediction)?

```

def input_fn(request_body,
request_content_type):
    data =
    np.array(eval(request_body)).reshape(1, -1)
    return data

```

```

def output_fn(prediction,
response_content_type):
    return str(prediction)

```

above functions same for all types

Great question 👍

The `input_fn` and `output_fn` in SageMaker are **content-type dependent**, not **model-type dependent**.

That means:

- If your model is `.pkl`, `.joblib`, `.onnx`, `.h5`, `.pt`, etc., the **loading and prediction logic** changes in `model_fn` and `predict_fn`.
- But `input_fn` (how you parse incoming request) and `output_fn` (how you format outgoing response) can often stay the **same**, provided your API accepts **JSON arrays / CSV / raw NumPy arrays** as input and outputs a simple number/list.

✅ Example – Common `input_fn` and `output_fn`

```
python

import numpy as np
import json

# Convert request body → model input
def input_fn(request_body, request_content_type):
    if request_content_type == "application/json":
        data = json.loads(request_body)
        return np.array(data).reshape(1, -1)
    elif request_content_type == "text/csv":
        data = np.fromstring(request_body, sep=",")
        return data.reshape(1, -1)
    else:
        raise ValueError(f"Unsupported content type: {request_content_type}")

# Convert model prediction → response
def output_fn(prediction, response_content_type):
    if response_content_type == "application/json":
        return json.dumps(prediction.tolist())
    elif response_content_type == "text/plain":
        return str(prediction)
    else:
        raise ValueError(f"Unsupported response type: {response_content_type}")
```

**Key Point:**

- These two functions (`input_fn` , `output_fn`) **don't change much across model types**.
- What **does** change is:
 - `model_fn` → how you **load** (`.pkl` , `.pt` , `.onnx` , etc.)
 - `predict_fn` → how you **call inference** (e.g., `model.predict()` , `session.run()` , `torch_model(input)` , etc.)

So, yes — the version you wrote works **for all model types**, as long as:

- Your request body matches what your model expects (numbers, arrays, tensors).
- You reshape properly before passing to the model.



Do you want me to show you a **full inference.py template** with `model_fn` , `input_fn` , `predict_fn` , `output_fn` all together — for each format (`.pkl` , `.onnx` , `.h5` , `.pt` , etc.) separately?

how to deploy huggingface LLM models

Great question . Deploying Hugging Face LLMs (Large Language Models) on Amazon SageMaker can be done using the HuggingFace Inference DLCs (Deep Learning Containers) or with a custom `inference.py` if you want full control.

Let me break it down step by step for you



1. Using HuggingFace SageMaker DLC (Recommended for LLMs)

If your model is on Hugging Face Hub, you don't need to pickle, just point SageMaker to it:

```
python

from sagemaker.huggingface import HuggingFaceModel
import sagemaker
```



```

role = "arn:aws:iam::123456789012:role/service-
role/SageMakerExecutionRole"

# Hugging Face Model Class
huggingface_model = HuggingFaceModel(
    model_data=None, # if you want to load from S3, put the
tar.gz path
    role=role,
    transformers_version="4.38",
    pytorch_version="2.1",
    py_version="py310",
    env={
        "HF_MODEL_ID": "mistralai/Mistral-7B-Instruct-v0.2",
# HuggingFace Hub Model
        "SM_NUM_GPUS": "1", # Ensure GPU
    }
)

# Deploy
predictor = huggingface_model.deploy(
    initial_instance_count=1,
    instance_type="ml.g5.2xlarge"
)

# Query the endpoint
response = predictor.predict({
    "inputs": "What is the capital of France?"
})
print(response)

```

👉 Here, SageMaker automatically downloads the model from Hugging Face Hub.

👉 You just pass `HF_MODEL_ID`.

👉 No need for `inference.py` unless you want custom logic.

✅ 2. Deploy Your Own Local Model

If you fine-tuned and saved your model (`.pt` , `.bin` , `.safetensors` , etc.), first **upload it to S3**:

```

bash

aws s3 cp model.tar.gz s3://my-bucket/models/my-llm/

```

Your `model.tar.gz` should have this structure:

```
pgsql

model.tar.gz
├─ pytorch_model.bin
├─ config.json
├─ tokenizer.json
├─ tokenizer_config.json
├─ special_tokens_map.json
└─ generation_config.json
```

Then deploy:

```
python

huggingface_model = HuggingFaceModel(
    model_data="s3://my-bucket/models/my-llm/model.tar.gz",
    role=role,
    transformers_version="4.38",
    pytorch_version="2.1",
    py_version="py310",
)

predictor = huggingface_model.deploy(
    initial_instance_count=1,
    instance_type="ml.g5.2xlarge"
)

response = predictor.predict({"inputs": "Tell me a joke about
AI"})
print(response)
```

✓ 3. Deploy with Custom `inference.py`

If you want more control (e.g., custom pre/post-processing), you provide `inference.py`.

