Project-455

Scarlette Chu

April 2025

Introduction

Model Architecture and Conditioning

My first attempt for implementing conditional PixelCNN++ was introducing Feature-wise Linear Modulation (FiLM) as the conditioning mechanism. FiLM conditions the network on external class labels by applying per-channel transformations to the intermediate feature maps.

I modified the `gated_resnet` module to accept class embeddings, which were passed through a two-layer multilayer perceptron (MLP) to produce scale (γ) and shift (β) parameters. These parameters were applied to the residual block features:

$$\text{FiLM}(x) = \gamma \odot x + \beta$$

where $\odot$ denotes element-wise multiplication.

The MLP takes one-hot encoded class labels as input and outputs two vectors of size equal to the number of feature channels. These FiLM parameters are injected into the network at each residual block, replacing the original skip connections used for conditioning. Doing this allowed each class to learn a unique per-channel transformation at every residual block. I was hoping this would provide a stronger and more flexible conditioning versus additive methods.

Training Setup

In order to see how the image generation was improving on a per class basis, I adjusted the sampling logic in the training script to generate the sample size per class and calculated the FID across all classes. This helped me determine if the model was actually learning relevant features while monitoring wandb.

In my first iteration of FiLM implementation, I changed sampling to every 10 epochs to understand the change in FID at a finer level. At epoch 0 I got high FID scores around 150 which only kept increasing for the next 30 epochs. This suggested that the generated samples were getting further from the test data distributions. In order to determine the root cause, I sent debug print statements for the gamma mean and standard deviations. I found the gamma mean to be close to zero with values oscillating +/-0.05 from zero and a standard deviation close to 0.5. Having an oscillating gamma value centered around 0 was resulting in activations getting suppressed, inverted or unevenly amplified. This resulted in the model learning noisy and unstable representations resulting in bad samples and a high FID. After investigating this further I found that prior work (Perez et al., 2018; Karras et al., 2019) emphasizes initializing γ near 1 and β near 0 to preserve the signal flow early in training. I realized that my FiLM MLP had no such initialization strategy, which likely contributed to the instability. I adjusted the initialization in my FiLM class to produce neutral modulation parameters at the start of training to avoid extreme modulations at the beginning of training. I then monitored these parameters throughout training to ensure β increase throughout training

Results

To understand how well the FiLM-conditioned PixelCNN++ model performed, we looked at FID scores, Bits-Per-Dimension (BPD), and sample quality at different stages of training.

FID Score

I evaluated FID at epochs 0, 25, 50, 75, and 100 using 4 generated image per class. As shown in Figure 5, the FID score dropped significantly early on—from 38.2 at epoch 0 to a low of 30.2 at epoch 25—indicating that the model was quickly learning to produce more realistic samples. However, by epoch 50, the FID spiked back up to 37.5, suggesting some instability or confusion in the model's conditioning. After that, FID gradually recovered, landing around 31.4 by epoch 100. Overall, FID improved over time, but the mid-training spike shows that sample quality wasn't improving in a straight line. The final FID over the 100 generated images was 26.67.

Bits-Per-Dimension (BPD)

The training and validation BPD curves (Figures 3 and 4) steadily decreased throughout training. BPD started above 6.0 and dropped below 3.2, showing that the model got better at modeling the data distribution. The fact that BPD kept improving even when FID got worse suggests that the model was still learning, but those improvements didn't immediately translate to better-looking images. This gap between likelihood and perceptual quality seems to be pretty common in generative models.

Qualitative Evaluation

Figure 2 shows early samples from class 3, generated after 50 epochs. These images had some structure, but also showed strange patches of bright, colorful pixels—likely a sign of unstable FiLM conditioning. Interestingly, those glitches disappeared by epoch 75, and the samples became more coherent. This suggests that even though the conditioning started out noisy, the model eventually learned to use it more effectively. In the future, this kind of instability could probably be reduced by initializing the FiLM MLP more carefully, clamping the γ values, or using learning rate warmup in the early stages of training.

Classification Accuracy

Unfortunately, I ran out of time and wasn't able to get `classification_evaluation.py` working before the deadline. While I did focus on building and testing the generative side of the model, classification results are still pending and could be evaluated in future work.

(To be completed after running `classification_evaluation.py`)

Challenges Analysis

In my very first attempt in understanding the model archetecture I tried implementing a label embedding only style where at each residual black I passed a label embedding, this mirrored what was done in the original Conditional PixelCNN++ paper. After debugging this approach and finally getting training to run, yet FID consistenyl failed to compute.

At first, the training script would crash entirely. To keep training from breaking, I wrapped the FID calculation in a try/except block — which stopped the crash but revealed another issue: no sample images were being generated or logged to Weights Biases.

