How to Deploy Diffusion Models

Aniket Maurya
Developer Advocate at Lightning AI

Lightning AI

We want to hear from you!

https://bit.ly/3TVmrWn

Lightning AI

Agenda

- Introduction to Diffusion Models & Text-to-image Generators
- Building a Prediction API
- Autoscaling the Model Server
- Dynamic Batching
- Deploying custom version of Stable Diffusion

Powered by: Muse Lightning App

Explore the code

Woman painting a red egg in a dali landscape

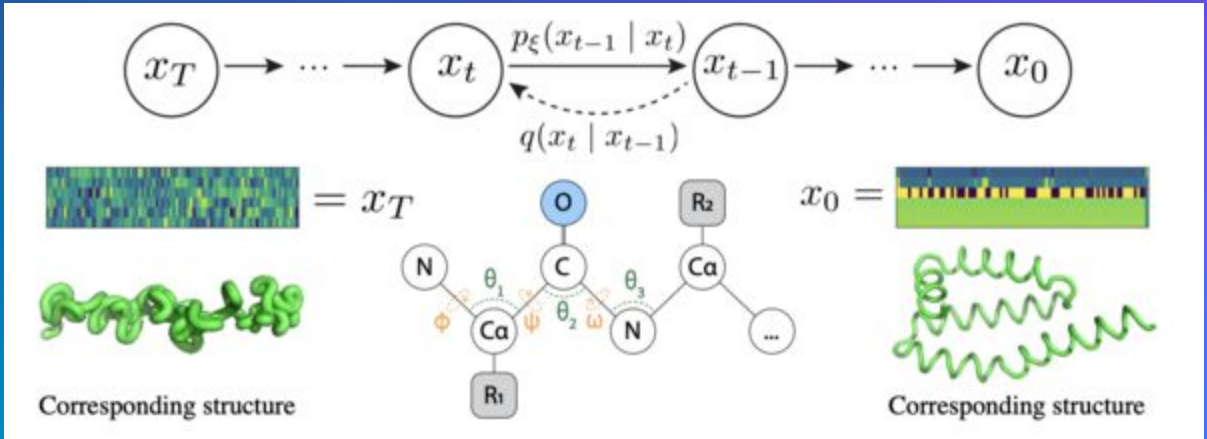
Muse

Inference latency (GPU auto-scaled): 2,000ms

Lightning AI

Source: Linkedin

Protein Backbone Generation



Source: Arxiv

Learn more:

https://youtu.be/AQrMWH8aC0Q

https://newsletter.sebastianraschka.com/issues/ahead-of-ai-1-a-diffusion-of-innovations-1400764

Deploying Diffusion Models

Stable Diffusion in Production

lightning.ai/pages/community/tutorial/deploy-diffusion-models

MUSE

Stable Diffusion in Production

lightning.ai/pages/community/tutorial/deploy-diffusion-models

Muse as a Blueprint

- A blueprint for any kind of model deployment
- Serves model on CPU or GPU
- Autoscale infrastructure based on traffic
- Dynamic GPU batching for requests
- React UI connected to the backend
- Rate Limiter and system monitoring

Model Server

- Loads the model
- Defines prediction functionality
- Exposes REST API

Building the Model Server

```
class StableDiffusionServe(L.LightningWork):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self._model = None


    def build_model(self):
        return load_model("model_path")
```

Building the Model Server

```
@torch.inference_mode()
def predict(self, prompt: str):
    preds = self._model(
        prompt,
        height=height,
        width=width,
        num_inference_steps=num_inference_steps,
    )
    return pil_to_base64(preds.images[0])
```

```python
def run(self):
    # 1. Define model
    # 2. Create FastAPI endpoint
    # 3. generate image from the prompt
    # 4. return base 64 image

    if self._model is None:
        self._model = self.build_model()

    app = FastAPI()

    @app.post("/api/predict")
    def predict_api(data):
        result = self.predict(data).result()
        return result

    uvicorn.run(app, host=self.host, port=self.port)
```

```python
def run(self):
    # 1. Define model
    # 2. Create FastAPI endpoint
    # 3. generate image from the prompt
    # 4. return base 64 image

    if self._model is None:
        self._model = self.build_model()

    app = FastAPI()

    @app.post("/api/predict")
    def predict_api(data):
        result = self.predict(data).result()
        return result

    uvicorn.run(app, host=self.host, port=self.port)
```

Initialize model

```
def run(self):
    # 1. Define model
    # 2. Create FastAPI endpoint
    # 3. generate image from the prompt
    # 4. return base 64 image

    if self._model is None:
        self._model = self.build_model()

    app = FastAPI()

    @app.post("/api/predict")
    def predict_api(data):
        result = self.predict(data).result()
        return result

    uvicorn.run(app, host=self.host, port=self.port)
```

Initialize FastAPI

```python
def run(self):
    # 1. Define model
    # 2. Create FastAPI endpoint
    # 3. generate image from the prompt
    # 4. return base 64 image

    if self._model is None:
        self._model = self.build_model()

    app = FastAPI()

    @app.post("/api/predict")
    def predict_api(data):
        result = self.predict(data).result()
        return result

    uvicorn.run(app, host=self.host, port=self.port)
```

