Understanding Diffusion Models Through Conditional Sine Wave Generation - release 2025.1

joepareti54@gmail.com 25. Feb. 2025

Introduction

This is the next release following the 2025-release0.

It is about a development program to mimic the diffusion process in Alpha Fold 3, which improves on an initial attempt by means of a training step and of a more realistic conditioning algorithm, while continuing to be super-simplified compared to Alpha Fold 3

Code - new_code_learn_conditioning_0003.py -

```
"""
Conditional Diffusion Model for Sine Wave Generation

A minimal implementation of a conditional diffusion model for generating sine
waves.
```

```
Uses pure input data approach similar to AF3, with improved training
stability.

The program implements:
1. A neural network (ConditionedDenoiseModel) that learns to denoise data
2. A diffusion process (Diffusion) that handles noise addition and removal
3. Training data generation and model training utilities
4. Visualization of results

Key Features:
- 1000-step diffusion process
- Enhanced neural network architecture (101→256→256→256→100)
- Improved training stability with gradient clipping
- Multiple training steps per batch
- Pure input data approach
- Binary conditioning (-1 or 1) for wave orientation

Dependencies:
    torch: Neural network and tensor operations
    torch.nn: Neural network modules
    matplotlib.pyplot: Visualization
    numpy: Numerical operations

Main Components:

1. ConditionedDenoiseModel:
   - Input: 101 dimensions (100 points + 1 condition)
   - Hidden layers: 3x256 units with ReLU
   - Output: 100 dimensions
   - Conditioning: Concatenation-based

2. Diffusion:
   - 1000 diffusion steps
   - Linear noise schedule (beta: 1e-5 to 0.01)
   - Forward process: add_noise()
   - Reverse process: sample()

3. Training:
   - 2000 epochs
   - Batch size: 64
   - Learning rate: 1e-5
   - Adam optimizer
   - MSE loss
   - Gradient clipping at 1.0
   - 4 training steps per batch

4. Data Generation:
   - 2000 samples
   - Pure sine waves
   - Binary conditions (-1, 1)
   - Target: regular/inverted waves

5. Visualization:
```

```
    - Three subplots:
      a. Input sine wave
      b. Generated waves (same noise, different conditions)
      c. Target waves
    - Grid and legend for clarity
    - Fixed y-axis limits (-1.2 to 1.2)

Usage:
    Run the script directly to train the model and visualize results:
    $ python new_code_learn_conditioning_0003.py

Output:
    - Training progress (loss every 100 epochs)
    - Three-panel visualization of results


"""
#
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
import numpy as np

class ConditionedDenoiseModel(nn.Module):
    """
    Neural network model that learns to denoise data conditioned on a binary
input.
    Enhanced architecture with increased capacity for better learning.
    """
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(101, 256),  # 100 points + 1 condition
            nn.ReLU(),
            nn.Linear(256, 256),
            nn.ReLU(),
            nn.Linear(256, 256),
            nn.ReLU(),
            nn.Linear(256, 100)   # Output: 100 points
        )

    def forward(self, x, condition):
        x_input = torch.cat([x, condition], dim=1)
        return self.net(x_input)

class Diffusion:
    """
    Implements the diffusion process for adding and removing noise from data.
    Modified for more stable diffusion process.
    """
    def __init__(self, n_steps=1000):
        self.n_steps = n_steps
        self.betas = torch.linspace(1e-5, 0.01, n_steps)
```

```
        self.alphas = 1 - self.betas
        self.alphas_cumprod = torch.cumprod(self.alphas, dim=0)

    def add_noise(self, x, t):
        noise = torch.randn_like(x)
        alpha_t = self.alphas_cumprod[t].view(-1, 1)
        noisy_x = torch.sqrt(alpha_t) * x + torch.sqrt(1 - alpha_t) * noise
        return noisy_x, noise

    def sample(self, model, condition, n_steps=None):
        if n_steps is None:
            n_steps = self.n_steps

        x = torch.randn(1, 100)  # Start from noise

        for t in reversed(range(n_steps)):
            pred_noise = model(x, condition.view(1, 1))
            alpha_t = self.alphas[t]
            alpha_t_cumprod = self.alphas_cumprod[t]

