Comprehensive Guide: Deploying SDXL on Amazon SageMaker (Detailed)

1. Prepare Your Environment

1.1 AWS Account Setup

1. Ensure you have an AWS account with the necessary permissions.
    - You'll need permissions for S3, ECR, and SageMaker.
    - If you don't have an account, create one at https://aws.amazon.com/
2. Install and configure the AWS CLI on your local machine:

    ```
    pip install awscli
    aws configure
    ```

    - When configuring, you'll be prompted to enter your AWS Access Key ID, Secret Access Key, default region, and output format.

1.2 Local Environment Setup

1. Install required Python packages:

    ```
    pip install boto3 sagemaker
    ```

    - boto3 is the AWS SDK for Python, which we'll use to interact with various AWS services.
    - sagemaker is the SageMaker Python SDK, which provides convenient abstractions for working with SageMaker.

2. Prepare Your Models

1. Ensure your base model, refiner model, and LoRA weights are ready.
    - These should be in a format compatible with the Hugging Face Diffusers library.
2. Create an S3 bucket (if you don't have one already):

    ```
    aws s3 mb s3://your-sdxl-bucket
    ```

    - Replace `your-sdxl-bucket` with a globally unique bucket name.
    - This bucket will store your models and training data.

3. Upload your models to S3:

   ```
   aws s3 cp --recursive ./base s3://your-sdxl-bucket/sdxl-models/base/
   aws s3 cp --recursive ./refiner s3://your-sdxl-bucket/sdxl-models/refi
   aws s3 cp --recursive ./Trained_lora s3://your-sdxl-bucket/sdxl-models
   ```

   - These commands assume your models are in local directories named `base`, `refiner`, and `Trained_lora`.
   - Adjust the paths as necessary based on your local file structure.

3. Set Up Your SageMaker Project

1. Create a new directory for your SageMaker project:

   ```
   mkdir sdxl-sagemaker && cd sdxl-sagemaker
   ```

2. Create the following files in your project directory:

3.1 requirements.txt

This file lists all the Python packages required for your project.

```
diffusers==0.19.3
transformers==4.31.0
torch==2.0.1
accelerate==0.21.0
compel==2.0.2
Pillow==10.0.0
fastapi==0.100.0
uvicorn==0.23.1
boto3==1.28.15
```

3.2 Dockerfile

This Dockerfile sets up the environment for your SageMaker endpoint.

```
FROM pytorch/pytorch:2.0.0-cuda11.7-cudnn8-runtime

RUN pip install -U pip

COPY requirements.txt .
RUN pip install -r requirements.txt

COPY inference.py /opt/ml/code/inference.py
COPY train_dreambooth_lora_sdxl.py /opt/ml/code/train_dreambooth_lora_sdxl

ENV PYTHONUNBUFFERED=TRUE
ENV PYTHONDONTWRITEBYTECODE=TRUE
ENV PATH="/opt/ml/code:${PATH}"

WORKDIR /opt/ml/code
```

```
CMD ["python", "inference.py"]
```

Explanation: - We start from a PyTorch base image that includes CUDA support. - We install the required Python packages. - We copy our inference and training scripts into the container. - We set some environment variables and the working directory. - The CMD instruction specifies that the `inference.py` script should be run when the container starts.

