

Evaluating and Debugging Generative AI


[**Lesson 1: Introduction to Weights & Biases \(wandb\) for ML Monitoring and Debugging**](#)


[**Lesson 2: Training Diffusion Models with "wandb"**](#)

[**Lesson 3: Comparing Diffusion Model Outputs and Evaluating LLM Models**](#)

[**Lesson 4: Evaluating Large Language Models \(LLMs\)**](#)

[**Lesson 5: Fine-Tuning Large Language Models \(LLMs\)**](#)

 Weights & Biases



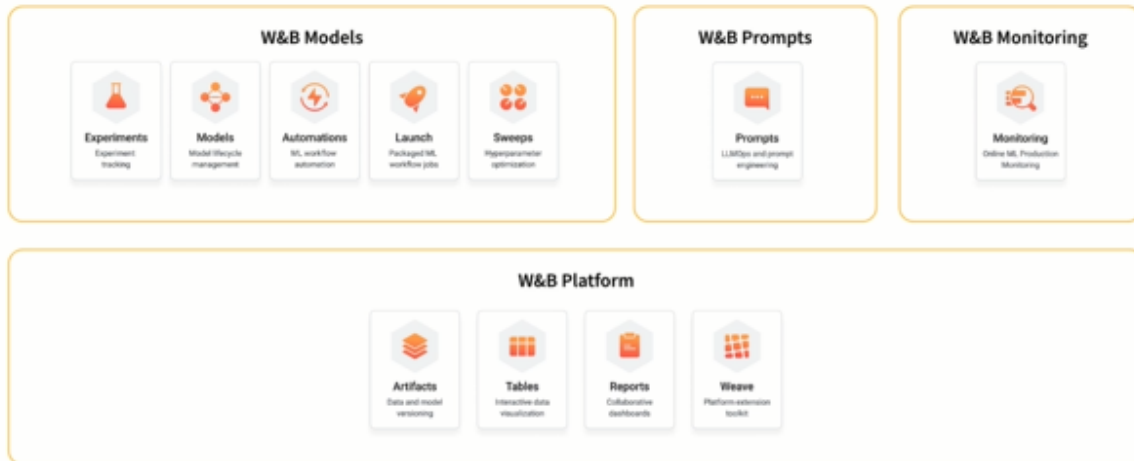
Evaluating and Debugging Generative AI

Using Weights & Biases Tools

- ❖ Instrument W&B in an ML training pipeline
- ❖ Training diffusion models
- ❖ Evaluating diffusion models
- ❖ Evaluating LLMs
- ❖ Fine-tuning LLMs

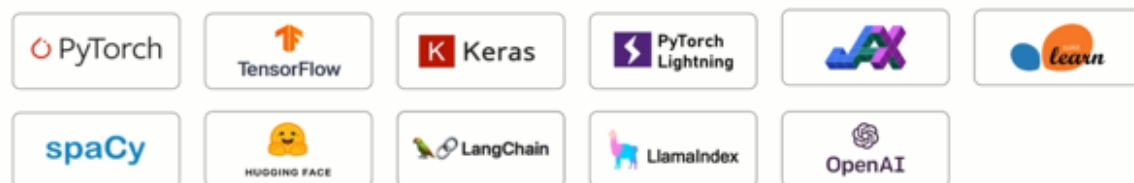
Weights & Biases MLOps Portfolio

Tools for Machine Learning Practitioners



W&B MLOps Platform

Integrated into every popular ML framework



Runs on every cloud or in your own infra



****Lesson 1: Introduction to Weights & Biases (wandb) for ML Monitoring and Debugging****

Why Use Weights & Biases

Debugging and evaluating Generative AI

- ❖ Integrate quickly, track & version automatically
- ❖ Visualize your data and uncover critical insights
- ❖ Improve performance so you can evaluate and deploy with confidence



****Overview****

- Introduction to instrumenting weights and biases in ML training code.
- "wandb" for monitoring, debugging, and evaluating ML pipelines.

****Benefits of "wandb"****

- Real-time monitoring of metrics, CPU, and GPU usage.
- Version control for code and model checkpoints.
- Centralized, interactive dashboard for visualization.
- Configurable reports for model evaluation and bug discussion.

Instrumenting W&B

Integrate with any Python script

```
pip install wandb
import wandb

# 1. Organize your hyperparameters
config = {'learning_rate': 0.001}

# 2. Start wandb run
wandb.init(project='gpt5', config=config)

# Model training here

# 3. Log metrics over time to visualize
performance
wandb.log({"loss": loss})

# 4. When working in a notebook, finish
wandb.finish()
```

****Incorporating "wandb"****

- Install "wandb" with `pip install wandb`.
- Import "wandb" library and prepare hyperparameters in a config object.

****Initiating a "wandb" Run****

- Call `wandb.init` with project name and config to start a run.
- A "run" corresponds to a machine learning experiment.

****Logging Metrics****

- Use `wandb.log` to track and visualize metrics during training.
- Log metrics when needed for analysis.

****Training Process Integration****

- Incorporate "wandb" into your training loop.
- Log metrics using `wandb.log` during training.
- Record validation metrics at end of each epoch.

****Finishing a Run****

- Optionally, call `wandb.finish` to end a "wandb" run.
- Especially recommended when using notebooks.

****Training Sprite Classification Model Example****

- Import necessary libraries, including "wandb".
- Define a simple classifier model with linear layers.

****Training Function Modification****

- Modify training function to include "wandb" logging.
- Call `wandb.init`` and pass project name and config.
- Log metrics using `wandb.log`` during training.

****"wandb" Cloud Platform and Login****

- Use "wandb" Cloud Platform for this course.
- Log in to "wandb" using personal account and API key.
- Login enables experiment tracking and result saving.

****Training the Model****

- Run training code to observe progress.
- Data is logged to "wandb" server for result storage.

****Visualization and Comparison on "wandb"****

- Access "wandb" project page to view results.
- View training loss and validation metrics.
- Compare different runs for performance analysis.