SageMaker expects these hooks:

```
python

# inference.py
from transformers import AutoModelForCausalLM, AutoTokenizer
import torch
```

```
def model_fn(model_dir):
    model = AutoModelForCausalLM.from_pretrained(model_dir)
    tokenizer = AutoTokenizer.from_pretrained(model_dir)
    return {"model": model, "tokenizer": tokenizer}

def input_fn(request_body, request_content_type):
    return request_body # raw text

def predict_fn(data, model_dict):
    model = model_dict["model"]
    tokenizer = model_dict["tokenizer"]

    inputs = tokenizer(data,
return_tensors="pt").to(model.device)
    outputs = model.generate(**inputs, max_new_tokens=100)
    return tokenizer.decode(outputs[0],
skip_special_tokens=True)

def output_fn(prediction, response_content_type):
    return prediction
```

Your **model.tar.gz** should include:

```
arduino

model.tar.gz
├─ inference.py
├─ pytorch_model.bin
├─ config.json
├─ tokenizer.json
└─ ...
```

✓ 4. Supported Model Types for HuggingFace DLC

- GPT-2 / GPT-NeoX / LLaMA / Falcon / Mistral
- Seq2Seq models (BART, T5, etc.)
- Embedding models (Sentence Transformers)
- Custom fine-tuned models

⚡ Summary

- If the model is on Hugging Face Hub → use DLC + `HF_MODEL_ID`.

- If local fine-tuned model → upload `tar.gz` to S3.
- If custom pre/post-processing → add `inference.py`.

👉 Do you want me to show you how to **deploy Falcon-7B-Instruct** as a working end-to-end example (with generation params like temperature, top_p, max_length)?

how to deploy ollama models

Good question 👍 Let's break it down.

💠 Ollama Models vs SageMaker

Ollama is **not natively supported on AWS SageMaker** like Hugging Face or TensorFlow/PyTorch models. Instead, Ollama provides its **own runtime** (locally or in a container) that loads and serves models (e.g., LLaMA, Mistral, Gemma).

To deploy Ollama models on AWS, you basically run the **Ollama server** inside a container (EC2, ECS, or EKS) and expose it via an API endpoint.

✅ Steps to Deploy Ollama Models on AWS

1. Setup EC2 Instance

- Choose a GPU EC2 instance (e.g., `g4dn.xlarge`, `g5.xlarge`, or larger depending on the model).
- Install Docker.

```
bash

sudo apt update && sudo apt install -y docker.io
sudo systemctl start docker
sudo systemctl enable docker
```

2. Run Ollama in Docker

Ollama has an official Docker image.

```
bash
```

```
docker run -d --gpus all -p 11434:11434 ollama/ollama
```

This starts the Ollama API server at `http://<EC2_IP>:11434`.

3. Pull a Model

Once the server is running, exec into the container and pull a model.

```
bash

docker exec -it <container_id> ollama pull llama2
```

You can replace `llama2` with `mistral`, `gemma`, `codellama`, etc.

4. Query the Model

From your EC2 or any client:

```
bash

curl http://<EC2_IP>:11434/api/generate -d '{
  "model": "llama2",
  "prompt": "Write a haiku about AWS SageMaker"
}'
```

5. Optional: Deploy with ECS/EKS

- Package the `ollama/ollama` Docker image into an **ECS task** or **Kubernetes pod**.
 - Use an **ALB (Application Load Balancer)** or **API Gateway** to expose it securely.
 - Attach IAM roles, scaling policies, etc.
-

6. Access from Python

```
python

import requests

url = "http://<EC2_IP>:11434/api/generate"
```

```
payload = {  
    "model": "llama2",  
    "prompt": "Explain Ollama deployment on AWS in 3 steps"  
}  
resp = requests.post(url, json=payload)  
print(resp.json())
```

⚡ Comparison with Hugging Face on SageMaker

- **Hugging Face models on SageMaker** → Direct SDK integration (`HuggingFaceModel`, `predictor`).
- **Ollama models** → Need to run Ollama runtime in a container (ECS/EKS/EC2), then expose REST API.

👉 Do you want me to give you a **ready-to-use AWS ECS Fargate deployment (serverless Ollama)**, or should I stick with **EC2 GPU deployment**?

ollama models deploy in sagemaker

Great question 👍

Currently, **Ollama** is designed as a **local runtime** for LLMs (like `llama2`, `mistral`, `gemma`, etc.) where you run them on your own server/laptop with a GPU. AWS SageMaker does **not** have native support for Ollama models (unlike Hugging Face or TensorFlow).

But—you can **still deploy Ollama models on SageMaker** by treating Ollama as a **custom container inference server**.

