def q\_posterior\_mean\_variance(self, x\_start, x\_t, t):  
 *"""  
 Compute the mean and variance of the diffusion posterior:  
 q(x\_{t-1} | x\_t, x\_0)  
 """* assert x\_start.shape == x\_t.shape  
 posterior\_mean = (  
 \_extract\_into\_tensor(self.posterior\_mean\_coef1, t, x\_t.shape) \* x\_start  
 + \_extract\_into\_tensor(self.posterior\_mean\_coef2, t, x\_t.shape) \* x\_t  
 )  
 posterior\_variance = \_extract\_into\_tensor(self.posterior\_variance, t, x\_t.shape)  
 posterior\_log\_variance\_clipped = \_extract\_into\_tensor(  
 self.posterior\_log\_variance\_clipped, t, x\_t.shape  
 )  
 assert (  
 posterior\_mean.shape[0]  
 == posterior\_variance.shape[0]  
 == posterior\_log\_variance\_clipped.shape[0]  
 == x\_start.shape[0]  
 )  
 return posterior\_mean, posterior\_variance, posterior\_log\_variance\_clipped

def p\_mean\_variance(self, model, x, t, clip\_denoised=True, denoised\_fn=None, model\_kwargs=None):

"""

Apply the model to get p(x\_{t-1} | x\_t), as well as a prediction of

the initial x, x\_0.

:param model: the model, which takes a signal and a batch of timesteps

as input.

:param x: the [N x C x ...] tensor at time t.

:param t: a 1-D Tensor of timesteps.

:param clip\_denoised: if True, clip the denoised signal into [-1, 1].

:param denoised\_fn: if not None, a function which applies to the

x\_start prediction before it is used to sample. Applies before

clip\_denoised.

:param model\_kwargs: if not None, a dict of extra keyword arguments to

pass to the model. This can be used for conditioning.

:return: a dict with the following keys:

- 'mean': the model mean output.

- 'variance': the model variance output.

- 'log\_variance': the log of 'variance'.

- 'pred\_xstart': the prediction for x\_0.

"""

if model\_kwargs is None:

model\_kwargs = {}

B, F, C = x.shape[:3]

assert t.shape == (B,)

model\_output = model(x, t, \*\*model\_kwargs)

if isinstance(model\_output, tuple):

model\_output, extra, \_ = model\_output

else:

extra = None

if self.model\_var\_type in [ModelVarType.LEARNED, ModelVarType.LEARNED\_RANGE]:

assert model\_output.shape == (B, F, C \* 2, \*x.shape[3:])

model\_output, model\_var\_values = th.split(model\_output, C, dim=2)

min\_log = \_extract\_into\_tensor(self.posterior\_log\_variance\_clipped, t, x.shape)

max\_log = \_extract\_into\_tensor(np.log(self.betas), t, x.shape)

# The model\_var\_values is [-1, 1] for [min\_var, max\_var].

frac = (model\_var\_values + 1) / 2

model\_log\_variance = frac \* max\_log + (1 - frac) \* min\_log

model\_variance = th.exp(model\_log\_variance)

def process\_xstart(x):

if denoised\_fn is not None:

x = denoised\_fn(x)

if clip\_denoised:

return x.clamp(-1, 1)

return x

if self.model\_mean\_type == ModelMeanType.START\_X:

pred\_xstart = process\_xstart(model\_output)

else:

pred\_xstart = process\_xstart(

self.\_predict\_xstart\_from\_eps(x\_t=x, t=t, eps=model\_output)

)

model\_mean, \_, \_ = self.q\_posterior\_mean\_variance(x\_start=pred\_xstart, x\_t=x, t=t)

assert model\_mean.shape == model\_log\_variance.shape == pred\_xstart.shape == x.shape

return {

"mean": model\_mean,

"variance": model\_variance,

"log\_variance": model\_log\_variance,

"pred\_xstart": pred\_xstart,

"extra": extra,

}

def \_vb\_terms\_bpd(

self, model, x\_start, x\_t, t, clip\_denoised=True, model\_kwargs=None

):

"""

Get a term for the variational lower-bound.

The resulting units are bits (rather than nats, as one might expect).

This allows for comparison to other papers.