I experimented with multiple changes:

- I restructured the conditioning logic.
- I switched operating systems from Windows to Linux.

- I set up a fresh environment with Conda and PyTorch.

None of these resolved the problem. Every setup resulted in the same FID failure.

At one point, I manually generated images and ran a local FID script. It returned an error due to an *imaginary component* in the distance calculation, which led me to research how FID behaves when the generated image distribution has **too little variance**. This was confusing, since even with *random weights at epoch 0*, others didn't encounter the same issue.

Eventually, I ran the code on **Google Colab**, and it worked. FID computed successfully, and sample images were generated as expected. But Colab introduced new problems: frequent disconnections, browser crashes, and unpredictable freezing on my local machine. I tried connecting Colab to my local GPU, but FID broke again with the same error.

Final Workflow

In the end, I settled on this workflow:

1. Run training on my **local GPU in Ubuntu** for stability.
2. Save sample images after each epoch.
3. **Export those images to Colab** to manually run FID evaluation.

It was not ideal, but it was reliable. This ultimately absorbed a lot of the time that was spent on this project, taking away my ability to adjust hyperparameters and experiment with different approaches.

Learnable Class Embedding

I wanted to use the one-hot vector method as a first attempt because it was easier to implement and easier to debug during training. I also wanted to adjust hyperparameters to measure the effect of the models' performance before experimenting with another approach.

Before encountering the FID issues, I did research on using a learnable embedding layer to represent class labels. In the paper "Conditional Image Generation with PixelCNN Decoderes" (Van Den Oord et al. 2016), the authors experimented conditioning PixelCNN models using learned embeddings. Their approach enhanced the model's ability to generate diverse and class-consistent images. I was eager to attempt this approach due to the flexibility in allowing the model to learn semantic relationships between classes. The current one-hot vectors treat every class as equally distant from each other, which is not the case for the classes given in this assignment. The classes we have to condition on are burgers, pizza, pandas, and kolas. The two food classes and two animal classes may have embeddings that are close due their semantic similarities. Implementing a learned class embedding could allow the model to generalize features like fur texture, or shape more effectively.

Bonus Questions

1. Why Do Masked Convolution Layers Ensure the Autoregressive Property of PixelCNN++?

PixelCNN++ models the joint distribution of image pixels x as a product of conditions:

$$p(x) = \prod_{i=1}^{n} p(x_i \mid x_1, \ldots, x_{i-1})$$

In this setup, the value of each pixel x_i should only depend on the pixels that have already been generated, which should be those above and to the left of it.

The model enforces this autoregressive behavior, the model **masked convolutions**. These are convolutional layers where a binary mask is applied to the filter weights to zero out connections to future pixels. For example, when generating the pixel (i, j), the region of the input image that influences this particular output value only includes:

- Pixels from rows $< i$
- Pixels in the same row but with columns $< j$

The above ensures that the prediction for a pixel doesn't "cheat" by looking at information from pixels that haven't been generated yet (looking into the future). Without this masking, the model could access the current pixel value during training, which violates the autoregressive assumption and leads to incorrect training dynamics.

2. What Are the Advantages of Using a Mixture of Logistics in PixelCNN++?

Instead of using softmax across all 256 pixels, PixelCNN++ uses a **mixture of logistics** to model the output distribution. This is specifically chosen as it offers many advantages:

1. **Continuous modeling:** Instead of treating pixel values as discrete categories, a mixture of logistics allows the model to operate over a continuous range $[0, 1]$. This better captures the smoothness in real/natural iamges.
2. **Smaller output size:** For each pixel channel, the model outputs parameters for a small number (e.g., 5 or 10) of logistic components, each with its own mean, scale, and mixing coefficient. This reduces memory and speeds up training.
3. **Better sample quality:** The logistic mixture can represent distributions with more than one peak. This means it can model uncertain regions more effectively. Such areas include those with texture or soft edges, this can lead to more realistic samples.
4. **Improved sampling flexibility:** When generating new pixels, the model can sample directly from the continuous distribution defined by the mixture. This avoids the blocky transitions that sometimes result from categorical sampling and also helps to produce more natural-looking images.

Overall, using a mixture of logistics makes the model sample more natural images and makes the model more computationally efficient.

