Perception Prioritized Training of Diffusion Models

marii

```
from fastai.basics import *
from itertools import accumulate
```

```
%load_ext autoreload
%autoreload 2
```

Background

First, a bit of review. Each x_t is dependent on some x_{t-1} in a Markov chain defined below. This we can also define in terms of a_t and x_0, so that we can calculate the amount of noise at each step without the calculation being Markovian(or dependent on the previous step). This essentially gives us a formula that is the original image x_0 plus some noise ϵ

$q(x_t|x_{t-1}) = \mathcal{N}(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t\mathbf{I})$

$\epsilon \sim \mathcal{N}(0, \mathbf{I})$

$a_t := \prod_{s=1}^{t} 1 - \beta_s$

$x_t = \sqrt{\alpha_t}x_0 + \sqrt{1-\alpha_t}\epsilon$

Both α_t and β_t are of interest to us, so we define the code here. We have 1000 values, one value for each time step t.

```
def at(Bt): return torch.cumprod(1-Bt,-1)
Bt=torch.linspace(1e-4,0.02,1000) #schedule used in DDPM paper
at(Bt).shape
```

```
torch.Size([1000])
```

Signal-to-Noise Ratio

Signal-to-noise ratio is very important for this paper. We if we put the below two formulas close together the relationship should be clear, remembers that x_0 is the original image, and ϵ is our noise.

$x_t = \sqrt{\alpha_t} x_0 + \sqrt{1-\alpha_t}\epsilon$

$SNR(t) = \frac{\alpha_t}{1-\alpha_t}$

```
def snr(at): return at/(1-at)
```

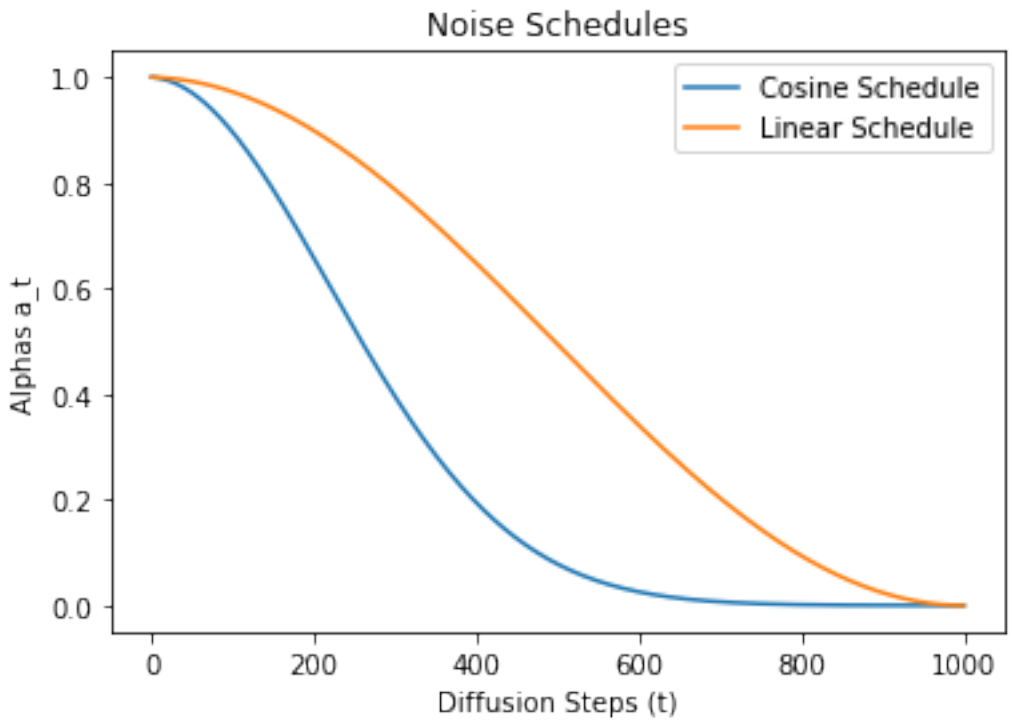
Another noise schedule that is investigated is the cosine noise schedule, we define the code for it below.

```
s=0.008 #from Improved Denoising Diffusion Probabilistic Models
def cos_sched(at):
    at = torch.linspace(0,1,at.shape[0])
    return torch.cos((at+s)/(1+s)*torch.pi/2)**2
```