****Hyperparameter Tuning****

- Experiment with hyperparameters to improve model.
- Modify hyperparameters for better performance.

****Comparing Experiments****

- Use project page to compare experiment results.
- Hover over runs to view training curves.
- Utilize runs table for side-by-side metric and hyperparameter comparison.

****Filtering and Sorting Experiments****

- Apply filters to focus on specific runs.
- Sort runs by metrics to identify top performers.

****Detailed Run Overview****

- View details of specific run in detail view.
- Access Git repo and commit hash for code reference.
- Capture uncommitted changes in diff patch.
- Config captures settings for reproducibility and communication.

****Conclusion and Next Lesson****

- Summary of lesson content.
- Teaser for next lesson on generative AI model training with "wandb."

Introduction to W&B

We will add `wandb` to sprite classification model training, so that we can track and visualize important metrics, gain insights into our model's behavior and make informed decisions for model improvements. We will also see how to compare and analyze different experiments, collaborate with team members, and reproduce results effectively.

```
In [ ]: import math
        from pathlib import Path
        from types import SimpleNamespace
        from tqdm.auto import tqdm
        import torch
        import torch.nn as nn
        import torch.nn.functional as F
        from torch.optim import Adam
        from utilities import get_dataloaders

        import wandb
```

Sprite classification

We will build a simple model to classify sprites. You can see some examples of sprites and corresponding classes in the image below.



```
In [ ]: INPUT_SIZE = 3 * 16 * 16
        OUTPUT_SIZE = 5
        HIDDEN_SIZE = 256
        NUM_WORKERS = 2
        CLASSES = ["hero", "non-hero", "food", "spell", "side-facing"]
        DATA_DIR = Path('./data/')
        DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")

        def get_model(dropout):
            "Simple MLP with Dropout"
            return nn.Sequential(
                nn.Flatten(),
                nn.Linear(INPUT_SIZE, HIDDEN_SIZE),
                nn.BatchNorm1d(HIDDEN_SIZE),
                nn.ReLU(),
                nn.Dropout(dropout),
                nn.Linear(HIDDEN_SIZE, OUTPUT_SIZE)
            ).to(DEVICE)
```

```
In [ ]: def validate_model(model, valid_dl, loss_func):
        "Compute the performance of the model on the validation dataset"
        model.eval()
        val_loss = 0.0
        correct = 0

        with torch.inference_mode():
            for i, (images, labels) in enumerate(valid_dl):
                images, labels = images.to(DEVICE), labels.to(DEVICE)

                # Forward pass
                outputs = model(images)
                val_loss += loss_func(outputs, labels) * labels.size(0)

                # Compute accuracy and accumulate
                _, predicted = torch.max(outputs.data, 1)
                correct += (predicted == labels).sum().item()

        return val_loss / len(valid_dl.dataset), correct / len(valid_dl.dataset)
```

W&B account

[Sign up](https://wandb.ai/site) for a free account at <https://wandb.ai/site> and then login to your wandb account to store the results of your experiments and use advanced W&B features. You can also continue to learn in anonymous mode. If you have an existing W&B account, and your browser automatically logs you in, be sure to use it here to avoid confusion.

```
In [ ]: wandb.login(anonymous="allow")
```

Train model

Let's train the model with default config and check how it's doing in W&B.

```
In [ ]: train_model(config)
```

```
In [ ]: # So let's change the learning rate to a 1e-3
        # and see how this affects our results.
        config.lr = 1e-4
        train_model(config)
```

```
In [ ]: config.lr = 1e-4
        train_model(config)
```

```
In [ ]: config.dropout = 0.1
        config.epochs = 1
        train_model(config)
```

```
In [ ]: config.lr = 1e-3
        train_model(config)
```

```
In [ ]: # Let's define a config object to store our hyperparameters
config = SimpleNamespace(
    epochs = 2,
    batch_size = 128,
    lr = 1e-5,
    dropout = 0.5,
    slice_size = 10_000,
    valid_pct = 0.2,
)
```

```
In [ ]: def train_model(config):
    "Train a model with a given config"

    wandb.init(
        project="dlai_intro",
        config=config,
    )

    # Get the data
    train_dl, valid_dl = get_dataloaders(DATA_DIR,
                                         config.batch_size,
                                         config.slice_size,
                                         config.valid_pct)

    n_steps_per_epoch = math.ceil(len(train_dl.dataset) / config.batch_size)

    # A simple MLP model
    model = get_model(config.dropout)

    # Make the Loss and optimizer
    loss_func = nn.CrossEntropyLoss()
    optimizer = Adam(model.parameters(), lr=config.lr)

    example_ct = 0

    for epoch in tqdm(range(config.epochs), total=config.epochs):
        model.train()

        for step, (images, labels) in enumerate(train_dl):
            images, labels = images.to(DEVICE), labels.to(DEVICE)

            outputs = model(images)
            train_loss = loss_func(outputs, labels)
            optimizer.zero_grad()
            train_loss.backward()
            optimizer.step()

            example_ct += len(images)
            metrics = {
                "train/train_loss": train_loss,
                "train/epoch": epoch + 1,
                "train/example_ct": example_ct
            }
            wandb.log(metrics)

        # Compute validation metrics, Log images on last epoch
        val_loss, accuracy = validate_model(model, valid_dl, loss_func)
        # Compute train and validation metrics
        val_metrics = {
            "val/val_loss": val_loss,
            "val/val_accuracy": accuracy
        }
        wandb.log(val_metrics)

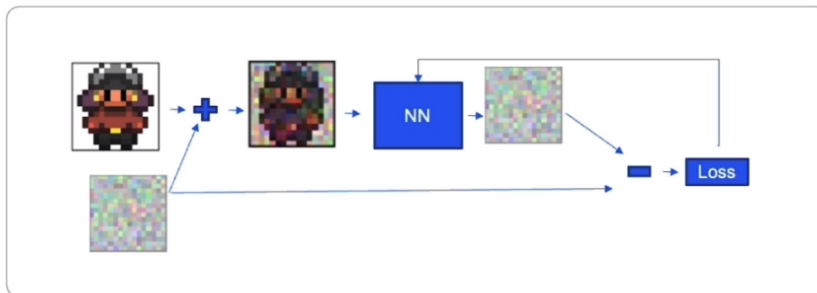
    wandb.finish()
```


Lesson 2: Training Diffusion Models with "wandb"

Training a Diffusion Model

Tracking progress with W&B

- ❖ NN learns to predict noise—really learns the distribution of what is not noise
- ❖ Sample random timestep (noise level) per image to train more stably.