Appendix

A Additional Figures

Citations

Perez, E., Strub, F., de Vries, H., Dumoulin, V., & Courville, A. (2018). *FiLM: Visual Reasoning with a General Conditioning Layer*. Proceedings of the AAAI Conference on Artificial Intelligence, 32(1). https://arxiv.org/abs/1709.07871

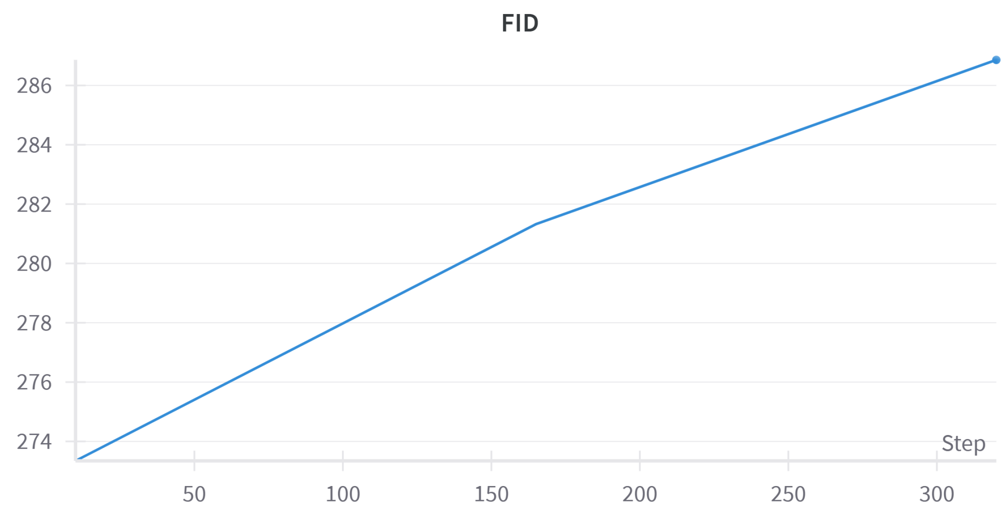

Figure 1: Training run on colab notebook before initializing (γ) and (β) values

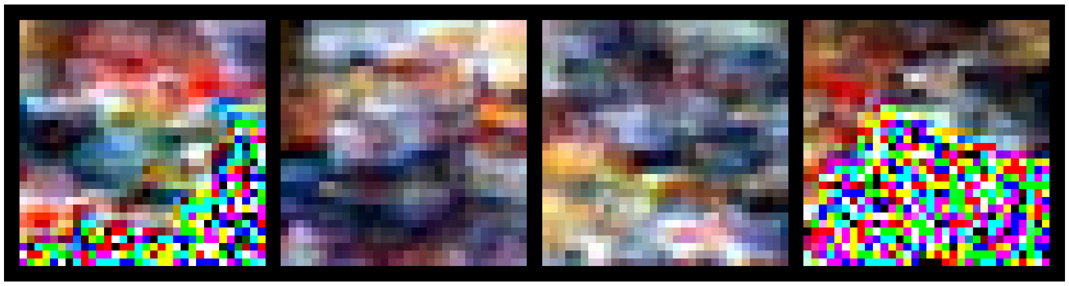

Figure 2: First sampling of class 3 after 50 epoch of training. Sharp injections of colourful pixels can be seen among coherent areas.

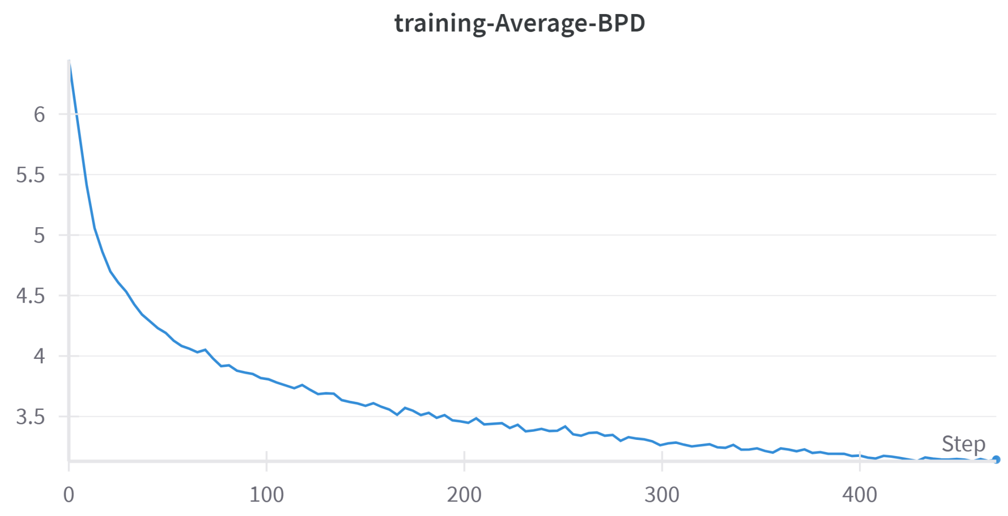

Figure 3: Training Average BPD on FiLM implementation.