3.3 inference.py

This script handles both inference and training requests. Here's a detailed breakdown:

```
import os
import torch
from diffusers import DiffusionPipeline, DPMSolverMultistepScheduler
from compel import Compel, ReturnedEmbeddingsType
import json
import base64
from io import BytesIO
from fastapi import FastAPI, HTTPException
import asyncio
import subprocess
import boto3

app = FastAPI()

def model_fn(model_dir):
    base_path = os.path.join(model_dir, 'base')
    refiner_path = os.path.join(model_dir, 'refiner')
    lora_path = os.path.join(model_dir, 'Trained_lora')

    # Load the base model
    base = DiffusionPipeline.from_pretrained(
        base_path,
        torch_dtype=torch.float16,
        variant="fp16",
        use_safetensors=True,
    ).to("cuda")

    # Load LoRA weights
    base.load_lora_weights(
        lora_path,
        weight_name="pytorch_lora_weights.safetensors"
    )

    # Load the refiner model
    refiner = DiffusionPipeline.from_pretrained(
        refiner_path,
        text_encoder_2=base.text_encoder_2,
        vae=base.vae,
```

```
        torch_dtype=torch.float16,
        use_safetensors=True,
        variant="fp16",
    ).to("cuda")

    # Set up Compel for text embedding
    compel = Compel(
        tokenizer=[base.tokenizer, base.tokenizer_2],
        text_encoder=[base.text_encoder, base.text_encoder_2],
        returned_embeddings_type=ReturnedEmbeddingsType.PENULTIMATE_HIDDEN
        requires_pooled=[False, True])

    compel_refiner = Compel(
        tokenizer=[refiner.tokenizer_2],
        text_encoder=[refiner.text_encoder_2],
        returned_embeddings_type=ReturnedEmbeddingsType.PENULTIMATE_HIDDEN
        requires_pooled=[True],
    )

    return base, refiner, compel, compel_refiner

model = None
subprocess_manager = None

def initialize():
    global model, subprocess_manager
    model = model_fn("/opt/ml/model")
    subprocess_manager = SubprocessManager()

@app.post("/invoke")
async def invoke(input_data: dict):
    global model
    base, refiner, compel, compel_refiner = model
    prompt = input_data['prompt']
    negative_prompt = input_data.get('negative_prompt', '')

    # Generate text embeddings
    conditioning, pooled = compel(prompt)
    negative_conditioning, negative_pooled = compel(negative_prompt)
    conditioning_refiner, pooled_refiner = compel_refiner(prompt)
    negative_conditioning_refiner, negative_pooled_refiner = compel_refine

    # Generate image with base model
    image = base(
        prompt_embeds=conditioning,
        pooled_prompt_embeds=pooled,
        negative_prompt_embeds=negative_conditioning,
        negative_pooled_prompt_embeds=negative_pooled,
        num_inference_steps=40,
        denoising_end=0.8,
        output_type="latent",
    ).images[0]
```

```
    # Refine the image
    refiner_result = refiner(
        prompt_embeds=conditioning_refiner,
        pooled_prompt_embeds=pooled_refiner,
        negative_prompt_embeds=negative_conditioning_refiner,
        negative_pooled_prompt_embeds=negative_pooled_refiner,
        num_inference_steps=40,
        denoising_start=0.8,
        image=image,
    ).images[0]

    # Convert image to base64 string
    buffered = BytesIO()
    refiner_result.save(buffered, format="PNG")
    img_str = base64.b64encode(buffered.getvalue()).decode()

    return {'image': img_str}

class SubprocessManager:
    def __init__(self):
        self.process = None
        self.status = "Not Started"

    async def run_script(self, command):
        self.status = "Running"
        try:
            self.process = await asyncio.create_subprocess_shell(
                command,
                stdout=asyncio.subprocess.PIPE,
                stderr=asyncio.subprocess.PIPE
            )
            stdout, stderr = await self.process.communicate()
            self.status = "Completed" if self.process.returncode == 0 else
            return stdout.decode(), stderr.decode()
        except Exception as e:
            self.status = "Failed"
            raise e
        finally:
            self.process = None

    def get_status(self):
        return self.status

@app.post("/train")
async def train(collection_s3_path: str, prompt: str, output_dir_name: str
    global subprocess_manager
    if subprocess_manager.get_status() == "Running":
        raise HTTPException(status_code=400, detail="A training process is

    s3 = boto3.resource('s3')
    collection_bucket, collection_key = parse_s3_uri(collection_s3_path)
```

```
    local_collection_path = '/tmp/training_data'
    download_from_s3(s3, collection_bucket, collection_key, local_collecti

    output_s3_path = f"s3://{collection_bucket}/model_outputs/{output_dir_

    command = f"""
    accelerate launch train_dreambooth_lora_sdxl.py \
      --pretrained_model_name_or_path='/opt/ml/model/base'  \
      --instance_data_dir='{local_collection_path}' \
      --pretrained_vae_model_name_or_path="madebyollin/sdxl-vae-fp16-fix"
      --output_dir="/tmp/trained_model" \
      --mixed_precision="fp16" \
      --instance_prompt="{prompt}" \
      --resolution=1024 \
      --train_batch_size=2 \
      --gradient_accumulation_steps=2 \
      --gradient_checkpointing \
      --learning_rate=1e-4 \
      --lr_scheduler="constant" \
      --lr_warmup_steps=0 \
      --max_train_steps=500 \
      --seed="0"
    """

    training_task = asyncio.create_task(subprocess_manager.run_script(comm

    async def upload_results():
        await training_task
        upload_to_s3(s3, '/tmp/trained_model', collection_bucket, f"model_

    asyncio.create_task(upload_results())

    return {"detail": "Training process started", "output_s3_path": output

@app.get("/train-status")
def train_status():
    global subprocess_manager
    return {"status": subprocess_manager.get_status()}

def parse_s3_uri(uri):
    parts = uri.replace("s3://", "").split("/")
    bucket = parts.pop(0)
    key = "/".join(parts)
    return bucket, key

def download_from_s3(s3, bucket, key, local_path):
    os.makedirs(local_path, exist_ok=True)
    for obj in s3.Bucket(bucket).objects.filter(Prefix=key):
        if not obj.key.endswith('/'):
            target = os.path.join(local_path, os.path.relpath(obj.key, key
            if not os.path.exists(os.path.dirname(target)):
```

```
                os.makedirs(os.path.dirname(target))
            s3.Bucket(bucket).download_file(obj.key, target)

def upload_to_s3(s3, local_dir, bucket, s3_path):
    for root, _, files in os.walk(local_dir):
        for file in files:
            local_file = os.path.join(root, file)
            relative_path = os.path.relpath(local_file, local_dir)
            s3_file = os.path.join(s3_path, relative_path)
            s3.Bucket(bucket).upload_file(local_file, s3_file)

@app.on_event("startup")
async def startup_event():
    initialize()

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

Explanation: - We use FastAPI to create a web server that can handle both inference and training requests. - The `model_fn` function loads the base model, refiner model, and LoRA weights, and sets up the Compel embedder. - The `/invoke` endpoint handles inference requests. It generates text embeddings, creates an image with the base model, and then refines it. - The `/train` endpoint handles training requests. It downloads training data from S3, runs the training script, and uploads the results back to S3. - We use asyncio to handle concurrent requests and manage the training subprocess.

3.4 traindreamboothlora_sdxl.py

This file should contain your SDXL training script. Ensure it's compatible with the command in the train function of inference.py. The exact contents will depend on your specific training requirements.

3.5 buildandpush.sh

This script builds your Docker image and pushes it to Amazon ECR.