- ❖ A diffusion model learns how to iteratively remove small amounts of noise from an image
- ❖ We use the same code as on the [by Deeplearning.ai, so let's just](#) "How Diffusion Models Work" course.

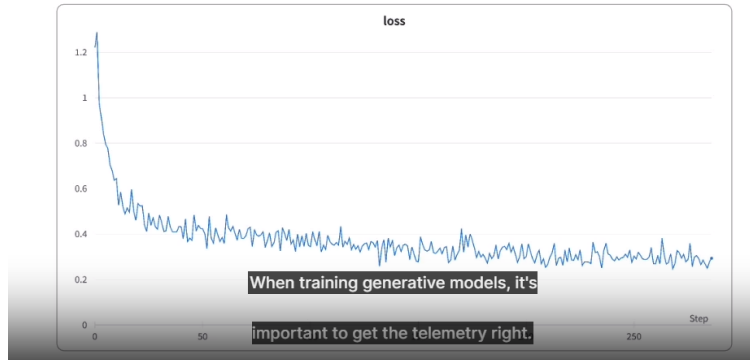
Diffusion Models Overview

- Diffusion models are denoising models.
- Trained to remove noise from images, not generate them.
- Noise added to images following a scheduler, model predicts noise.
- Samples generated by removing noise iteratively.

Training a Diffusion Model

Tracking progress with W&B

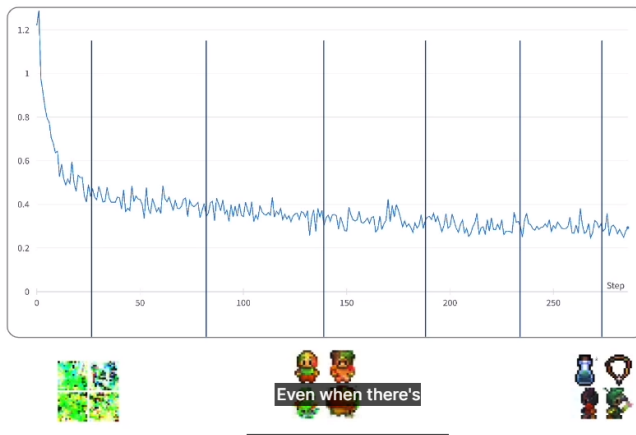
- ❖ Telemetry is very important when it comes to training generative models.
- ❖ For the diffusion training we can:
 - Keep track of the loss and relevant metrics



Training a Diffusion Model

Tracking progress with W&B

- ❖ For the diffusion training we can:
 - Keep track of the loss and relevant metrics
 - Sample images from the model during training
 - Safely store and version model checkpoints



Importance of Telemetry

- Metrics like loss curve are important but might not reflect image quality.
- Regularly sample from the model during training for better insight.
- Image quality might improve even when loss plateaus.

****Logging Samples and Model Checkpoints****

- Uploading samples to "wandb" for visualization.
- Saving model checkpoints for organization.

****Notebook Overview****

- Using DeepLearning.ai's diffusion model training notebook.
- Importing relevant libraries and "wandb."
- Creating an account for result tracking (or anonymous logging).

****Environment Variables Setup****

- Defining paths for model and checkpoint storage.
- Utilizing CUDA GPU if available.

****Hyperparameters Setup****

- Using a simple namespace to set varying hyperparameters.
- Importing "ddpm" noise scheduler and sampler.

****Creating the Neural Network****

- Creating the neural network to be trained.

****Data Loading and Optimization****

- Using a sample dataset.
- Creating a data loader and setting up an optimizer.

****Training Loop Setup****

- Choosing noise for sampling.
- Preparing for the training loop.

****Training Phase****

- Initializing a "wandb" run to track training.
- Passing project name, classification, and job type.
- Logging configuration and passing `wandb.config`.

****Standard Training Loop****

- Running forward and backward passes for several epochs.
- Logging metrics to "wandb," including loss and learning rate.

****Saving Model Checkpoints****

- Saving model checkpoints every few epochs.
- Using "wandb" artifact to version and store files.

****Image Logging****

- Logging sample images using "wandb.log" and "wandb.image."

****Finishing the Run****

- Calling `wandb.finish` to conclude the run.

****Visualizing Training Progress****

- Viewing loss curve and sample images in "wandb" workspace.

****Model Registry****

- Linking the trained model to the Model Registry.
- Centralized location for best model versions.
- Lineage tracking and Git commit reference.

****Conclusion and Next Lesson****

- Recap of lesson content.
- Teaser for the next lesson on sampling a diffusion model

Training a Diffusion Model with Weights and Biases (W&B) ¶

In this notebook we will instrument the training of a diffusion model with W&B. We will use the Lab3 notebook from the ["How diffusion models work"](#) course. We will add:

- Logging of the training loss and metrics
- Sampling from the model during training and uploading the samples to W&B
- Saving the model checkpoints to W&B

```
In [ ]: from types import SimpleNamespace
        from pathlib import Path
        from tqdm.notebook import tqdm
        import torch
        import torch.nn.functional as F
        from torch.utils.data import DataLoader
        import numpy as np
        from utilities import *

        import wandb
```

We encourage you to create an account to get the full user experience from W&B

```
In [ ]: wandb.login(anonymous="allow")
```

Setting Things Up

```
In [ ]: # we are storing the parameters to be logged to wandb
DATA_DIR = Path('./data/')
SAVE_DIR = Path('./data/weights/')
SAVE_DIR.mkdir(exist_ok=True, parents=True)
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"

config = SimpleNamespace(
    # hyperparameters
    num_samples = 30,

    # diffusion hyperparameters
    timesteps = 500,
    beta1 = 1e-4,
    beta2 = 0.02,

    # network hyperparameters
    n_feat = 64, # 64 hidden dimension feature
    n_cfeat = 5, # context vector is of size 5
    height = 16, # 16x16 image

    # training hyperparameters
    batch_size = 100,
    n_epoch = 32,
    lrate = 1e-3,
)
```

Setup DDPM noise scheduler and sampler (same as in the Diffusion course).

