Study Material - Youtube

Document Information

- **Generated:** 2025-08-26 07:24:49
- **Source:** https://www.youtube.com/watch?v=BKqUPihkNmY
- **Platform:** Youtube
- **Word Count:** 2,257 words
- **Estimated Reading Time:** ~11 minutes
- **Number of Chapters:** 24
- **Transcript Available:** Yes (analyzed from video content)

Table of Contents

Video Overview

This video lecture provides a comprehensive tutorial on **Classifier-Guided Diffusion Models**. It moves beyond standard unconditional Denoising Diffusion Probabilistic Models (DDPMs) to demonstrate how to steer the generation process to create images of a specific, desired class. The instructor first recaps the mathematical foundation of the reverse process in DDPMs and then introduces the core components and code for implementing classifier guidance. The tutorial is highly practical, featuring a complete code walkthrough in PyTorch, from training the necessary models to performing guided sampling to generate specific digits from the MNIST dataset.

Learning Objectives

Upon completing this study material, you will be able to:

- Understand the motivation and high-level concept of classifier-guided diffusion.
- Identify the two key models required: an unconditional DDPM and a noise-aware classifier.
- Comprehend the architectural requirements of a noise-aware classifier, specifically its need to process both a noisy image and its corresponding timestep.
- Follow the two-stage training process: first training the unconditional DDPM, then training the noise-aware classifier on noisy data.
- Grasp the guided sampling (inference) algorithm, which combines the unconditional prediction from the DDPM with the gradient from the classifier.
- Analyze the PyTorch code that implements the entire pipeline, from model definition to final image generation.

Prerequisites

To fully understand the content of this video, you should have a solid understanding of:

- **Denoising Diffusion Probabilistic Models (DDPMs):** Familiarity with the forward (noising) and reverse (denoising) processes.
- **Noise Prediction in DDPMs:** Knowledge of how a model (typically a U-Net) is trained to predict the noise added to an image at a given timestep.
- **Neural Networks:** Concepts of Convolutional Neural Networks (CNNs) and Multi-Layer Perceptrons (MLPs).
- **PyTorch:** Intermediate proficiency in PyTorch, including defining `nn.Module`, writing training loops, and understanding automatic differentiation (`autograd`).

Key Concepts Covered

- Classifier-Guided Sampling
- Noise-Aware Classifier
- Unconditional DDPM Training
- Time Embedding in Neural Networks
- Classifier Gradient Computation
- Guided Mean Calculation
- Reverse Process Sampling with Guidance

Classifier-Guided Diffusion: A Deep Dive

The central theme of this lecture is to control the output of a diffusion model. A standard DDPM is an unconditional generator; it learns the overall data distribution and generates random samples from it. Classifier guidance is a technique that introduces control, allowing us to specify *what* we want to generate (e.g., “a cat,” “the digit 9”).

Intuitive Foundation

Imagine you have a skilled but blindfolded sculptor (the DDPM) who is excellent at carving random shapes from a block of marble (noise). Now, you introduce a guide (the classifier) who can see the sculpture and knows what a “cat” should look like. At each step of the sculpting process, the guide looks at the current shape and whispers directions to the sculptor: “a bit more curve here for the tail,” “make this part pointier for the ears.”

In our case: * The **DDPM** is the “sculptor,” performing the reverse diffusion process to turn noise into an image. * The **Noise-Aware Classifier** is the “guide,” examining the noisy intermediate image `x_t` at each step. * The **“whispered directions”** are the gradients from the classifier. The classifier tells the DDPM how to slightly alter the image `x_t` to make it look more like the target class.

This process is visualized in the diagram below, which outlines the two-stage training and final guided inference.

```
flowchart TD
    subgraph Training Phase
        A["Train Unconditional DDPM<br/>(Noise Predictor)"] --> B["Freeze DDPM"];
        C["Train Noise-Aware Classifier<br/>On noisy images (x_t) and timesteps (t)"] --> D["Freeze Cla
    end

    subgraph Inference Phase (Guided Sampling)
        E["Start with pure noise (x_T)"] --> F{"Loop t = T-1 down to 0"};
        F --> G["1. Get unconditional mean ( _uncond)<br/>from DDPM"];
        F --> H["2. Get classifier gradient (  log p(y|x_t))<br/>from Classifier"];
        G & H --> I["3. Combine to get guided mean<br/> _guided =  _uncond + nudge"];
        I --> J["4. Sample next image (x_{t-1})<br/>using  _guided"];
        J --> F;
        F -->|t=0| K["Final Image"];
    end

    B --> G;
    D --> H;
```

Figure 1: A flowchart illustrating the complete process of training and inference for classifier-guided diffusion as explained in the lecture.

Component 1: The Noise-Aware Classifier

A standard image classifier is trained on clean images. However, during the reverse diffusion process, the model operates on noisy intermediate images `x_t`. A standard classifier would perform poorly on such inputs. Therefore, we need a **noise-aware classifier**.