✅ Steps to Deploy Ollama Models on SageMaker

1. Build a Docker container with Ollama

Create a Dockerfile:

```
dockerfile
```

```
FROM nvidia/cuda:12.1.1-runtime-ubuntu22.04

# Install dependencies
RUN apt-get update && apt-get install -y curl unzip && rm -rf /var/lib/apt/lists/*

# Install Ollama
RUN curl -fsSL https://ollama.com/download/OllamaLinux.zip -o ollama.zip \
    && unzip ollama.zip -d /usr/local/bin \
    && rm ollama.zip

# Expose port
EXPOSE 11434

# Start Ollama server
CMD ["ollama", "serve"]
```

This container will run Ollama inside SageMaker.

2. Push the container to Amazon ECR

```
bash

aws ecr create-repository --repository-name ollama-sagemaker
docker build -t ollama-sagemaker .
docker tag ollama-sagemaker:latest <account_id>.dkr.ecr.
<region>.amazonaws.com/ollama-sagemaker:latest
docker push <account_id>.dkr.ecr.
<region>.amazonaws.com/ollama-sagemaker:latest
```

3. Create SageMaker Model

```
python

import boto3
from sagemaker.model import Model

role = "<your-sagemaker-execution-role>"

model = Model(
    image_uri="<account_id>.dkr.ecr.
```

```

<region>.amazonaws.com/ollama-sagemaker:latest",
    role=role
)

predictor = model.deploy(
    initial_instance_count=1,
    instance_type="ml.g4dn.xlarge", # GPU instance
    endpoint_name="ollama-endpoint"
)

```

4. Send requests to the endpoint

Since Ollama exposes an HTTP API (`localhost:11434/api/generate`), SageMaker will proxy requests:

```

python

import requests

endpoint_url = predictor.endpoint_name

response = requests.post(
    f"https://runtime.sagemaker.
    <region>.amazonaws.com/endpoints/{endpoint_url}/invocations",
    json={
        "model": "llama2",
        "prompt": "Write a poem about SageMaker and Ollama"
    }
)

print(response.json())

```

⚠ Challenges

- Ollama is **GPU-heavy**; use SageMaker GPU instances (`g4dn` , `g5` , `p4d`).
- Model downloading: by default, Ollama downloads models on first run. To avoid cold starts, **bake the `.ollama` model into the container** or preload them in `/root/.ollama` .
- Scaling: SageMaker can scale Ollama containers, but Ollama itself is single-node oriented.

👉 So in short:

You **wrap Ollama inside a custom container**, push to ECR, then deploy with SageMaker as a normal model.

Do you want me to **write a full working** `inference.py` **wrapper** that takes SageMaker requests and forwards them to Ollama inside the container?

how to deploy unsloth models

It looks like **Unsloth** doesn't have native integration with AWS SageMaker—so deploying its models there requires a workaround.

How to Deploy Unsloth Models on AWS SageMaker

Unsloth offers blazing-fast LLM fine-tuning and inference by leveraging custom GPU kernels and Triton back-ends [GitHub](#) [Unsloth](#) . However, SageMaker doesn't support Unsloth out-of-the-box.

You can still deploy Unsloth models on SageMaker by wrapping your Unsloth inference runtime into a custom Docker container and deploying that container as a SageMaker endpoint.

Step-by-Step Guide

1. Create a Docker Container for Unsloth

Write a `Dockerfile` like this:

Dockerfile

```
FROM nvidia/cuda:12.1.1-runtime-ubuntu22.04

RUN apt-get update && apt-get install -y build-essential curl
git python3 python3-pip && rm -rf /var/lib/apt/lists/*

# Install Unsloth and dependencies
RUN pip install unsloth torch triton

# Expose port for inference API
```

```
EXPOSE 8080
```

```
# Add your inference script
COPY inference.py /opt/program/inference.py

# Run your app (e.g., Flask)
CMD ["python3", "/opt/program/inference.py"]
```

2. Write `inference.py` to Serve Your Model

Use Flask (or FastAPI) to load the model using Unsloth's API and expose a REST endpoint:

```
python

from flask import Flask, request, jsonify
import torch
from unsloth import FastLanguageModel

app = Flask(__name__)

# Load your model at startup—adjust model_name as needed
model, tokenizer = FastLanguageModel.from_pretrained(
    model_name="unsloth/gemma-3-4b-instruct",
    load_in_4bit=True, fast_inference=True
)

@app.route("/invocations", methods=["POST"])
def invoke():
    prompt = request.json.get("prompt", "")
    max_length = request.json.get("max_length", 50)
    inputs = tokenizer(prompt,
return_tensors="pt").to(model.device)
    with torch.no_grad():
        outputs = model.generate(**inputs,
max_new_tokens=max_length)
        text = tokenizer.decode(outputs[0],
skip_special_tokens=True)
        return jsonify({"generated_text": text})

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8080)
```