- `perturb_input`: Adds noise to the input image at the corresponding timestep on the schedule
- `sample_ddpm_context`: Generate images using the DDPM sampler, we will use this function during training to sample from the model regularly and see how our training is progressing

```
In [ ]: # setup ddpm sampler functions
        perturb_input, sample_ddpm_context = setup_ddpm(config.beta1,
                                                         config.beta2,
                                                         config.timesteps,
                                                         DEVICE)
```

```
In [ ]: # construct model
        nn_model = ContextUnet(in_channels=3,
                               n_feat=config.n_feat,
                               n_cfeat=config.n_cfeat,
                               height=config.height).to(DEVICE)
```

```
In [ ]: # Load dataset and construct optimizer
        dataset = CustomDataset.from_np(path=DATA_DIR)
        dataloader = DataLoader(dataset,
                                batch_size=config.batch_size,
                                shuffle=True)
        optim = torch.optim.Adam(nn_model.parameters(), lr=config.lrate)
```

Training

We choose a fixed context vector with 6 samples of each class to guide our diffusion

```
In [ ]: # Noise vector
        #  $x_T \sim N(0, 1)$ , sample initial noise
        noises = torch.randn(config.num_samples, 3,
                              config.height, config.height).to(DEVICE)

        # A fixed context vector to sample from
        ctx_vector = F.one_hot(torch.tensor([0,0,0,0,0,0,
                                             1,1,1,1,1,1,
                                             2,2,2,2,2,2,
                                             3,3,3,3,3,3,
                                             4,4,4,4,4,4]), # hero
                              # non-hero
                              # food
                              # spell
                              # side-facing
                              5).to(DEVICE).float())
```

The following training cell takes very long to run on CPU, we have already trained the model for you on a GPU equipped machine.

```

In [ ]: # create a wandb run
run = wandb.init(project="dlai_sprite_diffusion",
                 job_type="train",
                 config=config)

# we pass the config back from W&B
config = wandb.config

for ep in tqdm(range(config.n_epoch), leave=True, total=config.n_epoch):
    # set into train mode
    nn_model.train()
    optim.param_groups[0]['lr'] = config.lr*(1-ep/config.n_epoch)

    pbar = tqdm(dataloader, leave=False)
    for x, c in pbar: # x: images c: context
        optim.zero_grad()
        x = x.to(DEVICE)
        c = c.to(DEVICE)
        context_mask = torch.bernoulli(torch.zeros(c.shape[0]) + 0.8).to(DEVICE)
        c = c * context_mask.unsqueeze(-1)
        noise = torch.randn_like(x)
        t = torch.randint(1, config.timesteps + 1, (x.shape[0],)).to(DEVICE)
        x_pert = perturb_input(x, t, noise)
        pred_noise = nn_model(x_pert, t / config.timesteps, c=c)
        loss = F.mse_loss(pred_noise, noise)
        loss.backward()
        optim.step()

        wandb.log({"loss": loss.item(),
                  "lr": optim.param_groups[0]['lr'],
                  "epoch": ep})

    # save model periodically
    if ep%4==0 or ep == int(config.n_epoch-1):
        nn_model.eval()
        ckpt_file = SAVE_DIR/f"context_model.pth"
        torch.save(nn_model.state_dict(), ckpt_file)

        artifact_name = f"{wandb.run.id}_context_model"
        at = wandb.Artifact(artifact_name, type="model")
        at.add_file(ckpt_file)
        wandb.log_artifact(at, aliases=[f"epoch_{ep}"])

        samples, _ = sample_ddpm_context(nn_model,
                                         noises,
                                         ctx_vector[:config.num_samples])

        wandb.log({
            "train_samples": [
                wandb.Image(img) for img in samples.split(1)
            ]
        })

# finish W&B run
wandb.finish()

```

****Lesson 3: Comparing Diffusion Model Outputs and Evaluating LLM Models****

Comparing model outputs

Managing Models

Model registry

a central system of record for your models

- Publish production-ready models
- Move model versions through the lifecycle from staging to production
- Collaborate on models across teams
- Audit model lineage across training, evaluation and production
- Automate downstream actions

"We will evaluate `staging` model from the registry"

Comparing model outputs

Visualizing samples

W&B Tables

- Log, query, and analyze tabular data including rich media: images, videos, molecules, etc.
- Compare changes precisely across models,

```
table = wandb.Table(columns=['col1', ...])
table.add_data(...)
wandb.log({'predictions': table})
```

****Introduction****

- Comparing diffusion model outputs in this lesson.
- Starting with the model trained in the previous lesson.

****Model Registry Overview****

- Model Registry as central system for machine learning models.
- Manages lifecycle from staging to production.
- Detailed lineage tracking during training, evaluation, production.
- Automates downstream tasks for efficiency.

****Tables for Comparison and Evaluation****

- Using tables for data logging, query, analysis.
- Create a table, define columns, update rows, log with "wandb.log."

****Pulling Model from Registry****

- Pulling model from Model Registry using "wandb.Api."
- Retrieving model and run information.

****Loading Model Weights****

- Loading model weights from artifact.
- Recreating model using original parameters.

****Diffusion Sampler Setup****

- Setting up "ddpm" diffusion sampler.
- Defining fixed noises and context vector.

****Comparing "ddpm" and "ddim" Samplers****

- Importing another sampler, "ddim," for comparison.
- Generating samples using both samplers.

****Creating a Visual Table****

- Constructing a table for visual comparison.
- Adding rows with images, class name, input noise.