As explained at **01:34**, this classifier has a crucial modification: it must be trained on noisy images and must take the **timestep `t`** as an additional input. This allows the classifier to understand the level of noise in the image and make a more informed prediction.

Architecture and Implementation

The instructor presents the `NoisyClassifier` class in PyTorch at **01:34**.

```python
# As shown at 01:34
class NoisyClassifier(nn.Module):
    def __init__(self, num_classes=10):
        super().__init__()
        self.time_emb_dim = 32

        # Time embedding, similar to the U-Net
        self.time_mlp = nn.Sequential(
            SinusoidalPositionEmbeddings(self.time_emb_dim),
            nn.Linear(self.time_emb_dim, self.time_emb_dim),
            nn.ReLU()
        )

        # Standard CNN architecture
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
        # The time embedding will be added after this layer
        self.fc1 = nn.Linear(32 * 8 * 8, 256) # Assuming 32x32 input, pooled twice
        self.fc2 = nn.Linear(256, num_classes)
        self.relu = nn.ReLU()

    def forward(self, x, time):
        # Embed the timestep
        t_emb = self.time_mlp(time)

        # Pass through CNN layers
        x = self.relu(self.pool(self.conv1(x)))
        x = self.relu(self.pool(self.conv2(x)))

        # Add time embedding to the features
        # We need to process the time embedding to match the feature dimensions
        t_emb_transformed = F.interpolate(t_emb.unsqueeze(-1).unsqueeze(-1), size=x.shape[2:], mode='nea
        x = x + t_emb_transformed

        # Flatten and pass through fully connected layers
        x = x.view(-1, 32 * 8 * 8)
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Key Implementation Details:

1. **Time Embedding (02:00):** The timestep `t` is first converted into a high-dimensional vector using `SinusoidalPositionEmbeddings`, just like in the U-Net model. This vector is then passed through a small MLP.
2. **Feature Extraction:** The input image `x` is passed through a standard sequence of convolutional and pooling layers.
3. **Adding Time Information (02:28):** The processed time embedding `t_emb` is reshaped and interpolated to match the spatial dimensions of the image features from the CNN. It is then added to these features. This step makes the classifier "aware" of the noise level.
4. **Classification Head:** The combined features are flattened and passed through fully connected layers to produce the final class logits.

Training the Classifier

The training process for the `NoisyClassifier` is shown at **03:34**. It involves the following steps for each item in a batch: 1. Start with a clean image `x_0` and its true label `y`. 2. Sample a random timestep `t` from `[1, T]`. 3. Generate a noisy image `x_t` by applying the forward diffusion process to `x_0` for `t` steps. 4. Feed both the noisy image `x_t` and the timestep `t` into the `NoisyClassifier`. 5. Calculate the `CrossEntropyLoss` between the classifier's output logits and the true label `y`. 6. Perform backpropagation to update the classifier's weights.

> **Important Note:** This training happens *after* the unconditional DDPM has already been trained and its weights are frozen. The two models are trained sequentially.

The Guided Sampling Algorithm

Once both the DDPM and the noise-aware classifier are trained, we can perform guided sampling to generate an image of a specific `target_class`. The process, detailed in the `guided_sample` function at **04:53**, modifies the mean of the reverse sampling distribution at each step.

Mathematical Intuition

The reverse process samples x_{t-1} from a distribution whose mean $\mu_\theta(x_t, t)$ is predicted by the DDPM. This is the **unconditional mean**.

To guide the process, we want to shift this mean in a direction that increases the likelihood of the target class `y`. The direction of steepest ascent for the log-likelihood $\log p_\phi(y|x_t)$ is given by its gradient with respect to the input, $\nabla_{x_t} \log p_\phi(y|x_t)$. This is the **classifier gradient**.

The new, **guided mean** is a combination of the original unconditional mean and this gradient-based nudge:

$$\mu_{guided}(x_t, y) = \mu_{unconditional}(x_t) + \text{nudge}$$

The "nudge" is the classifier gradient, scaled by the posterior variance Σ_t and a `guidance_scale` parameter `s` that controls the strength of the guidance:

$$\text{nudge} = s \cdot \Sigma_t \cdot \nabla_{x_t} \log p_\phi(y|x_t)$$

The final sampling step becomes:

$$x_{t-1} \sim \mathcal{N}(x_{t-1}; \mu_{guided}(x_t, y), \Sigma_t)$$

Code Implementation of Guided Sampling

The instructor walks through the implementation starting at **05:48**.

Step 1: Unconditional Prediction (06:33) First, the unconditional DDPM predicts the noise. From this, the unconditional mean is computed using the standard reverse process formula. This is done within a `torch.no_grad()` context because we don't want to affect the DDPM.

```
# Predict noise unconditionally
with torch.no_grad():
    predicted_noise = ddpm_model(img, t)

# Compute the unconditional reverse mean
unconditional_mean = sqrt_recip_alphas_t * (img - betas_t * predicted_noise / sqrt_one_minus_alphas_cump
```

Step 2: Compute Classifier Gradient (08:10) This is the core of the guidance mechanism.