            if t > 0:
                noise = torch.randn_like(x)
            else:
                noise = 0

            x = (1 / torch.sqrt(alpha_t)) * (
                x - (1 - alpha_t) / torch.sqrt(1 - alpha_t_cumprod) *
pred_noise
            ) + torch.sqrt(self.betas[t]) * noise

        return x

def generate_training_data(n_samples=2000):
    """
    Generate training data using pure input approach.
    Returns separate input data, conditions, and targets.
    """
    x = np.linspace(0, 10, 100)
    base_sine = torch.FloatTensor(np.sin(x))

    # Input data: always regular sine waves
    input_data = base_sine.repeat(n_samples, 1)

    # Conditions: -1 or 1
    conditions = (torch.randint(0, 2, (n_samples, 1)) * 2 - 1).float()

    # Target data: regular or inverted sine waves based on conditions

    target_data = base_sine.repeat(n_samples, 1) * conditions
    return input_data, conditions, target_data

def train_model(n_epochs=2000, batch_size=64):
    """
```

```
    Train the diffusion model with improved stability measures.
    """
    model = ConditionedDenoiseModel()
    diffusion = Diffusion()
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-5)

    input_data, conditions, target_data = generate_training_data()

    for epoch in range(n_epochs):
        total_loss = 0
        batches = 0

        for i in range(0, len(input_data), batch_size):
            input_batch = input_data[i:i+batch_size]
            batch_conditions = conditions[i:i+batch_size]
            target_batch = target_data[i:i+batch_size]

            # Multiple training steps per batch
            for _ in range(4):
                t = torch.randint(0, diffusion.n_steps,
(input_batch.shape[0],))
                noisy_batch, noise = diffusion.add_noise(target_batch, t)
                pred_noise = model(noisy_batch, batch_conditions)

                loss = nn.MSELoss()(pred_noise, noise)
                total_loss += loss.item()
                batches += 1

                optimizer.zero_grad()
                loss.backward()
                torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)  #
Added gradient clipping
                optimizer.step()

        if (epoch + 1) % 100 == 0:
            avg_loss = total_loss / batches
            print(f'Epoch {epoch+1}, Average Loss: {avg_loss:.6f}')

    return model, diffusion

def visualize_results(model, diffusion):
    """
    Visualize the results of the diffusion model.
    """
    model.eval()
    with torch.no_grad():
        plt.figure(figsize=(15, 5))

        # Show the input sine wave
        plt.subplot(131)
        x = np.linspace(0, 10, 100)
        plt.plot(np.sin(x))
        plt.title('Input Sine Wave')
```

```
        plt.grid(True)
        plt.ylim(-1.2, 1.2)

        # Generate with both conditions from the SAME noise
        plt.subplot(132)
        condition_regular = torch.tensor([1.0])
        condition_inverted = torch.tensor([-1.0])

        # Use same noise for both generations
        torch.manual_seed(42)  # For reproducibility
        noise = torch.randn(1, 100)

        torch.manual_seed(42)  # Reset seed to use same noise
        generated_regular = diffusion.sample(model, condition_regular)
        torch.manual_seed(42)  # Reset seed to use same noise
        generated_inverted = diffusion.sample(model, condition_inverted)

        plt.plot(generated_regular[0].numpy(), label='Regular
(condition=1)')
        plt.plot(generated_inverted[0].numpy(), label='Inverted
(condition=-1)')
        plt.title('Generated Waves\nfrom Same Noise')
        plt.legend()
        plt.grid(True)
        plt.ylim(-1.2, 1.2)

        # Show target waves
        plt.subplot(133)
        plt.plot(np.sin(x), label='Target Regular')
        plt.plot(-np.sin(x), label='Target Inverted')
        plt.title('Target Waves')
        plt.legend()
        plt.grid(True)
        plt.ylim(-1.2, 1.2)

        plt.tight_layout()
        plt.show()

def main():
    """
    Main function to train the model and visualize results.
    """
    print("Training model...")
    model, diffusion = train_model()

    print("\nGenerating visualizations...")
    visualize_results(model, diffusion)

if __name__ == "__main__":
    main()
```