****Logging the Table****

- Calling "wandb.init" with project name and job type.
- Setting table name and logging it with "wandb.log."

****Visualizing Comparison Results****

- Opening run to view the uploaded table.
- Exploring rows with sample images and information.

****Grouping and Filtering Samples****

- Grouping images by class for side-by-side comparison.
- Hiding unnecessary columns for better visualization.

****Creating and Sharing a Report****

- Creating a report with the sample table.
- Adding context and notes to explain findings.
- Publishing report to make it available for colleagues.

****Conclusion and Next Lesson****

- Summary of lesson content.
- Teaser for the next lesson on evaluating an LLM model.

Sampling from a diffusion model

In this notebook we will sample from the previously trained diffusion model.

- We are going to compare the samples from DDPM and DDIM samplers
- Visualize mixing samples with conditional diffusion models

```
In [ ]: from pathlib import Path
        from types import SimpleNamespace
        import torch
        import torch.nn.functional as F
        import numpy as np
        from utilities import *

        import wandb
```

```
In [ ]: wandb.login(anonymous="allow")
```

Setting Things Up

```
In [ ]: # Wandb Params
MODEL_ARTIFACT = "dlai-course/model-registry/SpriteGen:latest"
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"

config = SimpleNamespace(
    # hyperparameters
    num_samples = 30,

    # ddpm sampler hyperparameters
    timesteps = 500,
    beta1 = 1e-4,
    beta2 = 0.02,

    # ddim sampler hp
    ddim_n = 25,

    # network hyperparameters
    height = 16,
)
```

In the previous notebook we saved the best model as a wandb Artifact (our way of storing files during runs). We will now load the model from wandb and set up the sampling loop.

```
In [ ]: def load_model(model_artifact_name):
        """Load the model from wandb artifacts"""
        api = wandb.Api()
        artifact = api.artifact(model_artifact_name, type="model")
        model_path = Path(artifact.download())

        # recover model info from the registry
        producer_run = artifact.logged_by()

        # Load the weights dictionary
        model_weights = torch.load(model_path/"context_model.pth",
                                   map_location="cpu")

        # create the model
        model = ContextUnet(in_channels=3,
                            n_feat=producer_run.config["n_feat"],
                            n_cfeat=producer_run.config["n_cfeat"],
                            height=producer_run.config["height"])

        # Load the weights into the model
        model.load_state_dict(model_weights)

        # set the model to eval mode
        model.eval()
        return model.to(DEVICE)
```

```
In [ ]: nn_model = load_model(MODEL_ARTIFACT)
```

Sampling

We will sample and log the generated samples to wandb.

```
In [ ]: _, sample_ddpm_context = setup_ddpm(config.beta1,
                                             config.beta2,
                                             config.timesteps,
                                             DEVICE)
```

Let's define a set of noises and a context vector to condition on.

```
In [ ]: # Noise vector
#  $x_T \sim N(0, 1)$ , sample initial noise
noises = torch.randn(config.num_samples, 3,
                     config.height, config.height).to(DEVICE)

# A fixed context vector to sample from
ctx_vector = F.one_hot(torch.tensor([0,0,0,0,0,0, # hero
                                   1,1,1,1,1,1, # non-hero
                                   2,2,2,2,2,2, # food
                                   3,3,3,3,3,3, # spell
                                   4,4,4,4,4,4]), # side-facing
                      5).to(DEVICE).float()
```

Let's bring that faster DDIM sampler from the diffusion course.

```
In [ ]: sample_ddim_context = setup_ddim(config.beta1,
                                          config.beta2,
                                          config.timesteps,
                                          DEVICE)
```

Sampling:

let's compute ddpm samples as before

```
In [ ]: ddpm_samples, _ = sample_ddpm_context(nn_model, noises, ctx_vector)
```

For DDIM we can control the step size by the `n` param:

```
In [ ]: ddim_samples, _ = sample_ddim_context(nn_model,
                                              noises,
                                              ctx_vector,
                                              n=config.ddim_n)
```

Visualizing generations on a Table

Let's create a `wandb.Table` to store our generations

```
In [ ]: table = wandb.Table(columns=["input_noise", "ddpm", "ddim", "class"])
```

We can add the rows to the table one by one, we also cast images to `wandb.Image` so we can render them correctly in the UI

```
In [ ]: for noise, ddpm_s, ddim_s, c in zip(noises,
                                             ddpm_samples,
                                             ddim_samples,
                                             to_classes(ctx_vector)):

    # add data row by row to the Table
    table.add_data(wandb.Image(noise),
                  wandb.Image(ddpm_s),
                  wandb.Image(ddim_s),
                  c)
```

we log the table to W&B, we can also use `wandb.init` as a context manager, this way we ensure that the run is finished when exiting the manager.

```
In [ ]: with wandb.init(project="dlai_sprite_diffusion",
                       job_type="samplers_battle",
                       config=config):

    wandb.log({"samplers_table":table})
```