```
# Enable gradients on the image
with torch.enable_grad():
    img_with_grad = img.detach().requires_grad_(True)

    # Get classifier logits
    logits = classifier_model(img_with_grad, t)
    log_probs = F.log_softmax(logits, dim=-1)

    # Select the log probability of the target class
    selected_log_probs = log_probs.gather(dim=-1, index=y.view(-1, 1))

    # Compute the gradient of the log probability w.r.t. the image
    classifier_grad = torch.autograd.grad(selected_log_probs.sum(), img_with_grad)[0]
```

Step 3: Combine Means with Guidance Nudge (08:55) The unconditional mean and the classifier gradient are combined.

```
# Retrieve posterior variance for this timestep
posterior_variance_t = get_index_from_list(posterior_variance, t, img.shape)

# Calculate the guidance nudge
guidance_nudge = guidance_scale * posterior_variance_t * classifier_grad

# Calculate the final guided mean
guided_mean = unconditional_mean + guidance_nudge
```

Step 4: Sample the Next Image (09:35) A new, slightly less noisy image `x_{t-1}` is sampled using the guided mean.

```
# Inject Gaussian noise for the sampling step
noise = torch.randn_like(img)
# If at the final step (t=0), no additional noise is added
img = guided_mean + torch.sqrt(posterior_variance_t) * noise
```

This loop repeats until `t=0`, at which point `img` is the final generated image.

Visual Elements from the Video

- **Mathematical Equations (00:11, 06:43):** The video displays handwritten equations for the mean of the posterior distribution in DDPMs. The key equation for the reverse mean predicted by the model is:

$$\mu_\theta(x_t) = \frac{1}{\sqrt{\alpha_t}} \left(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \hat{\epsilon}_\theta(x_t) \right)$$

  This formula is implemented in the code to calculate the `unconditional_mean`.

- **Code Interface (00:33 onwards):** The majority of the lecture is a screen-share of a VS Code environment running a Jupyter Notebook. This shows the file structure, the Python code for the models and training loops, and the output of the code execution.

- **Generated Images (05:19, 10:06):** The final output of the guided sampling process is displayed as a grid of 16 images. These are grayscale images of the digit '9', demonstrating that the guidance was successful. While clearly resembling the target digit, some images show artifacts or distortions, which the instructor notes is due to a short training duration.

  A screenshot of the generated images for the digit '9' using classifier guidance, as shown at 05:19.

Self-Assessment for This Video

1. **Question:** Why is a "noise-aware" classifier necessary for this technique, and what two inputs does it require?
   - **Answer:** It is necessary because the guidance happens at every step of the reverse process, which involves noisy intermediate images (`x_t`). A standard classifier trained on clean images would fail. It requires the noisy image `x_t` and the corresponding timestep `t` to understand the context and noise level.
2. **Question:** What is the "classifier gradient," and what does it represent intuitively?
   - **Answer:** The classifier gradient is $\nabla_{x_t} \log p_\phi(y|x_t)$. It is the gradient of the log-probability of the target class `y` with respect to the input image `x_t`. Intuitively, it represents the direction in which to modify the pixels of `x_t` to make it most resemble the target class `y`.

3. **Question:** Explain the role of the `guidance_scale` parameter in the `guided_sample` function. What would happen if this value was set to 0? What if it was set to a very large number?
    - **Answer:** The `guidance_scale` controls the strength of the classifier's influence on the generation process.
        - If `guidance_scale = 0`, the guidance nudge becomes zero, and the sampling process becomes purely unconditional, equivalent to a standard DDPM.
        - If `guidance_scale` is very large, the classifier's gradient will dominate the sampling process. This can lead to images that are very clearly of the target class but may sacrifice diversity and realism, as the model is forced too strongly away from the natural data distribution learned by the DDPM.
4. **Application Exercise:** Modify the provided code to generate images of a different digit (e.g., '3'). What is the only line you need to change to achieve this during inference?
    - **Answer:** You would only need to change the `target_digit` variable that is passed to the `guided_sample` function. For example: `target_digit = 3`.

Key Takeaways from This Video

- **Control is Possible:** Diffusion models can be controlled to generate specific content using guidance techniques.
- **Classifier Guidance is a Powerful Technique:** By combining an unconditional DDPM with a noise-aware classifier, we can guide the sampling process towards a desired class.
- **The Key is the Gradient:** The core mechanism involves using the gradient of the classifier's log-likelihood with respect to the input image to "nudge" the reverse process.
- **Sequential Training:** The process requires a two-stage training pipeline: first, train the generator (DDPM), then train the guide (classifier).
- **Guidance is a Trade-off:** The strength of the guidance (`guidance_scale`) is a hyperparameter that balances class-faithfulness with sample diversity and quality.