3. Build and Push to ECR

```
bash
```

```

docker build -t unsloth-sagemaker .
docker tag unsloth-sagemaker:latest <account>.dkr.ecr.
<region>.amazonaws.com/unsloth-sagemaker:latest
docker push <account>.dkr.ecr.<region>.amazonaws.com/unsloth-
sagemaker:latest

```

4. Deploy as SageMaker Model

```

python

from sagemaker.model import Model
import sagemaker

role = sagemaker.get_execution_role()
session = sagemaker.Session()

model = Model(
    image_uri="<your-ecr-image>",
    role=role,
    sagemaker_session=session
)

predictor = model.deploy(
    initial_instance_count=1,
    instance_type="ml.g5.xlarge", # or other GPU-enabled
instance
    endpoint_name="unsloth-endpoint"
)

```

5. Consume the Endpoint

```

python

import requests
import sagemaker

endpoint_name = "unsloth-endpoint"
runtime = sagemaker.runtime

response = runtime.invoke_endpoint(
    EndpointName=endpoint_name,
    ContentType="application/json",
    Body='{"prompt": "Tell me a haiku about AI", "max_length":
50}'
)

```

```
)
print(response["Body"].read().decode())
```

Why This Works for Unsloth

Unsloth relies on GPU kernels and Triton-based acceleration—none of which are supported by SageMaker’s native serving containers. By embedding the Unsloth runtime into your own container and using the standard HTTP interface, you regain full control and compatibility.

Summary Table

Step	Description
Dockerize	Build container with Unsloth, Triton, and dependencies
Inference Script	Load model with Unsloth and serve HTTP endpoint via Flask
Push to ECR	Upload container to AWS Elastic Container Registry
SageMaker Deployment	Deploy image as endpoint using SageMaker <code>Model.deploy()</code>
Invocation	Send HTTP calls via SageMaker runtime or standard requests

Let me know if you’d like me to help further—e.g. adding multi-request batching, error handling, or autoscaling your endpoint!

 Sources

```
can't we do like

# inference.py
```

```

from transformers import AutoTokenizer,
AutoModelForCausalLM, pipeline
from peft import PeftModel

def model_fn(model_dir):
    base_model = "unsloth/mistral-7b-bnb-4bit"
    model =
AutoModelForCausalLM.from_pretrained(base
_model, load_in_4bit=True,
device_map="auto")
    model = PeftModel.from_pretrained(model,
model_dir)
    tokenizer =
AutoTokenizer.from_pretrained(base_model)
    return pipeline("text-generation",
model=model, tokenizer=tokenizer)

def predict_fn(input_data, model):
    prompt = input_data["prompt"]
    return model(prompt,
max_new_tokens=200)

```

You're on the right track 👍 — your `inference.py` structure with `model_fn` and `predict_fn` is valid for SageMaker custom containers, and it **can work for Unsloth fine-tuned models** with some small adjustments. Let me refine it:

Key points:

1. **Base model & PEFT adapter** — You load the quantized base (`unsloth/mistral-7b-bnb-4bit`) and then apply your fine-tuned adapter weights from `model_dir` . ✅ Good.
2. **Tokenizer** — Load from the base model (sometimes also from `model_dir` if your fine-tuning changed vocab).
3. **Pipeline vs. direct model** — Hugging Face `pipeline` works, but for LLMs in SageMaker it's often better to return the model & tokenizer and handle generation explicitly in `predict_fn` for more control (e.g., temperature, top_p, etc.).
4. **Input/output format** — Ensure JSON serialization, since SageMaker endpoints usually pass JSON request/response bodies.

Improved Example: `inference.py`

```
python

import torch
from transformers import AutoTokenizer, AutoModelForCausalLM
from peft import PeftModel

# 1. Load model + adapter
def model_fn(model_dir):
    base_model = "unsloth/mistral-7b-bnb-4bit"

    # Load quantized base model
    model = AutoModelForCausalLM.from_pretrained(
        base_model,
        load_in_4bit=True,
        device_map="auto",
        torch_dtype=torch.float16
    )

    # Load PEFT adapter from model_dir (your fine-tuned weights)
    model = PeftModel.from_pretrained(model, model_dir)

    # Load tokenizer
    tokenizer = AutoTokenizer.from_pretrained(base_model)

    return {"model": model, "tokenizer": tokenizer}

# 2. Handle prediction
def predict_fn(input_data, model_dict):
    model = model_dict["model"]
    tokenizer = model_dict["tokenizer"]

    prompt = input_data.get("prompt", "")

    inputs = tokenizer(prompt,
return_tensors="pt").to(model.device)
    outputs = model.generate(
        **inputs,
        max_new_tokens=200,
        temperature=0.7,
        top_p=0.9,
        do_sample=True,
    )
```

```

        generated_text = tokenizer.decode(outputs[0],
skip_special_tokens=True)
        return {"generated_text": generated_text}

# 3. Input function (for SageMaker JSON input)
def input_fn(request_body, request_content_type):
    if request_content_type == "application/json":
        import json
        return json.loads(request_body)
    else:
        raise ValueError(f"Unsupported content type:
{request_content_type}")

# 4. Output function (for SageMaker JSON output)
def output_fn(prediction, response_content_type):
    if response_content_type == "application/json":
        import json
        return json.dumps(prediction)
    else:
        return str(prediction)