****Lesson 4: Evaluating Large Language Models (LLMs)****

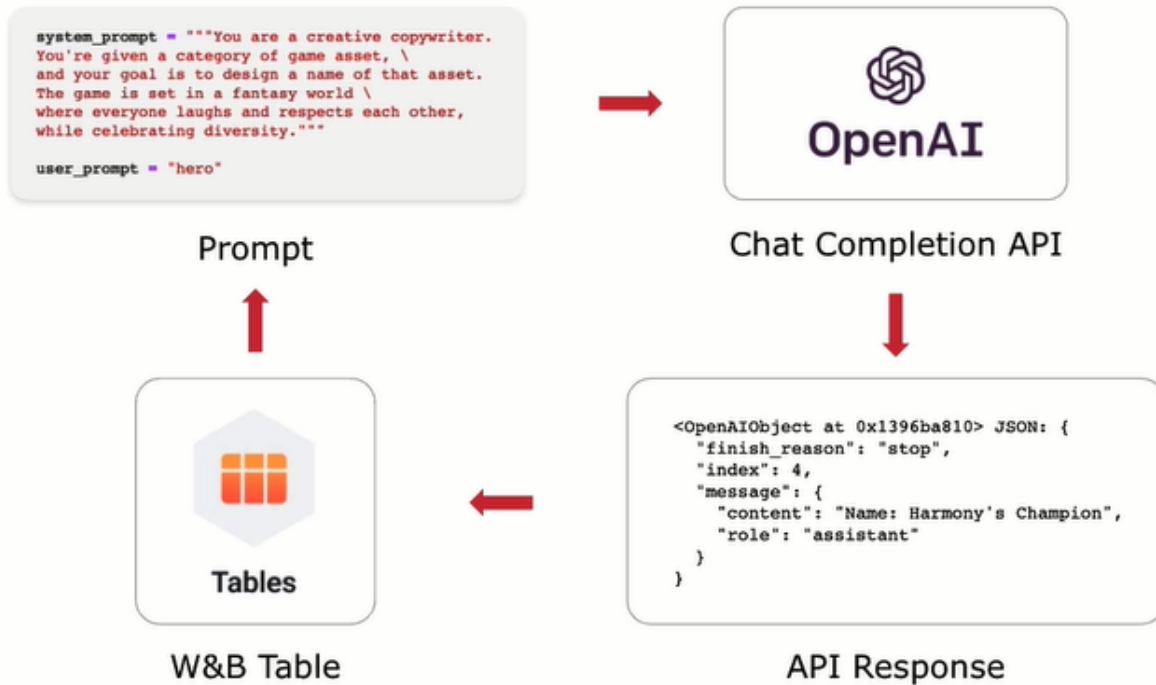
Evaluating LLMs

Using Weights & Biases Prompts

- 1.** Using APIs with Tables
- 2.** Tracking LLM chain spans with Tracer
- 3.** Tracking Langchain Agents

Evaluating LLMs

Calling OpenAI APIs



****Example 1: Using API for LLM Evaluation****

- Designing system and user prompts.
- Calling OpenAI API using chat completion.
- Parsing and logging results with "wandb" tables.

Evaluating LLMs

Tracking LLM chain spans with Tracer

1. Pick a virtual world

- input
- output
- start time
- end time result
- status

Trace
World
Picker

2. Generate description

- input
- output
- start time
- end time result
- status

Trace
OpenAI

Trace
MyChain

****Example 2: Tracing LLM Chains with a Tool Called Tracer****

- Creating a custom LLM chain.
- Using Tracer for tracking and debugging complex chains.
- Illustrating chain concept with World Picker and Name Generator.

Evaluating LLMs

Tracking Langchain Agent

ReAct Agent: looping through reasoning (what should I do), Actions (using tools), Observations (what have I learned)

WorldPicker

pick a virtual game world for your character or item naming

NameValidator

validate if the name is properly generated

LLM

ChatOpenAI()

```
# enable wandb tracing
os.environ["LANGCHAIN_WANDB_TRACING"] =
"true"

# run langchain agent
agent.run(
    "Find a virtual game world for me and
    imagine the name of a hero in that world"
)
```

chain where each step

****Example 3: LLM Chains with LangChain Agents****

- Introducing LangChain agent concept.
- Using WorldPicker and NameValidator tools.
- Running queries and analyzing results.

****Conclusion and Teaser****

- Summarizing lesson content.
- Teasing the next lesson on fine-tuning LLMs.

LLM Evaluation and Tracing with W&B

1. Using Tables for Evaluation

In this section, we will call OpenAI LLM to generate names of our game assets. We will use W&B Tables to evaluate the generations.

```
In [ ]: import os
import random
import time
import datetime

import openai

from tenacity import (
    retry,
    stop_after_attempt,
    wait_random_exponential, # for exponential backoff
)
import wandb
from wandb.sdk.data_types.trace_tree import Trace
```

```
In [ ]: # get openai API key
import os
import openai
import sys
sys.path.append('../..')

from dotenv import load_dotenv, find_dotenv
_ = load_dotenv(find_dotenv()) # read local .env file

openai.api_key = os.environ['OPENAI_API_KEY']
```

```
In [ ]: PROJECT = "dlai_llm"
MODEL_NAME = "gpt-3.5-turbo"
```

```
In [ ]: wandb.login(anonymous="allow")
```

```
In [ ]: run = wandb.init(project=PROJECT, job_type="generation")
```

Simple generations

Let's start by generating names for our game assets using OpenAI `ChatCompletion`, and saving the resulting generations in W&B Tables.

```
In [ ]: @retry(wait=wait_random_exponential(min=1, max=60), stop=stop_after_attempt(6))
def completion_with_backoff(**kwargs):
    return openai.ChatCompletion.create(**kwargs)
```

```
In [ ]: def generate_and_print(system_prompt, user_prompt, table, n=5):
    messages=[
        {"role": "system", "content": system_prompt},
        {"role": "user", "content": user_prompt},
    ]
    start_time = time.time()
    responses = completion_with_backoff(
        model=MODEL_NAME,
        messages=messages,
        n = n,
    )
    elapsed_time = time.time() - start_time
    for response in responses.choices:
        generation = response.message.content
        print(generation)
    table.add_data(system_prompt,
        user_prompt,
        [response.message.content for response in responses.choices],
        elapsed_time,
        datetime.datetime.fromtimestamp(responses.created),
        responses.model,
        responses.usage.prompt_tokens,
        responses.usage.completion_tokens,
        responses.usage.total_tokens
    )
```

```

In [ ]: system_prompt = """You are a creative copywriter.
You're given a category of game asset, \
and your goal is to design a name of that asset.
The game is set in a fantasy world \
where everyone laughs and respects each other,
while celebrating diversity."""

In [ ]: # Define W&B Table to store generations
columns = ["system_prompt", "user_prompt", "generations", "elapsed_time", "timestamp", \
          "model", "prompt_tokens", "completion_tokens", "total_tokens"]
table = wandb.Table(columns=columns)

In [ ]: user_prompt = "hero"
generate_and_print(system_prompt, user_prompt, table)

In [ ]: user_prompt = "jewel"
generate_and_print(system_prompt, user_prompt, table)

In [ ]: wandb.log({"simple_generations": table})
run.finish()