Define the REST API

```python
def run(self):
    # 1. Define model
    # 2. Create FastAPI endpoint
    # 3. generate image from the prompt
    # 4. return base 64 image

    if self._model is None:
        self._model = self.build_model()

    app = FastAPI()

    @app.post("/api/predict")
    def predict_api(data):
        result = self.predict(data).result()
        return result

    uvicorn.run(app, host=self.host, port=self.port)
```

Launch the server

This wasn’t scalable

- Processing requests sequentially
- No batching on GPU
- No parallelization of servers

Scaling with Load Balancer

- Launch multiple parallel model servers
- Routes the client request to servers
- Batch the incoming requests

Scaling with Load Balancer

Batching	Performance
Without batching	less than 5 concurrent users
With batching	more than 10 concurrent users
With batching + more servers	100 concurrent users (sustained) 300 concurrent users (burst)

Generated images

Batched prompts

Aggregate Requests/
Timeout

Model Server

Model Server

Model Server

Scalable

* Dynamic Batching
* Auto Scaling

Load Balancer

/predict

B64 Image

client

Lightning AI

Generated images

Batched prompts

Aggregate Requests/
Timeout

Model Server

Model Server

Model Server

Scalable

* Dynamic Batching
* Auto Scaling

Load Balancer

/predict

B64 Image

client

Lightning AI

Generated images

Batched prompts

Aggregate Requests/
Timeout

Model Server

Model Server

Model Server

Scalable

* Dynamic Batching
* Auto Scaling

Load Balancer

/predict

B64 Image

client

Lightning AI

Load Balancer - Autoscaling

```
class MuseFlow(L.LightningFlow):
    def __init__(
        self,
        initial_num_workers: int = MUSE_MIN_WORKERS,
        autoscale_interval: int = 1 * 30,
        max_batch_size: int = 4,
        batch_timeout_secs: int = 10,
        gpu_type: str = MUSE_GPU_TYPE,
        max_workers: int = 20,
        autoscale_down_limit: Optional[int] = None,
        autoscale_up_limit: Optional[int] = None,
        load_testing: Optional[bool] = False
    )
```

Initial number of model servers

Load Balancer - Autoscaling

```
class MuseFlow(L.LightningFlow):
    def __init__(
        self,
        initial_num_workers: int = MUSE_MIN_WORKERS,
        autoscale_interval: int = 1 * 30,
        max_batch_size: int = 4,
        batch_timeout_secs: int = 10,
        gpu_type: str = MUSE_GPU_TYPE,
        max_workers: int = 20,
        autoscale_down_limit: Optional[int] = None,
        autoscale_up_limit: Optional[int] = None,
        load_testing: Optional[bool] = False
    )
```

Check autoscaling at interval of 30 seconds

Load Balancer - Dynamic Batching

```
class MuseFlow(L.LightningFlow):
    def __init__(
        self,
        initial_num_workers: int = MUSE_MIN_WORKERS,
        autoscale_interval: int = 1 * 30,
        max_batch_size: int = 4,
        batch_timeout_secs: int = 10,
        gpu_type: str = MUSE_GPU_TYPE,
        max_workers: int = 20,
        autoscale_down_limit: Optional[int] = None,
        autoscale_up_limit: Optional[int] = None,
        load_testing: Optional[bool] = False
    )
```

Maximum batch size

Load Balancer - Dynamic Batching

```
class MuseFlow(L.LightningFlow):
    def __init__(
        self,
        initial_num_workers: int = MUSE_MIN_WORKERS,
        autoscale_interval: int = 1 * 30,
        max_batch_size: int = 4,
        batch_timeout_secs: int = 10,
        gpu_type: str = MUSE_GPU_TYPE,
        max_workers: int = 20,
        autoscale_down_limit: Optional[int] = None,
        autoscale_up_limit: Optional[int] = None,
        load_testing: Optional[bool] = False
    )
```

Wait for 10 seconds to aggregate requests

Load Balancer - Autoscaling

```python
class MuseFlow(L.LightningFlow):
    def __init__(
        self,
        initial_num_workers: int = MUSE_MIN_WORKERS,
        autoscale_interval: int = 1 * 30,
        max_batch_size: int = 4,
        batch_timeout_secs: int = 10,
        gpu_type: str = MUSE_GPU_TYPE,
        max_workers: int = 20,
        autoscale_down_limit: Optional[int] = None,
        autoscale_up_limit: Optional[int] = None,
        load_testing: Optional[bool] = False
    )
```

Select the GPU type

Load Balancer - Autoscaling

```
class MuseFlow(L.LightningFlow):
    def __init__(
        self,
        initial_num_workers: int = MUSE_MIN_WORKERS,
        autoscale_interval: int = 1 * 30,
        max_batch_size: int = 4,
        batch_timeout_secs: int = 10,
        gpu_type: str = MUSE_GPU_TYPE,
        max_workers: int = 20,
        autoscale_down_limit: Optional[int] = None,
        autoscale_up_limit: Optional[int] = None,
        load_testing: Optional[bool] = False
    )
```