We can now take a look at the noise schedule per diffusion step t. Remeber low α_t means more noise ϵ.

```
plt.plot(at(Bt))
plt.plot(cos_sched(Bt))
plt.ylabel('Alphas a_t')
plt.xlabel('Diffusion Steps (t)')
plt.title('Noise Schedules')
plt.legend(['Cosine Schedule','Linear Schedule'])
```

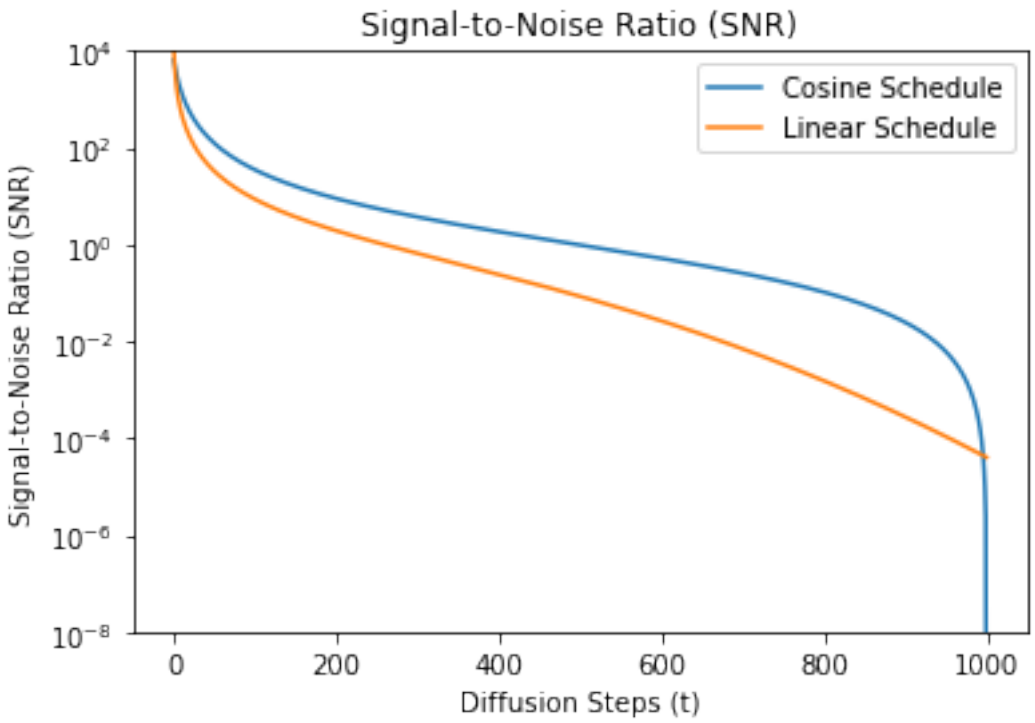
```
<matplotlib.legend.Legend at 0x7fd7e2d69870>
```



Now we can graph the signal to noise ratio for both the cosine and linear schedules.

```
plt.yscale('log')
plt.ylim(top=1e4,bottom=1e-8)
plt.plot(snr(cos_sched(Bt)))
plt.plot(snr(at(Bt)))
plt.ylabel('Signal-to-Noise Ratio (SNR)')
plt.xlabel('Diffusion Steps (t)')
plt.title('Signal-to-Noise Ratio (SNR)')
plt.legend(['Cosine Schedule','Linear Schedule'])
```

```
<matplotlib.legend.Legend at 0x7fd8e86670a0>
```



Notice that we get very similar results to the paper as seen below.