```

2. Using Tracer to log more complex chains

How can we get more creative outputs? Let's design an LLM chain that will first randomly pick a fantasy world, and then generate character names. We will demonstrate how to use Tracer in such scenario. We will log the inputs and outputs, start and end times, whether the OpenAI call was successful, the token usage, and additional metadata.

```

In [ ]: worlds = [
    "a mystic medieval island inhabited by intelligent and funny frogs",
    "a modern castle sitting on top of a volcano in a faraway galaxy",
    "a digital world inhabited by friendly machine learning engineers"
]

In [ ]: # define your config
model_name = "gpt-3.5-turbo"
temperature = 0.7
system_message = """You are a creative copywriter.
You're given a category of game asset and a fantasy world.
Your goal is to design a name of that asset.
Provide the resulting name only, no additional description.
Single name, max 3 words output, remember!"""

```

```

In [ ]: def run_creative_chain(query):
    # part 1 - a chain is started...
    start_time_ms = round(datetime.datetime.now().timestamp() * 1000)

    root_span = Trace(
        name="MyCreativeChain",
        kind="chain",
        start_time_ms=start_time_ms,
        metadata={"user": "student_1"},
        model_dict={"_kind": "CreativeChain"}
    )

    # part 2 - your chain picks a fantasy world
    time.sleep(3)
    world = random.choice(worlds)
    expanded_prompt = f'Game asset category: {query}; fantasy world description: {world}'
    tool_end_time_ms = round(datetime.datetime.now().timestamp() * 1000)

    # create a Tool span
    tool_span = Trace(
        name="WorldPicker",
        kind="tool",
        status_code="success",
        start_time_ms=start_time_ms,
        end_time_ms=tool_end_time_ms,
        inputs={"input": query},
        outputs={"result": expanded_prompt},
        model_dict={"_kind": "tool", "num_worlds": len(worlds)}
    )

    # add the TOOL span as a child of the root
    root_span.add_child(tool_span)

```

```

# part 3 - the LLMChain calls an OpenAI LLM...
messages=[
    {"role": "system", "content": system_message},
    {"role": "user", "content": expanded_prompt}
]

response = completion_with_backoff(model=model_name,
                                   messages=messages,
                                   max_tokens=12,
                                   temperature=temperature)

llm_end_time_ms = round(datetime.datetime.now().timestamp() * 1000)
response_text = response["choices"][0]["message"]["content"]
token_usage = response["usage"].to_dict()

llm_span = Trace(
    name="OpenAI",
    kind="llm",
    status_code="success",
    metadata={"temperature":temperature,
             "token_usage": token_usage,
             "model_name":model_name},
    start_time_ms=tool_end_time_ms,
    end_time_ms=llm_end_time_ms,
    inputs={"system_prompt":system_message, "query":expanded_prompt},
    outputs={"response": response_text},
    model_dict={"_kind": "Openai", "engine": response["model"], "model": response["object"]}
)

# add the LLM span as a child of the Chain span...
root_span.add_child(llm_span)

# update the end time of the Chain span
root_span.add_inputs_and_outputs(
    inputs={"query":query},
    outputs={"response": response_text})

# update the Chain span's end time
root_span.end_time_ms = llm_end_time_ms

# part 4 - log all spans to W&B by logging the root span
root_span.log(name="creative_trace")
print(f"Result: {response_text}")

```

```

In [ ]: # Let's start a new wandb run
        wandb.init(project=PROJECT, job_type="generation")

```

```

In [ ]: run_creative_chain("hero")

```

```

In [ ]: run_creative_chain("jewel")

```

```

In [ ]: wandb.finish()

```

Langchain agent

In the third scenario, we'll introduce an agent that will use tools such as WorldPicker and NameValidator to come up with the ultimate name. We will also use Langchain here and demonstrate its W&B integration.

```
In [ ]: # Import things that are needed generically
from langchain.agents import AgentType, initialize_agent
from langchain.chat_models import ChatOpenAI
from langchain.tools import BaseTool

from typing import Optional

from langchain.callbacks.manager import (
    AsyncCallbackManagerForToolRun,
    CallbackManagerForToolRun,
)
```

```
In [ ]: wandb.init(project=PROJECT, job_type="generation")
```

```
In [ ]: os.environ["LANGCHAIN_WANDB_TRACING"] = "true"
```

```
In [ ]: class WorldPickerTool(BaseTool):
    name = "pick_world"
    description = "pick a virtual game world for your character or item naming"
    worlds = [
        "a mystic medieval island inhabited by intelligent and funny frogs",
        "a modern anthill featuring a cyber-ant queen and her cyber-ant-workers",
        "a digital world inhabited by friendly machine learning engineers"
    ]

    def _run(
        self, query: str, run_manager: Optional[CallbackManagerForToolRun] = None
    ) -> str:
        """Use the tool."""
        time.sleep(1)
        return random.choice(self.worlds)

    async def _arun(
        self, query: str, run_manager: Optional[AsyncCallbackManagerForToolRun] = None
    ) -> str:
        """Use the tool asynchronously."""
        raise NotImplementedError("pick_world does not support async")

class NameValidatorTool(BaseTool):
    name = "validate_name"
    description = "validate if the name is properly generated"

    def _run(
        self, query: str, run_manager: Optional[CallbackManagerForToolRun] = None
    ) -> str:
        """Use the tool."""
        time.sleep(1)
        if len(query) < 20:
            return f"This is a correct name: {query}"
        else:
            return f"This name is too long. It should be shorter than 20 characters."

    async def _arun(
        self, query: str, run_manager: Optional[AsyncCallbackManagerForToolRun] = None
    ) -> str:
        """Use the tool asynchronously."""
        raise NotImplementedError("validate_name does not support async")
```

```
In [ ]: llm = ChatOpenAI(temperature=0.7)
```

```
In [ ]: tools = [WorldPickerTool(), NameValidatorTool()]
agent = initialize_agent(
    tools,
    llm,
    agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
    handle_parsing_errors=True,
    verbose=True
)
```

```
In [ ]: agent.run(
    "Find a virtual game world for me and imagine the name of a hero in that world"
)
```

```
In [ ]: agent.run(
    "Find a virtual game world for me and imagine the name of a jewel in that world"
)
```

```
In [ ]: agent.run(
    "Find a virtual game world for me and imagine the name of food in that world."
)
```

```
In [ ]: wandb.finish()
```