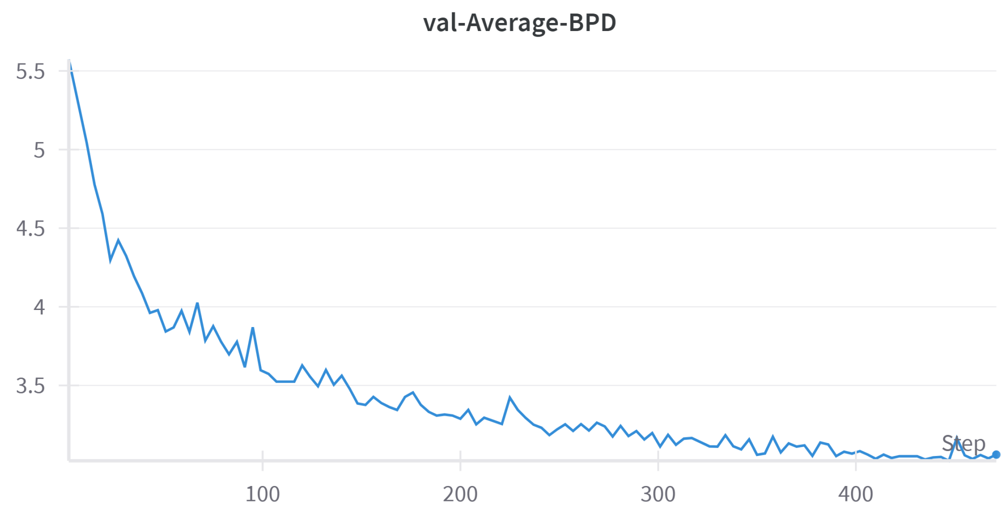

Figure 4: Validation Average BPD on FiLM implementation.

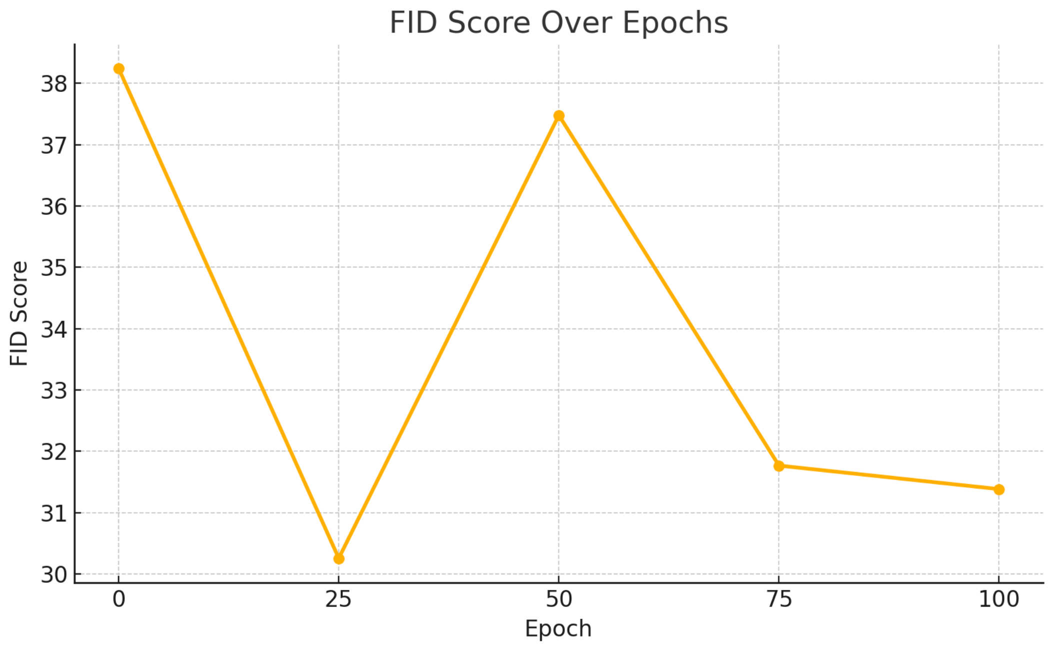

Figure 5: Validation Average BPD on FiLM implementation.

van den Oord, A., Kalchbrenner, N., Espeholt, L., Vinyals, O., Graves, A., & Kavukcuoglu, K. (2016). *Conditional Image Generation with PixelCNN Decoders.* Advances in Neural Information Processing Systems, 29. https://arxiv.org/abs/1606.05328