Maximum number of parallel model servers

Load Balancer - Autoscaling

```
class MuseFlow(L.LightningFlow):
    def __init__(
        self,
        initial_num_workers: int = MUSE_MIN_WORKERS,
        autoscale_interval: int = 1 * 30,
        max_batch_size: int = 4,
        batch_timeout_secs: int = 10,
        gpu_type: str = MUSE_GPU_TYPE,
        max_workers: int = 20,
        autoscale_down_limit: Optional[int] = None,
        autoscale_up_limit: Optional[int] = None,
        load_testing: Optional[bool] = False
    )
```

Downscale if traffic is below the limit

Load Balancer - Autoscaling

```
class MuseFlow(L.LightningFlow):
    def __init__(
        self,
        initial_num_workers: int = MUSE_MIN_WORKERS,
        autoscale_interval: int = 1 * 30,
        max_batch_size: int = 4,
        batch_timeout_secs: int = 10,
        gpu_type: str = MUSE_GPU_TYPE,
        max_workers: int = 20,
        autoscale_down_limit: Optional[int] = None,
        autoscale_up_limit: Optional[int] = None,
        load_testing: Optional[bool] = False
    )
```

Upscale if traffic is above the limit

Load Balancer - Autoscaling

```python
class MuseFlow(L.LightningFlow):
    def __init__(
        self,
        initial_num_workers: int = MUSE_MIN_WORKERS,
        autoscale_interval: int = 1 * 30,
        max_batch_size: int = 4,
        batch_timeout_secs: int = 10,
        gpu_type: str = MUSE_GPU_TYPE,
        max_workers: int = 20,
        autoscale_down_limit: Optional[int] = None,
        autoscale_up_limit: Optional[int] = None,
        load_testing: Optional[bool] = False
    )
```

Code walkthrough

Deploy a custom version

Deploy on Cloud

```
lightning run app app.py
```

```
lightning run app app.py --cloud
```

Key Takeaways

- Implement Dynamic batching and autoscaling
- Increase performance from 5 concurrent users to a burst of 300+ concurrent users.
- Pure Python
- Lightning App manages cloud provisioning, GPU instance selection, disk allocation and other infrastructure challenges.

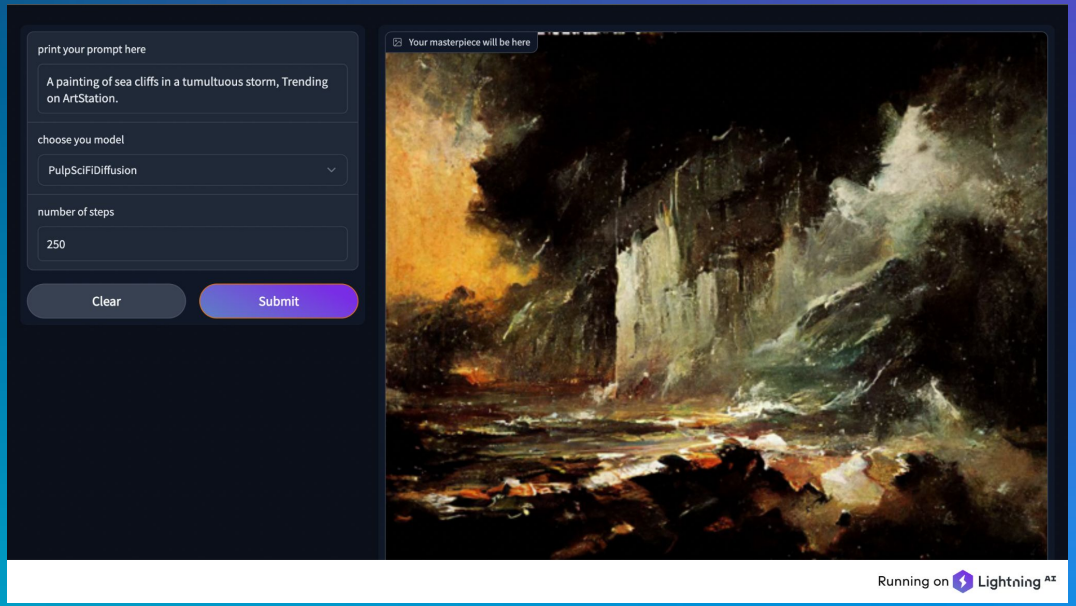
Key Takeaways

Support a burst of 300+ concurrent users

Only Python

Lightning manages cloud provisioning, GPU instance selection, disk allocation, etc.

Community Showcase



Irina
https://github.com/KogayIrina

Contribute to:
https://github.com/Lightning-AI-Dev/Awesome-Lightning

Use **#BuildWithLightning** &

Tag us to be featured!

Lightning AI

Resources

- http://lightning.ai/muse
- Discover What Lightning can do in 5 mins!
- meetup.com/pro/pytorch-lightning
- pytorch-lightning.slack.com

Thank You

Aniket Maurya

aniket@lightning.ai

twitter.com/aniketmaurya

linkedin.com/in/aniketmaurya

https://bit.ly/3TVmrWn