```
#!/bin/bash

# The name of our algorithm
algorithm_name=sdxl-sagemaker

account=$(aws sts get-caller-identity --query Account --output text)

# Get the region defined in the current configuration
region=$(aws configure get region)

fullname="${account}.dkr.ecr.${region}.amazonaws.com/${algorithm_name}:lat

# If the repository doesn't exist in ECR, create it.
```

```
aws ecr describe-repositories --repository-names "${algorithm_name}" > /de

if [ $? -ne 0 ]
then
    aws ecr create-repository --repository-name "${algorithm_name}" > /dev
fi

# Get the login command from ECR and execute it directly
aws ecr get-login-password --region ${region} | docker login --username AW

# Build the docker image locally with the image name and then push it to E
# with the full name.
docker build -t ${algorithm_name} .
docker tag ${algorithm_name} ${fullname}

docker push ${fullname}
```

Explanation: - This script automates the process of building your Docker image and pushing it to Amazon ECR. - It first checks if the ECR repository exists, creating it if necessary. - It then logs in to ECR, builds the Docker image, tags it, and pushes it to ECR.

4. Build and Push the Docker Image

1. Make the build script executable:

   ```
   chmod +x build_and_push.sh
   ```

2. Run the build script:

   ```
   ./build_and_push.sh
   ```

This will build your Docker image and push it to Amazon ECR, making it available for use with SageMaker.

5. Create a SageMaker Model

Use the following Python script to create a SageMaker model:

```
import boto3
import sagemaker

# Initialize SageMaker session and get the execution role
sagemaker_session = sagemaker.Session()
role = sagemaker.get_execution_role()

# Get the account ID and region
account = boto3.client('sts').get_caller_identity().get('Account')
region = boto3.session.Session().region_name

# Construct the ECR image URI
```

```
image = f'{account}.dkr.ecr.{region}.amazonaws.com/sdxl-sagemaker:latest'

# Set the model name
model_name = 'sdxl-model'

# Create the SageMaker model
model = sagemaker.model.Model(
    image_uri=image,
    model_data='s3://your-sdxl-bucket/sdxl-models/',
    role=role,
    name=model_name,
    sagemaker_session=sagemaker_session
)

# Create the model in SageMaker
model.create(instance_type='ml.g4dn.xlarge')
```

Explanation: - We initialize a SageMaker session and get the execution role. The role should have permissions to access the necessary AWS resources. - We get the AWS account ID and region, which we use to construct the ECR image URI. - We create a SageMaker Model object, specifying: - The image URI of our Docker container in ECR - The S3 path where our model artifacts are stored - The IAM role for SageMaker to assume - A name for our model - Finally, we call `create()` to create the model in SageMaker. We specify an instance type that will be used for deployment.

6. Create a SageMaker Endpoint

Use the following Python script to create and deploy a SageMaker endpoint:

```
# Set the endpoint name
endpoint_name = 'sdxl-endpoint'

# Deploy the model to create an endpoint
model.deploy(
    initial_instance_count=1,
    instance_type='ml.g4dn.xlarge',
    endpoint_name=endpoint_name
)
```

Explanation: - We set a name for our endpoint. - We call the `deploy()` method on our model to create an endpoint. - We specify: - The initial number of instances (1 in this case) - The instance type to use (ml.g4dn.xlarge, which is suitable for GPU workloads) - The name of the endpoint

This process may take several minutes as SageMaker provisions the necessary resources and deploys your model.

7. Use the Endpoint

Once your endpoint is deployed, you can use it for both inference and training.

7.1 For Inference

Here's how to use the endpoint for inference:

```
import boto3
import json
import base64
from PIL import Image
import io

# Create a SageMaker runtime client
runtime = boto3.client('sagemaker-runtime')

endpoint_name = 'sdxl-endpoint'
content_type = "application/json"

# Prepare the payload
payload = {
    "prompt": "A sleek, aerodynamic electric crossover concept car in a fu
    "negative_prompt": "malformed, extra wheels, poorly drawn, blurry, low
}

# Invoke the endpoint
response = runtime.invoke_endpoint(
    EndpointName=endpoint_name,
    ContentType=content_type,
    Body=json.dumps(payload)
)

# Parse the response
result = json.loads(response['Body'].read().decode())

# Decode the base64 image
image_data = base64.b64decode(result['image'])
image = Image.open(io.BytesIO(image_data))

# Save or display the image
image.save("generated_image.png")
image.show()
```

Explanation: - We create a SageMaker runtime client to interact with our endpoint. - We prepare a payload with our prompt and negative prompt. - We invoke the endpoint, sending our payload as a JSON string. - We parse the response, which contains a base64-encoded image. - We decode the image and can then save or display it.

7.2 For Training

Here's how to use the endpoint to start a training job:

```
import boto3
import json

# Create a SageMaker runtime client
runtime = boto3.client('sagemaker-runtime')

endpoint_name = 'sdxl-endpoint'
content_type = "application/json"

# Prepare the payload
payload = {
    "action": "train",
    "data": {
        "collection_s3_path": "s3://your-bucket/path/to/training/images",
        "prompt": "a photo of sks dog",
        "output_dir_name": "sks_dog_model"
    }
}

# Invoke the endpoint
response = runtime.invoke_endpoint(
    EndpointName=endpoint_name,
    ContentType=content_type,
    Body=json.dumps(payload)
)

# Parse the response
result = json.loads(response['Body'].read().decode())
print(result)
```

Explanation: - We use the same SageMaker runtime client as for inference. - We prepare a payload that includes: - The S3 path to our training images - The training prompt - A name for the output directory - We invoke the endpoint with this payload. - The response will include details about the started training job, including where the output will be stored in S3.

7.3 Checking Training Status

You can check the status of a training job like this:

```
import boto3
import json

runtime = boto3.client('sagemaker-runtime')

endpoint_name = 'sdxl-endpoint'
content_type = "application/json"
```

```
payload = {
    "action": "train-status"
}

response = runtime.invoke_endpoint(
    EndpointName=endpoint_name,
    ContentType=content_type,
    Body=json.dumps(payload)
)

result = json.loads(response['Body'].read().decode())
print(result)
```

This will return the current status of the training job.

8. Clean Up

When you're done using your endpoint, remember to delete it to avoid incurring unnecessary charges:

```
import boto3

sagemaker = boto3.client('sagemaker')

# Delete the endpoint
sagemaker.delete_endpoint(EndpointName='sdxl-endpoint')

# Delete the endpoint configuration
sagemaker.delete_endpoint_config(EndpointConfigName='sdxl-endpoint')

# Delete the model
sagemaker.delete_model(ModelName='sdxl-model')
```

This will delete the endpoint, endpoint configuration, and model from SageMaker.

Remember to also delete any unnecessary S3 buckets and ECR repositories to fully clean up your resources.

Conclusion

This guide walked you through the process of deploying an SDXL model on Amazon SageMaker, from setting up your environment to using the deployed endpoint for both inference and training. Remember to monitor your usage and costs, and to clean up resources when they're no longer needed.