```

✓ This way:

- SageMaker sends {"prompt": "Explain reinforcement learning in 2 lines"}
- Your endpoint returns {"generated_text": "Reinforcement learning is ..."}

Would you like me to also give you the `Dockerfile` + **serving entrypoint** so you can package this `inference.py` and deploy an Unsloth model on SageMaker from scratch?

inference.py is fine give complete deployment steps

Got it 👍 Let's go **step-by-step** to deploy your **Unsloth fine-tuned model** (PEFT/LoRA on Mistral, Llama, etc.) on **AWS SageMaker**. We'll use

the `inference.py` you already wrote.

1. Prepare Your Model Artifacts

You need to bundle the LoRA weights (from `unsloth`) into a tarball (`model.tar.gz`).

```
bash

# Assuming your LoRA adapter is in ./lora_model/
tar -czvf model.tar.gz lora_model/
```

Upload it to an **S3 bucket**:

```
bash

aws s3 cp model.tar.gz s3://<your-bucket-name>/unsloth-model/
```

2. inference.py (Already Done)

You already have:

```
python

# inference.py
from transformers import AutoTokenizer, AutoModelForCausalLM,
pipeline
from peft import PeftModel

def model_fn(model_dir):
    base_model = "unsloth/mistral-7b-bnb-4bit"
    model = AutoModelForCausalLM.from_pretrained(base_model,
load_in_4bit=True, device_map="auto")
    model = PeftModel.from_pretrained(model, model_dir)
    tokenizer = AutoTokenizer.from_pretrained(base_model)
    return pipeline("text-generation", model=model,
tokenizer=tokenizer)

def predict_fn(input_data, model):
    prompt = input_data["prompt"]
    return model(prompt, max_new_tokens=200)
```


3. requirements.txt

Create a `requirements.txt` for SageMaker inference container:

```
shell

transformers>=4.40.0
peft>=0.10.0
unsloth
accelerate
torch
```

4. Dockerfile (Custom SageMaker Image)

Because `unsloth` and 4-bit quantization require specific libs, you'll need a custom container.

```
dockerfile

FROM pytorch/pytorch:2.2.0-cuda12.1-cudnn8-runtime

# Install system deps
RUN apt-get update && apt-get install -y git && rm -rf /var/lib/apt/lists/*

# Upgrade pip
RUN pip install --upgrade pip

# Install python deps
COPY requirements.txt .
RUN pip install -r requirements.txt

# SageMaker Inference Toolkit
RUN pip install sagemaker-inference

# Copy inference script
COPY inference.py /opt/ml/model/code/inference.py

ENV SAGEMAKER_PROGRAM inference.py
```

5. Build & Push Docker Image

Replace `<account_id>`, `<region>`, `<repo_name>`:

```
bash

# Authenticate ECR
aws ecr create-repository --repository-name unsloth-inference
aws ecr get-login-password --region <region> | docker login --
username AWS --password-stdin <account_id>.dkr.ecr.
<region>.amazonaws.com

# Build and push
docker build -t unsloth-inference .
docker tag unsloth-inference:latest <account_id>.dkr.ecr.
<region>.amazonaws.com/unsloth-inference:latest
docker push <account_id>.dkr.ecr.
<region>.amazonaws.com/unsloth-inference:latest
```

6. Create SageMaker Model

Now tell SageMaker to use your custom image + model tarball:

```
python

import sagemaker
from sagemaker.model import Model
from sagemaker import get_execution_role

role = get_execution_role()
sess = sagemaker.Session()

image_uri = "<account_id>.dkr.ecr.
<region>.amazonaws.com/unsloth-inference:latest"
model_data = "s3://<your-bucket-name>/unsloth-
model/model.tar.gz"

unsloth_model = Model(
    image_uri=image_uri,
    model_data=model_data,
    role=role,
    name="unsloth-mistral"
)
```

7. Deploy Endpoint

```
python

predictor = unsloth_model.deploy(
    initial_instance_count=1,
    instance_type="ml.g5.2xlarge", # GPU instance for 4-bit
    endpoint_name="unsloth-mistral-endpoint"
)
```

8. Invoke Endpoint

```
python

response = predictor.predict({"prompt": "Explain reinforcement
learning in 2 lines."})
print(response)
```

✅ That's the **end-to-end deployment** flow:

- Package LoRA weights → S3
- Write `inference.py` (done)
- Create Docker image with deps
- Push to ECR
- Register model in SageMaker
- Deploy endpoint

Do you want me to also give you a **serverless (real-time inference) option without Docker** using SageMaker HuggingFace DLC + LoRA merge? That would be simpler but needs merging LoRA into base.

why docke ragain came

Good question 👍 — the **Docker step** comes back into the picture because of how **Amazon SageMaker custom models** work.