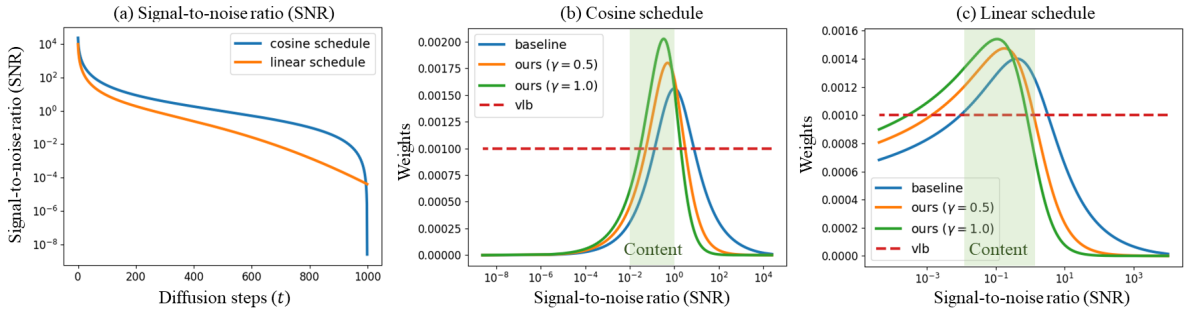


Figure 1: 161203299-8b02d76b-9c51-4529-8329-3ac08e9f3bc8.png

Continuous Weights

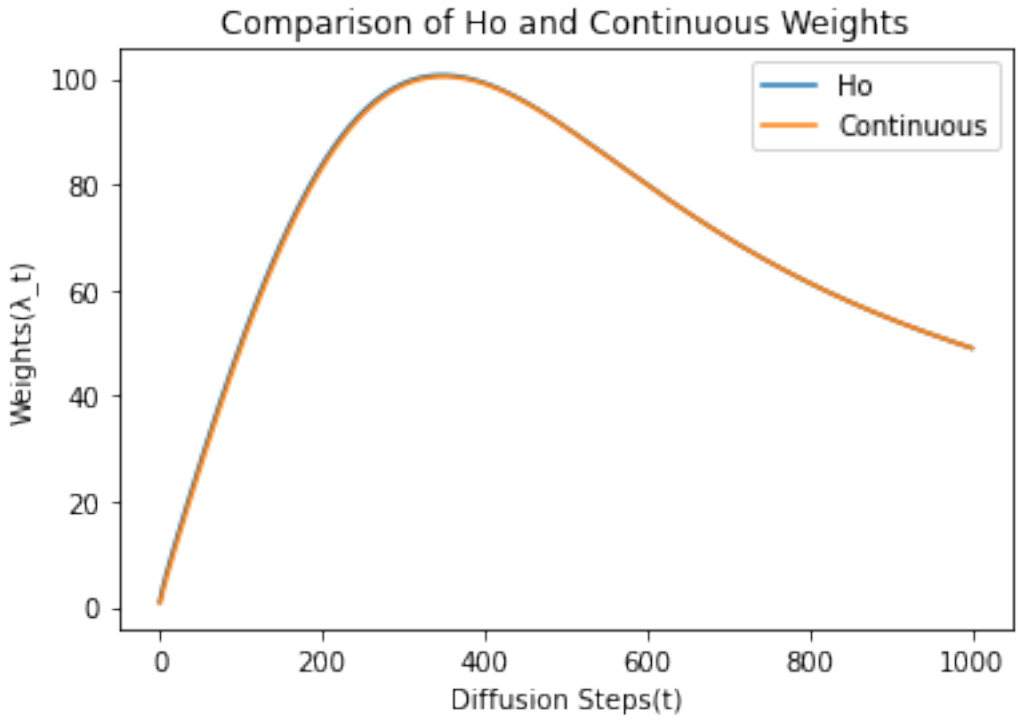
Okay, now we need to determine the weights above. This is the goal of the paper. First we look at the first contribution, a way to calculate the weights in terms of the signal-to-weight ratio, which is a continuous version of the weighting scheme introduced in the DDPM paper. The derivation of this is in the paper's appendix.

```
def Ho_weights(Bt,at):
    return (1-Bt)*(1-at)/Bt
def continuous_weights(at):
    weights = -snr(at[1:])/(snr(at[1:])-snr(at[:-1]))
    return torch.cat((weights[0:1],weights)) #we just make a copy of the first to get same
```