:return: a dict with the following keys:

- 'output': a shape [N] tensor of NLLs or KLs.

- 'pred\_xstart': the x\_0 predictions.

"""

true\_mean, \_, true\_log\_variance\_clipped = self.q\_posterior\_mean\_variance(

x\_start=x\_start, x\_t=x\_t, t=t

)

out = self.p\_mean\_variance(

model, x\_t, t, clip\_denoised=clip\_denoised, model\_kwargs=model\_kwargs

)

kl = normal\_kl(

true\_mean, true\_log\_variance\_clipped, out["mean"], out["log\_variance"]

)

kl = mean\_flat(kl) / np.log(2.0)

decoder\_nll = -discretized\_gaussian\_log\_likelihood(

x\_start, means=out["mean"], log\_scales=0.5 \* out["log\_variance"]

)

assert decoder\_nll.shape == x\_start.shape

decoder\_nll = mean\_flat(decoder\_nll) / np.log(2.0)

# At the first timestep return the decoder NLL,

output = th.where((t == 0), decoder\_nll, kl)

return {"output": output, "pred\_xstart": out["pred\_xstart"]}

def training\_losses(self, model, x\_start, t, model\_kwargs=None, noise=None):

"""

Compute training losses for a single timestep.

:param model: the model to evaluate loss on.

:param x\_start: the [N x C x ...] tensor of inputs.

:param t: a batch of timestep indices.

:param model\_kwargs: if not None, a dict of extra keyword arguments to

pass to the model. This can be used for conditioning.

:param noise: if specified, the specific Gaussian noise to try to remove.

:return: a dict with the key "loss" containing a tensor of shape [N].

Some mean or variance settings may also have other keys.

"""

if model\_kwargs is None:

model\_kwargs = {}

if noise is None:

noise = th.randn\_like(x\_start)

x\_t = self.q\_sample(x\_start, t, noise=noise)

terms = {}

elif self.loss\_type == LossType.MSE or self.loss\_type == LossType.RESCALED\_MSE:

model\_output, attentions, features = model(x\_t, t, \*\*model\_kwargs)

attentions, features = torch.mean(attentions, dim=-1), torch.mean(features, dim=-1)

Layer\_norm = nn.LayerNorm(attentions.shape[-1], elementwise\_affine=False, eps=1e-6)

attentions, features = Layer\_norm(attentions), Layer\_norm(features)

L\_prr = 0

per1 = (1 - pearson\_corrcoef(attentions.view(-1), features.view(-1)))

L\_prr = torch.mean(per1)

if self.model\_var\_type in [

ModelVarType.LEARNED,

ModelVarType.LEARNED\_RANGE,

]:

B, F, C = x\_t.shape[:3]

assert model\_output.shape == (B, F, C \* 2, \*x\_t.shape[3:])

model\_output, model\_var\_values = th.split(model\_output, C, dim=2)

# Learn the variance using the variational bound, but don't let

# it affect our mean prediction.

frozen\_out = th.cat([model\_output.detach(), model\_var\_values], dim=2)

terms["vb"] = self.\_vb\_terms\_bpd(

model=lambda \*args, r=frozen\_out: r,

x\_start=x\_start,

x\_t=x\_t,

t=t,

clip\_denoised=False,

)["output"]

if self.loss\_type == LossType.RESCALED\_MSE:

# Divide by 1000 for equivalence with initial implementation.

# Without a factor of 1/1000, the VB term hurts the MSE term.

terms["vb"] \*= self.num\_timesteps / 1000.0

target = {

ModelMeanType.PREVIOUS\_X: self.q\_posterior\_mean\_variance(

x\_start=x\_start, x\_t=x\_t, t=t

)[0],

ModelMeanType.START\_X: x\_start,

ModelMeanType.EPSILON: noise,

}[self.model\_mean\_type]

assert model\_output.shape == target.shape == x\_start.shape

terms["mse"] = mean\_flat((target - model\_output) \*\* 2)

if "vb" in terms:

terms["loss"] = terms["mse"] + terms["vb"]

else:

terms["loss"] = terms["mse"]

if attentions is None:

L\_prr = -1

terms["prr"] = L\_prr

else:

raise NotImplementedError(self.loss\_type)

return terms