Let me clarify:

🚦 Two Paths to Deploy Models in SageMaker

1. Pre-built SageMaker containers (No Docker)

- Use Hugging Face DLCs (`huggingface-pytorch-inference` , `huggingface-tensorflow-inference` , etc.).
- You don't write Dockerfiles.
- You just upload `inference.py` + your model artifacts to S3 and point SageMaker to them.
- **Limitation:** these DLCs don't (yet) include Unsloth models directly.

2. Custom Containers (Docker Required)

- If your model (like **Unsloth** fine-tuned ones) needs extra libs (`unsloth` , `peft` , `bitsandbytes` , etc.), the pre-built Hugging Face container won't have them.
- That's why you need a **Docker image**:
 - You start from a base SageMaker PyTorch container.
 - Add your `requirements.txt` (`unsloth` , `peft` , etc.).
 - Copy your `inference.py` .
 - Push the image to **Amazon ECR**.
 - Deploy via SageMaker.

Why your `inference.py` looks fine

Your code:

```
python

from transformers import AutoTokenizer, AutoModelForCausalLM,
pipeline
from peft import PeftModel

def model_fn(model_dir):
    base_model = "unsloth/mistral-7b-bnb-4bit"
    model = AutoModelForCausalLM.from_pretrained(base_model,
load_in_4bit=True, device_map="auto")
    model = PeftModel.from_pretrained(model, model_dir)
    tokenizer = AutoTokenizer.from_pretrained(base_model)
    return pipeline("text-generation", model=model,
tokenizer=tokenizer)

def predict_fn(input_data, model):
```

```
prompt = input_data["prompt"]
return model(prompt, max_new_tokens=200)
```

✅ This is **perfect for SageMaker** inference entry point.

But SageMaker will run it **inside a container**, so we need to ensure the container has `unsloth`, `peft`, `bitsandbytes`, etc. installed.

That's why Docker shows up again.

👉 If you're okay with **building a custom Docker image once**, then you'll never touch Docker again — you'll just call SageMaker APIs for deployment.

Do you want me to give you a **step-by-step for option 2 (custom container)**, or should I show you a **hacky way to force option 1 (no Docker, use Hugging Face DLC + pip install at runtime)**?

```
import torch
from transformers import AutoTokenizer,
AutoModelForCausalLM
from peft import PeftModel

# 1. Load model + adapter
def model_fn(model_dir):
    base_model = "unsloth/mistral-7b-bnb-4bit"

    # Load quantized base model
    model =
AutoModelForCausalLM.from_pretrained(
    base_model,
    load_in_4bit=True,
    device_map="auto",
    torch_dtype=torch.float16
)

    # Load PEFT adapter from model_dir (your
    fine-tuned weights)
    model = PeftModel.from_pretrained(model,
    model_dir)
```

```
# Load tokenizer
tokenizer =
AutoTokenizer.from_pretrained(base_model)

return {"model": model, "tokenizer":
tokenizer}

# 2. Handle prediction
def predict_fn(input_data, model_dict):
    model = model_dict["model"]
    tokenizer = model_dict["tokenizer"]

    prompt = input_data.get("prompt", "")

    inputs = tokenizer(prompt,
return_tensors="pt").to(model.device)
    outputs = model.generate(
        **inputs,
        max_new_tokens=200,
        temperature=0.7,
        top_p=0.9,
        do_sample=True,
    )

    generated_text =
tokenizer.decode(outputs[0],
skip_special_tokens=True)
    return {"generated_text": generated_text}

# 3. Input function (for SageMaker JSON
input)
def input_fn(request_body,
request_content_type):
    if request_content_type ==
"application/json":
        import json
        return json.loads(request_body)
    else:
        raise ValueError(f"Unsupported content
type: {request_content_type}")

# 4. Output function (for SageMaker JSON
```

```

output)
def output_fn(prediction,
response_content_type):
    if response_content_type ==
"application/json":
        import json
        return json.dumps(prediction)
    else:
        return str(prediction)

```

use above inference.py , dont use docker just
deploy unsloth models in sagemaker

Absolutely—here's how to deploy your **Unsloth LoRA adapter** with your exact `inference.py` , **without writing any Dockerfile**, using SageMaker's **Hugging Face Inference DLC**. We'll rely on `entry_point + source_dir + requirements.txt` , which the HF DLC supports for real-time endpoints.