We can now compare the unnormalized weights of the continuous weight schedule and the one in the DDPM paper. They are fairly close.

```
plt.plot(Ho_weights(Bt,at(Bt)))
plt.plot(continuous_weights(at(Bt)))
plt.ylabel('Weights( _t)')
plt.xlabel('Diffusion Steps(t)')
plt.title('Comparison of Ho and Continuous Weights')
plt.legend(['Ho','Continuous'])
```

```
<matplotlib.legend.Legend at 0x7fd7e2e0eb60>
```



Prioritized Weight Schedule

Next we can look at the prioritized weight schedule. The main contribution of the paper. λ_t is our continuous weights from above, k is a constant set to 1. γ is a hyperparameter that we can control, but it doesn't work so well at over 2, because "We empirically observed that over 2 suffers noise artifacts in the sample because it assigns almost zero weight to the clean-up stage" (quoting paper).

$\lambda'_t = \frac{\lambda_t}{(k+SNR(t))^\gamma}$

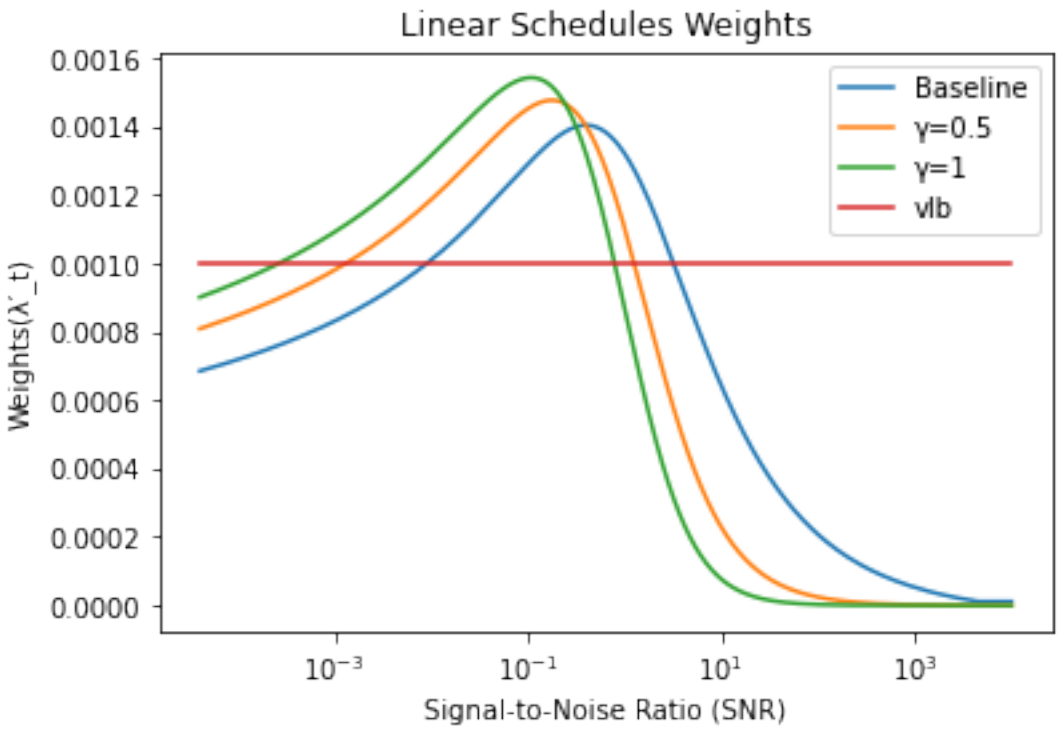
And, here is it in code. $\gamma = 0$ essentially turns the prioritized weighting mechanism off, and gives us the same result as the weighting mechanism in the DDPM paper.

```
k=1. #set for nice math reasons
def prioritized_weights(l,t,g=0.):
    return l/(k + snr(t))**g
```

Here we go ahead and generate weights based on **linear** and **cosine** noise schedules for different values of γ. Notice how it is similar to the results in the image from the paper above.