readme file in the local PC

/x/finance-2024/AI/healthscience/AlphaFold/diffusion
$ cat readme.txt
These python programs have been run locally
document:
https://docs.google.com/document/d/1RanW4yFKj7xxd7ULiZqZ7YudJ8Y7mWbbUlNkF7ttoD8/edit?usp=sharing

2025 document and code:
https://docs.google.com/document/d/1F2T6ldnV_5Fn9jxTGcOqKyLZCDAfv403nlafwOcr_CM/edit?usp=sharing

https://docs.google.com/document/d/1d4EmjJ0d-jlzA7u5pJSucBmIHQVmo9qtPqGQ8y5sISM/edit?usp=sharing [1]

Description

(0) Source

I used Anthropic.

(i) Purpose and Scope

This is a minimal educational implementation of a conditional diffusion model that demonstrates core concepts using sine waves. It's designed to be simple and interpretable, working with 1D data (sine waves) rather than complex structures like proteins or images.

(ii) Relation to AlphaFold 3

While this is vastly simplified, it illustrates a key concept used in AlphaFold 3: conditional generation. In AlphaFold:

- Conditions: protein sequences, multiple sequence alignments, templates
- Output: 3D protein structures
  Here:
- Condition: simple binary flag (1 or -1)
- Output: regular or inverted sine wave

(iii) Conditioning Mechanism

[1] This document

Conditioning appears in three key places:

```
# 1. Data Generation
target_data = base_sine.repeat(n_samples, 1) * conditions

# 2. Model Architecture
class ConditionedDenoiseModel(nn.Module):
    def forward(self, x, condition):
        x_input = torch.cat([x, condition], dim=1)

# 3. Generation/Sampling
condition_regular = torch.tensor([1.0])
condition_inverted = torch.tensor([-1.0])
```

(iv) Time Steps Implementation

The diffusion process uses 1000 time steps:

```
class Diffusion:
    def __init__(self, n_steps=1000):
        self.betas = torch.linspace(1e-5, 0.01, n_steps)  # Noise schedule
        self.alphas = 1 - self.betas
        self.alphas_cumprod = torch.cumprod(self.alphas, dim=0)
```

- Forward process: Gradually adds noise to clean data
- Reverse process: Iteratively denoises from t=999 to t=0

(v) Code Details and Dependencies

Libraries used:

- PyTorch (`torch`): Main deep learning framework
- NumPy (`np`): Numerical computations
- Matplotlib (`plt`): Visualization

Key Components:

1. **Neural Network** (`ConditionedDenoiseModel`):
    - Simple 3-layer MLP (101→256→256→256 -> 100)
    - ReLU activations
    - Takes noisy data + condition as input

2. **Diffusion Process** (`Diffusion` class):
    - Implements forward noising: `add_noise()`
    - Implements reverse denoising: `sample()`
    - Uses linear noise schedule
3. **Training Loop**:
    - Adam optimizer with learning rate 1e-5
    - MSE loss for noise prediction
    - Batch size of 64
    - 2000 epochs
    - Multiple training steps per batch
    - Gradient clipping for stability
4. **Data Generation**:
    - Creates sine waves using `np.sin()`
    - Converts to PyTorch tensors
    - Applies conditioning by multiplication
5. **Visualization**:
    - Three subplots showing:
        - Starting noise
        - Generated outputs
        - Target waves
    - Uses matplotlib for plotting

The code demonstrates how a simple conditional diffusion model can learn to generate different outputs (regular vs inverted sine waves) from the same starting noise, based on a conditioning signal.

(vi) How Components interact

The program starts at `main()`, which is a standalone function serving as the entry point. Its primary job is to call `train_model()` and then pass the results to `visualize_results()`.

`train_model()` is the core orchestrator function. It's not part of any class but instead creates and manages instances of two important classes:

1. It creates a `ConditionedDenoiseModel` instance, which is your neural network. This model:

    - Has a structure of input → hidden layers → output
    - Can process both the signal and a condition value
    - Will learn to predict noise in the signal
2. It creates a `Diffusion` instance, which handles all the noise-related operations:

    - Knows how to add noise gradually
    - Manages the denoising schedule
    - Handles the mathematics of noise levels at different timesteps

The training process then proceeds with `train_model()` coordinating between these instances:

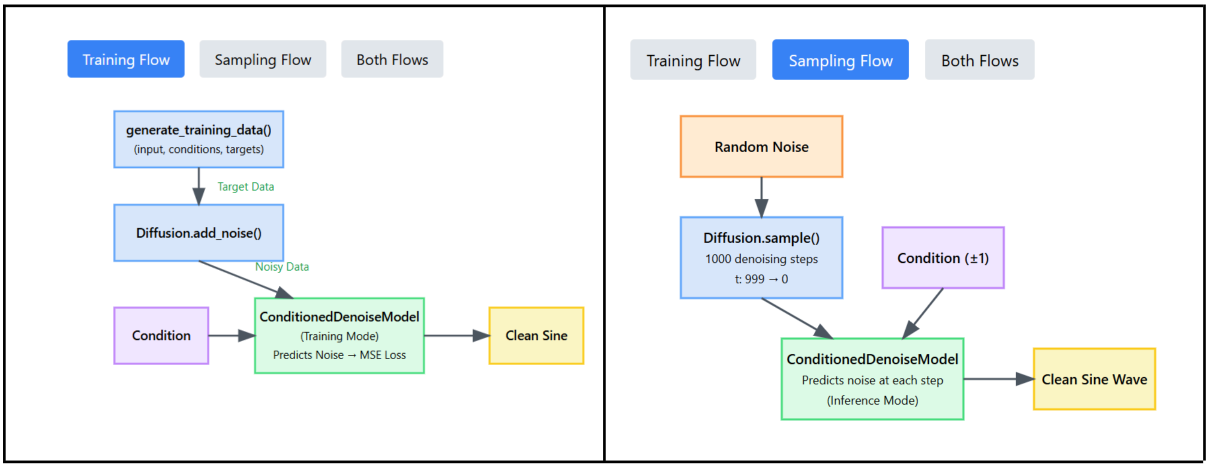
- It gets training data (sine waves and conditions)
- For each training step:
    - Asks the `Diffusion` instance to add noise to the data
    - Feeds the noisy data through the `ConditionedDenoiseModel` instance
    - Updates the model based on how well it predicted the noise

