CPEN455: Classification using PixelCNN++

Jeffrey Chan
CPEN455: Deep Learning
University of British Columbia
34667170

Abstract

This paper presents an implementation and investigation of the PixelCNN++ model for image classification tasks. PixelCNN++ is a generative model that is able to generate images pixel by pixel. More specifically, an unconditional PixelCNN++ is a model that is able to generate images from a dataset without any conditioning information. However, it can be shown that by add conditioning information to the PixelCNN++ model, we can train it to generate images of specific classes. We can exploit this conditional model to perform image classificaion of classes that we trained on.

1 Introduction

The purpose of this project is to understand and implement a conditional PixelCNN++ model for use as a generative model and classifier. Moreover, the goal is to train an accurate classifier for our course competition on a unlabeled dataset of various classes. Some benefits of an unconditional PixelCNN++ model is that is has tractable likelihood. This means that the probability of generating any given image can be computed efficiently. This is in contrast to some other generative models like Generative Adversarial Networks (GANs) which do not have a tractable likelihood. Moreover, the PixelCNN++ model incorporates several modifications to the original PixelCNN architecture that simplify the structure and improve performance. These include using a discretized logistic mixture likelihood, conditioning on whole pixels instead of RGB sub-pixels, and adding shortcut connections.

2 Model

The provided source code implements the PixelCNN++ model which models the joint distribution of pixels over an image x as

$$p_\theta(x) = \prod_{i=1}^{n} p_\theta(x_i|x_{<i}) \tag{1}$$

where x is the image, x_i is the pixel at location i, and $x_{<i}$ is the set of pixels before pixel i. To make our model generate images of specific classes, we can condition the model on the class label. This can be done via

$$p_\theta(x|c) = \prod_{i=1}^{n} p_\theta(x_i|x_{<i}, c) \tag{2}$$

where c is the class label [1].

2.1 Architecture

The key architectural difference between the conditional and the unconditional PixelCNN++ model is shown in Fig. 1.

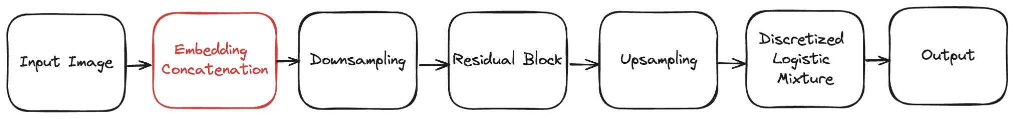


Figure 1: Core PixelCNN++ model architecture change.

The new block is the embedding concatenation block which takes the class label and embeds it into a tensor of weights the same size as the input image. This vector is then element wise added with the input image and passed through the rest of the model. Thus, this allows the model to learn from the class features and generate images of specific classes.

This embedding block can be added in various ways (i.e. concatenation, adding during the downsampling blocks, etc), but it was found through trial and error that the most accurate model was achieved by adding the embeddings after the first set of convolutions.

2.2 Loss function

In typical machine learning problems, we want to minimize the Kullback-Leibler (KL) divergence between the true distribution and the model distribution. This is given by

$$D_{KL}(p_\theta || p_{data}) = \sum_z p_{data}(z) \log \frac{p_{data}(z)}{p_\theta(z)} \tag{3}$$

$$= \sum_z p_{data}(z) \log p_{data}(z) - \sum_z p_{data}(z) \log p_\theta(z) \tag{4}$$

$$\approx -\sum_{i=1}^{N} \log p_\theta(x_i) \quad [4] \tag{5}$$

In Eq. 4, the first term is the self entropy of the data distribution and the second term is the cross entropy between the true distribution and the model distribution. It turns out the self entropy term is constant and can be ignored when minimizing the KL divergence. Thus, we can approximate the KL divergence as shown in Eq. 5 [1]. So to minimize the KL divergence, we can minimize the negative log likelihood of the model distribution. For our PixelCNN++ model, in order to reconstruct the probability of a batch, the discretized logistic mixture likelihood method is used. This method addresses the challenge of modelling the discrete 8-bit pixel values in an efficient manner. In summary, the probability of a sub-pixel to be of value x is defined by

$$p(x|\pi, \mu, s) = \sum_{i=0}^{K} \pi_i [\sigma(\frac{x + 0.5 - \mu_i}{s}) - \sigma(\frac{x - 0.5 - \mu_i}{s})] \tag{6}$$

where σ is the logistic sigmoid function, π_i is the mixture weight, μ_i is the mean, and s_i is the scale. The model is trained by minimizing the negative log likelihood of the model distribution [1]. For more information, please refer to the original paper [3].

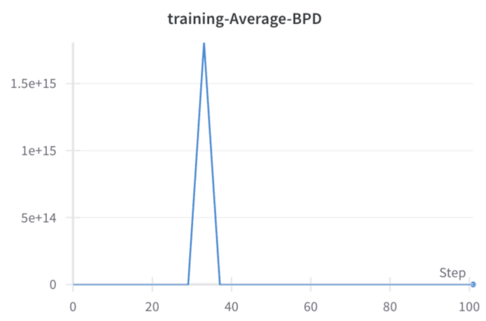
3 Experiments

3.1 Training Hardware

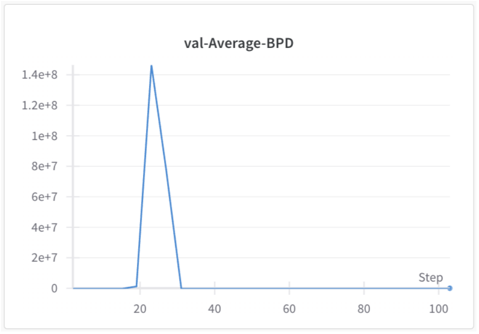
Training was experimented both on a M1 Macbook Pro and Google Colab T4 GPU. When training on the Macbook, numerical errors were encountered as shown in Fig. 2. This caused major issues in the loss computations which resulted in the model not learning. Thus, training was done further solely on Google Colab.