0) What we're doing

- Use your `inference.py` as the serving entrypoint.
- Ship only your **LoRA adapter weights** as `model.tar.gz` to S3 (SageMaker will unpack them into `/opt/ml/model` and pass that path to `model_fn(model_dir)`).
- Ask the HF DLC to **pip install** extra libs (`unsloth` , `peft` , etc.) from a `requirements.txt` in `source_dir` .
(HF docs + AWS blog confirm this flow for custom inference + requirements. [Hugging Face](#) [Amazon Web Services, Inc.](#) [Amazon SageMaker Documentation](#))

1) Make a small project folder

```

bash

deploy_unsloth/
├─ code/
│   └─ inference.py      # ← your file (paste it here
verbatim)
│   └─ requirements.txt

```

```
└─ model.tar.gz # ← LoRA adapter files packed (see
step 2)
```

code/requirements.txt (example)

```
graphql

transformers>=4.40.0
accelerate>=0.29.0
peft>=0.11.0
unsloth
bitsandbytes
torch # optional; DLC already includes torch, but harmless to
pin if needed
```

The HF DLC installs `requirements.txt` found in your `source_dir` at container startup for inference jobs when you provide `entry_point`.

[Hugging Face](#)
[Amazon Web Services, Inc.](#)
[Amazon SageMaker Documentation](#)

2) Package your LoRA adapter weights

If your adapter files are in `./lora_model/` (e.g., `adapter_config.json`, `adapter_model.bin`, etc.):

```
bash

cd lora_model
tar -czvf ../model.tar.gz ./
aws s3 cp ../model.tar.gz s3://<your-
bucket>/unsloth/model.tar.gz
```

SageMaker will untar this to `/opt/ml/model`, and that directory path is what your `model_fn(model_dir)` receives.

3) Paste your exact `inference.py` into `code/`

Use the file you provided (no changes needed):

```
python

# code/inference.py
import torch
```



```

from transformers import AutoTokenizer, AutoModelForCausalLM
from peft import PeftModel

def model_fn(model_dir):
    base_model = "unsloth/mistral-7b-bnb-4bit"
    model = AutoModelForCausalLM.from_pretrained(
        base_model,
        load_in_4bit=True,
        device_map="auto",
        torch_dtype=torch.float16
    )
    model = PeftModel.from_pretrained(model, model_dir)
    tokenizer = AutoTokenizer.from_pretrained(base_model)
    return {"model": model, "tokenizer": tokenizer}

def predict_fn(input_data, model_dict):
    model = model_dict["model"]
    tokenizer = model_dict["tokenizer"]
    prompt = input_data.get("prompt", "")
    inputs = tokenizer(prompt,
return_tensors="pt").to(model.device)
    outputs = model.generate(
        **inputs,
        max_new_tokens=200,
        temperature=0.7,
        top_p=0.9,
        do_sample=True,
    )
    generated_text = tokenizer.decode(outputs[0],
skip_special_tokens=True)
    return {"generated_text": generated_text}

def input_fn(request_body, request_content_type):
    if request_content_type == "application/json":
        import json
        return json.loads(request_body)
    else:
        raise ValueError(f"Unsupported content type:
{request_content_type}")

def output_fn(prediction, response_content_type):
    if response_content_type == "application/json":
        import json
        return json.dumps(prediction)

```

```

else:
    return str(prediction)

```

4) Deploy with the Hugging Face DLC (no Dockerfile needed)

```

python

import sagemaker
from sagemaker.huggingface import HuggingFaceModel

role = sagemaker.get_execution_role()
sess = sagemaker.Session()

# Your S3 path from step 2:
model_data = "s3://<your-bucket>/unsloth/model.tar.gz"

# Point to your code/ folder (contains inference.py + requirements.txt)
source_dir = "code"

# Pick DLC versions that support your stack
huggingface_model = HuggingFaceModel(
    role=role,
    model_data=model_data,          # the LoRA tarball
    entry_point="inference.py",     # must be at root of
    source_dir
    source_dir=source_dir,          # contains
    requirements.txt
    transformers_version="4.38",    # or newer that your
    region supports
    pytorch_version="2.1",         # choose a GPU-capable
    DLC
    py_version="py310",
    # Optional environment tips
    env={
        "HF_HUB_DISABLE_TELEMETRY": "1",
        # (Optional) cache to disk so repeated cold starts are
        faster:
        "TRANSFORMERS_CACHE": "/opt/ml/model/hf_cache",
    },
    sagemaker_session=sess,
)

```

```

predictor = huggingface_model.deploy(
    initial_instance_count=1,
    instance_type="ml.g5.2xlarge",    # GPU is required for
4-bit
    endpoint_name="unsloth-mistral"
)

```

Using `entry_point` + `source_dir` on a HF DLC is the documented way to add your custom inference script and auto-install dependencies for hosting.

Hugging Face

Amazon Web Services, Inc.