****Lesson 5: Fine-Tuning Large Language Models (LLMs)****

Training & Finetuning LLMs

Using Weights & Biases

Training from scratch

- 1.** Long & expensive training runs
- 2.** Expensive & difficult evaluations
- 3.** Monitoring is critical
- 4.** Ability to restore training from a checkpoint

Fine-tuning

- 1.** Efficient methods being developed
- 2.** Expensive & difficult evaluations

****Introduction****

- Discussing the need for fine-tuning or training new LLMs.
- Emphasizing the importance of debugging and evaluation during the process.

****Training LLMs from Scratch****

- Training LLMs from scratch is time-consuming and resource-intensive.
- Monitoring training progress, metrics, and using checkpoints.
- Utilizing Weights and Biases dashboard for insights and checkpoints.

****Fine-Tuning LLMs****

- Fine-tuning is more economical and feasible.
- Careful evaluation process is still crucial.
- Tailoring evaluation strategies based on intended model usage.

****Fine-Tuning Example with Hugging Face****

- Fine-tuning a small language model (TinyStories) on character backstories.
- Importing necessary libraries and logging in.
- Pulling dataset from Hugging Face hub and examining its structure.
- Preparing dataset for training by tokenizing and padding.
- Creating a causal language model for autoregressive language modeling.
- Setting up training arguments, streaming metrics to Weights and Biases.

****Fine-Tuning Process****

- Starting a new Weights and Biases run for training.
- Monitoring training progress with live metrics.
- Generating samples from the trained model and evaluating results.
- Using qualitative evaluation and potential metrics like unique words.

****Conclusion and Next Lesson****

- Recap of lesson content.
- Introduction to the next lesson on deploying LLMs to production.

Finetuning a language model

Let's see how to finetune a language model to generate character backstories using HuggingFace Trainer with wandb integration. We'll use a tiny language model (TinyStories-33M) due to resource constraints, but the lessons you learn here should be applicable to large models too!

```
In [ ]: from transformers import AutoTokenizer
        from datasets import load_dataset
        from transformers import AutoModelForCausalLM
        from transformers import Trainer, TrainingArguments
        import transformers
        transformers.set_seed(42)

        import wandb
```

```
In [ ]: wandb.login(anonymous="allow")
```

```
In [ ]: model_checkpoint = "roneneldan/TinyStories-33M"
```


Preparing data

We'll start by loading a dataset containing Dungeons and Dragons character biographies from Huggingface.

You can expect to get some warning here, this is ok

```
In [ ]: ds = load_dataset('MohamedRashad/characters_backstories')

In [ ]: # Let's take a look at one example
ds["train"][400]

In [ ]: # As this dataset has no validation split, we will create one
ds = ds["train"].train_test_split(test_size=0.2, seed=42)

In [ ]: # We'll create a tokenizer from model checkpoint
tokenizer = AutoTokenizer.from_pretrained(model_checkpoint, use_fast=False)

# We'll need padding to have same length sequences in a batch
tokenizer.pad_token = tokenizer.eos_token

# Define a tokenization function that first concatenates text and target
def tokenize_function(example):
    merged = example["text"] + " " + example["target"]
    batch = tokenizer(merged, padding='max_length', truncation=True, max_length=128)
    batch["labels"] = batch["input_ids"].copy()
    return batch

# Apply it on our dataset, and remove the text columns
tokenized_datasets = ds.map(tokenize_function, remove_columns=["text", "target"])

In [ ]: # Let's check out one prepared example
print(tokenizer.decode(tokenized_datasets["train"][900]['input_ids']))
```

Training

Let's finetune a pretrained language model on our dataset using HF Transformers and their wandb integration.

```
In [ ]: # We will train a causal (autoregressive) language model from a pretrained checkpoint
model = AutoModelForCausalLM.from_pretrained(model_checkpoint);

In [ ]: # Start a new wandb run
run = wandb.init(project='dlai_lm_tuning', job_type="training", anonymous="allow")

In [ ]: # Define training arguments
model_name = model_checkpoint.split("/")[-1]
training_args = TrainingArguments(
    f'{model_name}-finetuned-characters-backstories',
    report_to="wandb", # we need one line to track experiments in wandb
    num_train_epochs=1,
    logging_steps=1,
    evaluation_strategy = "epoch",
    learning_rate=1e-4,
    weight_decay=0.01,
    no_cuda=True, # force cpu use, will be renamed `use_cpu`
)

In [ ]: # We'll use HF Trainer
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_datasets["train"],
    eval_dataset=tokenized_datasets["test"],
)

In [ ]: # Let's train!
trainer.train()
```

```
In [ ]: transformers.logging.set_verbosity_error() # suppress tokenizer warnings

prefix = "Generate Backstory based on following information Character Name: "

prompts = [
    "Frogger Character Race: Aarakocra Character Class: Ranger Output: ",
    "Smarty Character Race: Aasimar Character Class: Cleric Output: ",
    "Volcano Character Race: Android Character Class: Paladin Output: ",
]

table = wandb.Table(columns=["prompt", "generation"])

for prompt in prompts:
    input_ids = tokenizer.encode(prefix + prompt, return_tensors="pt")
    output = model.generate(input_ids, do_sample=True, max_new_tokens=50, top_p=0.3)
    output_text = tokenizer.decode(output[0], skip_special_tokens=True)
    table.add_data(prefix + prompt, output_text)

wandb.log({'tiny_generations': table})
```

Note: LLM's don't always generate the same results. Your generated characters and backstories may differ from the video.

```
In [ ]: wandb.finish()
```