3.2 Data Augmentation

Various image augmentation techniques were performed on the training dataset to increase the generalization of the model. Experiments included using random rotations, flips, color jitter, gaussian



(a) Training Error Plot



(b) Validation Error Plot

Figure 2: Training Plots on the Macbook

noise and random cut outs. Moreover, normalizing the images to their dataset means and standard deviations yielded good results on top of the previous augmentations. The means and std values were computed on the dataset and used to normalize the images.

3.3 Hyperparameters

The hyperparameters used for training the model and their resulting classification accuracy on our test dataset is shown in Table 1.

Table 1: Hyperparameters used in various training runs.

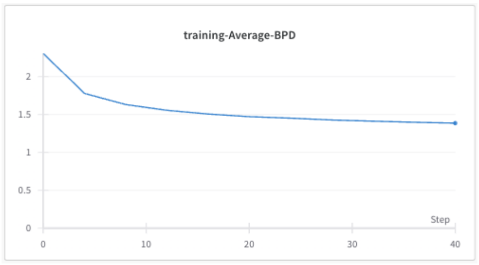
Experiment #	Batch Size	Epoch	# of Filters	# of ResNets	# of Logistic Mix	Accuracy (%)
1	64	60	90	3	5	80.4
2	64	20	30	2	4	49
3	32	40	120	3	6	58.4
4	64	60	40	4	6	77.1
5	64	40	80	3	5	76.2
6	128	60	90	3	5	74.3

It is important to note that these experiments were not conducted with the same embedding blocks and augmentation techniques and thus can't be used to compare directly. It was noticed that concatenating the embedding blocks an appending it to the channels of the image yielded poorer results that adding the the embeddings. To be concise, the concatenation was $B \times C \times H \times W + B \times D \times 1 \times 1 = B \times (C + D) \times H \times W$, vs the addition was $B \times D \times H \times W + B \times D \times 1 \times 1 = B \times D \times H \times W$, where D is the dimension downstream after first set of convolution. One reason for this is because when concatenating the class embedding to the input image chnnel, the network has to learn to extract the relevant class information and combine it with the pixel-level features from scratch. This can be challenging, especially if the relationship between the class labels and pixel values is complex or subtle. On the other hand, when you add the class embedding after the first convolutional pass, the network has already learned some meaningful pixel-level features. Perhaps, the class information can then be more easily integrated with these learned features.

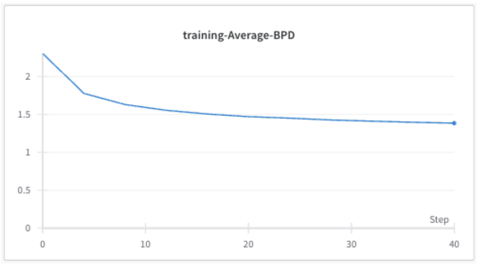
4 Conclusion

4.1 Results

The best model was trained had an accuracy of 80.4% on the test dataset and FID score of 48.7. To perform the classification tasks, we utilized our PixelCNN++ generative model and condition on each of the class labels and take the maximum likelihood class as the predicted class.

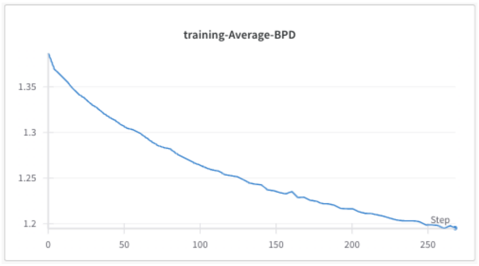


(a) Training Error Plot

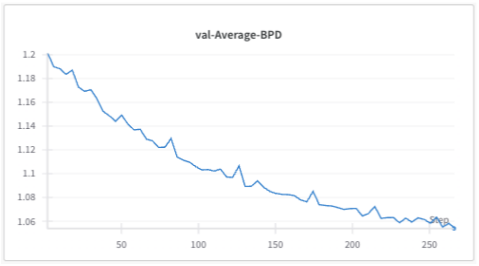


(b) Validation Error Plot

Figure 3: Loss plots for highest accuracy model (LR = 0.0003).



(a) Training Error Plot



(b) Validation Error Plot

Figure 4: More loss plots for highest accuracy model (LR = 0.00015).

4.2 Future Work

In the future, it would be interesting to experiment with different embedding block architectures to increase the parameters affected with the conditioning information. This can come in the form of adding a MLP layer to the class embedding or adding more convolutional layers.

Moreover, future work should include more combinations of hyperparameters and training on a larger dataset. With time and compute constraints, it was difficult to experiment with more hyperparameters and training on a larger dataset which I believe would have helped the model generalize better.

References

[1] Liao, Renjie (2024). PixelCNN Lecture 9 [PDF]. The University of British Columbia. https://lrjconan.github.io/UBC-CPEN455-DL/assets/slides_2024/pixel_cnn.pdf

[2] PyTorch (2023) Transforming and augmenting images. PyTorch Vision Stable Documentation. Retrieved April 21, 2024, from https://pytorch.org/vision/stable/transforms.html.

[3] Salimans, T., Karpathy, A., Chen, X., & Kingma, D.P. (2017) PixelCNN++: A PixelCNN Implementation with Discretized Logistic Mixture Likelihood and Other Modifications. In *ICLR*.

[4] Zhang, Qihang (2024). Conditional PixelCNN++ Tutorial 12 [PDF]. The University of British Columbia.