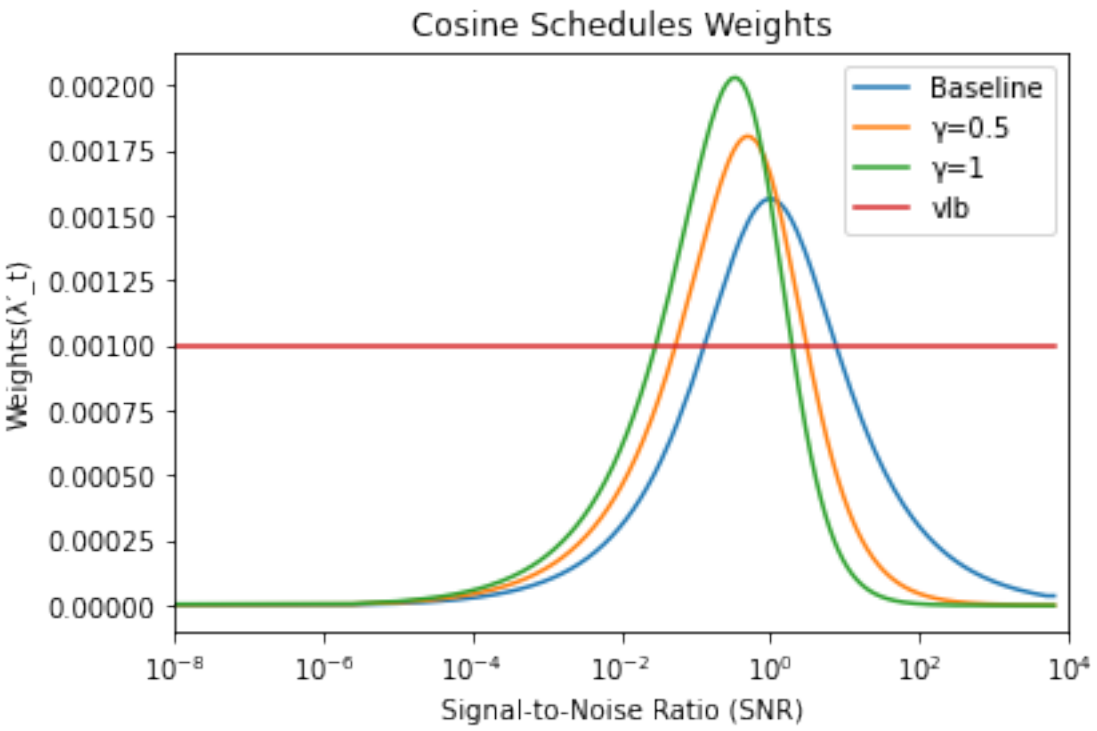
```
plt.xscale('log')
plt.plot(snr(at(Bt)),F.normalize(continuous_weights(at(Bt)),p=1.,dim=0))
plt.plot(snr(at(Bt)),F.normalize(prioritized_weights(continuous_weights(at(Bt)),at(Bt),g=0
plt.plot(snr(at(Bt)),F.normalize(prioritized_weights(continuous_weights(at(Bt)),at(Bt),g=1
plt.plot(snr(at(Bt)),torch.full_like(Bt,0.001))
plt.ylabel('Weights( _t)')
plt.xlabel('Signal-to-Noise Ratio (SNR)')
plt.title('Linear Schedules Weights')
plt.legend(['Baseline',' =0.5',' =1','vlb'])
```

```
<matplotlib.legend.Legend at 0x7fd7e2a625c0>
```




```
plt.xscale('log')
plt.xlim(left=1e-8,right=1e4)
plt.plot(snr(cos_sched(Bt)),F.normalize(continuous_weights(cos_sched(Bt)),p=1.,dim=0))
plt.plot(snr(cos_sched(Bt)),F.normalize(prioritized_weights(continuous_weights(cos_sched(B
plt.plot(snr(cos_sched(Bt)),F.normalize(prioritized_weights(continuous_weights(cos_sched(B
plt.plot(snr(cos_sched(Bt)),torch.full_like(Bt,0.001))
plt.ylabel('Weights( _t)')
plt.xlabel('Signal-to-Noise Ratio (SNR)')
plt.title('Cosine Schedules Weights')
plt.legend(['Baseline',' =0.5',' =1','vlb'])
```

```
<matplotlib.legend.Legend at 0x7fd7e2904d00>
```