After training, both the trained model and diffusion process handler are returned to `main()`, which then uses them to demonstrate the results through visualization.

Deploying in github

TBD

Appendix - Code details -



(i) Training Mode:

1. Data Flow:

- **Function `generate_training_data`**`(n_samples=2000)`:
    - Input data: `input_data = base_sine.repeat(n_samples, 1)`
    - Conditions: `conditions = (torch.randint(0, 2, (n_samples, 1)) * 2 - 1).float()`
    - Target data: `target_data = base_sine.repeat(n_samples, 1) * conditions`

2. For each training batch in `train_model(n_epochs=2000, batch_size=64)`:

- **Class `Diffusion.add_noise`**`(self, x, t)`:
    - Takes `target_batch` and random timestep `t = torch.randint(0, diffusion.n_steps, (input_batch.shape[0],))`
    - Returns `noisy_x, noise` using `alpha_t = self.alphas_cumprod[t].view(-1, 1)`
    - Computes: `noisy_x = torch.sqrt(alpha_t) * x + torch.sqrt(1 - alpha_t) * noise`
    - `Each batch undergoes 4 training iterations with different noise levels`

3. **Class `ConditionedDenoiseModel`**`(nn.Module)`:

- Method `forward(self, x, condition)`:
    - Input: `x_input = torch.cat([x, condition], dim=1)`
    - Network: `self.net = nn.Sequential(...)` with architecture 101→256→256→256→100
- Training in `train_model()`:
    - Predicts noise: `pred_noise = model(noisy_batch, batch_conditions)`
    - Loss: `loss = nn.MSELoss()(pred_noise, noise)`
    - Optimization: `torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)`

(ii) Sampling/Inference Mode:

Initial State:

- **`Random Noise Generation:`**
    - `x = torch.randn(1, 100) (Starting point for sampling)`
- **`Condition Setting:`**
    - `Regular wave: condition_regular = torch.tensor([1.0])`
    - `Inverted wave: condition_inverted = torch.tensor([-1.0])`

Denoising Process via **Diffusion.sample():**

- `Iterative Loop over 1000 steps`

```
x = (1 / torch.sqrt(alpha_t)) * (

    x - (1 - alpha_t) / torch.sqrt(1 - alpha_t_cumprod) * pred_noise

) + torch.sqrt(self.betas[t]) * noise
```

Results Visualization in **`visualize_results()`:**

- Demonstrates model's ability to:
    - Generate regular sine wave (condition = 1.0)
    - Generate inverted sine wave (condition = -1.0)
- Uses same initial noise for both generations
- Shows progression from noise to clean signal

(iii) Relation to Alpha Fold 3

While this is vastly simplified, it illustrates a key concept used in AlphaFold 3: conditional generation. In AlphaFold:
● Conditions: protein sequences, multiple sequence alignments, templates
● Output: 3D protein structures

Here:
● Condition: simple binary flag (1 or -1)
● Output: regular or inverted sine wave

Conditioning Comparison:
In this demo, conditioning is implemented through a simple binary flag concatenated with the input, which determines whether to generate a regular or inverted sine wave. This mirrors, in a simplified way, how AF3 uses amino acid representations to condition its diffusion process:

- Demo conditioning:
  * Single scalar value (-1 or 1)
  * Direct concatenation with input
  * Binary outcome (regular/inverted wave)

- AF3 conditioning:
  * Single representation: each amino acid's chemical and physical properties
  * Pair representation: relationships between amino acid pairs
  * Complex embedding of sequence information into the diffusion process
  * Multiple conditions interact to guide the protein folding trajectory

The key parallel is how both systems use conditioning to guide the denoising process toward specific outputs, though AF3 does this at a much more sophisticated level with multiple interacting conditions that capture the complex requirements of protein structure.