Amazon SageMaker Documentation

5) Invoke the endpoint

python

```

resp = predictor.predict({"prompt": "Explain reinforcement
learning in two lines."})
print(resp)

```

6) Clean up when done

python

```

predictor.delete_endpoint()

```

Important notes & gotchas

- **Internet access:** Your `model_fn` downloads `unsloth/mistral-7b-bnb-4bit` from the Hub at startup.

Ensure the endpoint **has egress** (don't deploy it in a private VPC with no internet).

If you can't allow egress, pre-bundle the base model into

`model.tar.gz` (large) and load it from `model_dir` instead of the hub.

- **Instance sizing:** 4-bit models are lighter, but still need a GPU (e.g., `ml.g5.2xlarge`). Scale up if OOM.

- **Cold start:** First container start will download the base model; subsequent starts are faster if you keep the endpoint warm and use `TRANSFORMERS_CACHE`.
- **Why this works:** HF DLC honors `inference.py` hooks and installs `requirements.txt` from `source_dir` for inference. No custom Dockerfile needed. Hugging Face Amazon Web Services, Inc.

If you want, I can also show the variant that **merges the LoRA** into the base weights offline (to avoid downloading the base at startup) and serves a single packaged model.

 Sources

don't use `HuggingFaceModel` use different

Got it 👍 — you don't want to use `HuggingFaceModel` class in SageMaker (which is Hugging Face-specific).

Instead, you can deploy your **Unsloth fine-tuned model** using the generic SageMaker `Model` class.

Here's a complete step-by-step workflow (no `HuggingFaceModel` involved):

1. Prepare `inference.py`

You already have this:

```
python

# inference.py
from transformers import AutoTokenizer, AutoModelForCausalLM,
pipeline
from peft import PeftModel

def model_fn(model_dir):
    base_model = "unsloth/mistral-7b-bnb-4bit"
    model = AutoModelForCausalLM.from_pretrained(base_model,
load_in_4bit=True, device_map="auto")
    model = PeftModel.from_pretrained(model, model_dir)
    tokenizer = AutoTokenizer.from_pretrained(base_model)
    return pipeline("text-generation", model=model,
tokenizer=tokenizer)
```

```
def predict_fn(input_data, model):  
    prompt = input_data["prompt"]  
    return model(prompt, max_new_tokens=200)
```

2. Create requirements.txt

So SageMaker installs dependencies inside the container:

```
shell  
  
transformers>=4.40.0  
accelerate>=0.29.0  
bitsandbytes  
peft  
torch>=2.1.0
```

3. Create a Dockerfile (only if you need custom container)

If you want to use SageMaker's PyTorch DLC, skip Docker.

If you want full control (e.g., Unsloth + 4bit), use Docker like this:

```
dockerfile  
  
FROM pytorch/pytorch:2.1.0-cuda11.8-cudnn8-runtime  
  
RUN pip install --upgrade pip  
COPY requirements.txt .  
RUN pip install -r requirements.txt  
  
# SageMaker requires this path  
COPY inference.py /opt/ml/model/code/inference.py  
  
ENV SAGEMAKER_PROGRAM inference.py
```

Build and push to ECR if using Docker.

4. Deploy with Generic Model class (no HuggingFaceModel)

```

python

import sagemaker
from sagemaker.model import Model
from sagemaker.predictor import Predictor
from sagemaker.serializers import JSONSerializer
from sagemaker.deserializers import JSONDeserializer

role = "<your-sagemaker-execution-role-arn>"
sess = sagemaker.Session()

# Option A: Use SageMaker PyTorch DLC
image_uri = sagemaker.image_uris.retrieve(
    framework="pytorch",
    region=sess.boto_region_name,
    version="2.1",
    py_version="py310",
    image_scope="inference"
)

# Option B: If you built custom Docker -> image_uri = "
<your_ecr_image_uri>"

model = Model(
    image_uri=image_uri,
    model_data="s3://<your-bucket>/<your-model-artifacts>.tar.gz", # uploaded model.tar.gz
    role=role,
    entry_point="inference.py", # only needed if using
    PyTorch DLC
    predictor_cls=Predictor,
)

predictor = model.deploy(
    initial_instance_count=1,
    instance_type="ml.g5.2xlarge", # or bigger depending on
    model
    serializer=JSONSerializer(),
    deserializer=JSONDeserializer(),
)

# Call inference
response = predictor.predict({"prompt": "Explain transformers
in NLP"})
print(response)

```

✅ Key difference: here we use `sagemaker.Model` instead of `HuggingFaceModel`.

That way, it works for **Unsloth**, **PEFT**, **LoRA**, or any custom model.

Do you want me to write the **end-to-end flow using only PyTorch DLC** (no Docker at all, just `model.tar.gz` upload)? That'll be simpler than managing a Dockerfile.