Results

Below you can see various results where the models performed better. Note, on the right, the this paper's model is named P2. For the middle table the schedule makes the most difference when a model is missing attention, suggesting the weighting introduced helps with global features. For the images, notice that the samples generated have better clobal features, though both are going well at smaller details. The authors believe this is because the weights help the model focus more on global features.

References

https://arxiv.org/abs/2204.00227

Dataset	Step	FID-50k↓ Base	FID-50k↓ Ours	KID-50k↓ Base	KID-50k↓ Ours
FFHQ	1000	7.86	**6.92**	3.85	**3.46**
	500	8.41	**6.97**	4.48	**3.56**
CUB	1000	9.60	**6.95**	3.49	**2.38**
	250	10.26	**6.32**	4.06	**1.93**
AFHQ-D	1000	12.47	**11.55**	4.79	**4.10**
	250	12.95	**11.66**	5.25	**4.20**
Flowers	250	20.01	**17.29**	16.8	**14.8**
MetFaces	250	44.34	**36.80**	22.1	**17.6**

	FID-10k↓ Base	FID-10k↓ Ours	KID-10k↓ Base	KID-10k↓ Ours
(a)	46.80	41.93 (−4.87)	22.6	20.5 (−2.1)
(b)	47.62	47.37 (−0.25)	23.4	22.7 (−0.7)
(c)	49.56	43.09 (−6.47)	24.3	20.6 (−3.7)
(d)	45.45	42.06 (−3.39)	21.1	18.9 (−2.2)
(e)	46.34	39.51 (−6.83)	23.0	17.4 (−5.6)

Table 3. **Comparison among various model configurations.** (a) Our default configuration (b) No BigGAN block (c) Self-attention only at bottleneck (8×8 resolution) (d) Two residual blocks (e) Learning rate $2.5e^{-5}$. Samples generated with 250 steps.

Dataset	Method	Type	FID↓
FFHQ	BigGAN ICLR'19 [3]	GAN	12.4
	UNet GAN CVPR'20 [37]	GAN	10.9
	StyleGAN2 CVPR'20 [21]	GAN	**3.73**
	NVAE NeurIPS'20 [44]	VAE	26.02
	VDVAE ICLR'21 [5]	VAE	33.5
	VQGAN CVPR'21 [11]	GAN+AR	9.6
	D2C NeurIPS'21 [38]	Diff	13.04
	Baseline (500 step)	Diff	8.41
	P2 (500 step)	Diff	6.97
	P2 (1000 step)	Diff	<u>6.92</u>
Oxford Flower	PGGAN ICLR'18 [17]	GAN	64.40
	StyleGAN1 CVPR'19 [20]	GAN	64.70
	MSG-GAN CVPR'20 [16]	GAN	<u>19.60</u>
	Baseline (250step)	Diff	20.01
	P2 (250step)	Diff	**17.29**
CelebA -HQ	PGGAN ICLR'18 [17]	GAN	8.03
	GLOW NeurIPS'18 [22]	Flow	68.93
	ALAE CVPR'20 [33]	GAN	19.21
	NVAE NeurIPS'20 [44]	VAE	29.76
	VAEBM ICLR'21 [51]	VAE+EM	20.38
	VQGAN CVPR'21 [11]	GAN+AR	10.70
	LSGM NeurIPS'21 [45]	VAE+Diff	<u>7.22</u>
	P2 (500step)	Diff	**6.91**

Figure 2: paper_results.png