DDPM

marii

Add explanation of why subclassing TensorImageBase, and what is TypeDispatch. (title needs to be full name)

We create two new Tensor Types, one for our noise, and one for our timestep.

```
class TensorNoise(TensorImageBase):pass
class TensorStep(TensorBase): pass
```

We would like normalize to denormalize our noise before showing it. This is so the noise in our image looks similar to the noise in our noised image.

```
@Normalize
def decodes(self, x:TensorNoise):
    f = to_cpu if x.device.type=='cpu' else noop
    return (x*f(self.std) + f(self.mean))
```

```
norm = Normalize.from_stats(*imagenet_stats)
show_images(norm.decode(img))
```

I patch ItemTransform here, so that it can work off of TypedTuples. Essentially if we have a `DiffusionTuple`, the transform will apply to that if it should apply to that type of tuple.

```
class ItemTransform(Transform):
    "A transform that always take tuples as items"
    _retain = True
    # Only showing important code
    def _call1(self:ItemTransform, x, name, **kwargs):
        if not _is_tuple(x): return getattr(super(), name)(x, **kwargs)
        y=self._call_tuple(name,x,**kwargs)
        if not self._retain: return y
        if is_listy(y) and not isinstance(y, tuple): y = tuple(y)
        return retain_type(y, x)
    def _call_tuple(self:ItemTransform, name, x, split_idx=None, **kwargs):
        f = getattr(super(), name)
        f2name='encodes' if name == '__call__' else 'decodes' if name == 'decode' else nam
        f2 = getattr(self, f2name)
        if isinstance(f2,TypeDispatch) and f2[type(x)] is not None:
            if split_idx!=self.split_idx and self.split_idx is not None: return x
            y = f2(x, **kwargs)
        else:
            y = f(list(x), **kwargs)
        return y
```

The general idea is to implement a named tuple, and use duck typing. In the future, we should look at the named tuple class and do something more similar to that.

```
class DiffusionTuple(fastuple):
    def __new__(cls, *rest):
        self=super().__new__(cls, *rest)
        i=0
        self.x=self[i]
        if(isinstance(self[i+1],TensorImage)): self.x0=self[i:=i+1]
        self.t=self[i:=i+1]
        if(len(self)>i+1): self.y=self[i:=i+1]
        if(len(self)>i+1): self.pred=self[i:=i+1]
        if(len(self)>i+1): self.sampled_pred=self[i:=i+1]
        return self
```

A little transform to make our tuple a `DiffusionTuple`

```
class ToDiffusionTuple(ItemTransform):
    order=100
    def encodes(self,xy):
```

```
        return DiffusionTuple(*xy[:-1],TensorNoise(xy[-1]))
```

This `Transform` expects y to contain an image, and just replaces it with noise. Our model tries to predict the noise in an image.

```
class LabelToNoise(ItemTransform):
    order=101
    def encodes(self,xy:DiffusionTuple):
        y=xy.y
        xy.y[:]=TensorNoise(torch.randn_like(y))
        return xy
```

```
diff_tuple=LabelToNoise.encodes(DiffusionTuple(img[0].detach().clone(),TensorStep(torch.te
```

We can access tuple elements by attributes. This is useful when you don't know what index a particular value is located.

```
diff_tuple.x.shape,diff_tuple.t.shape,diff_tuple.y.shape
```

```
(torch.Size([3, 320, 480]), torch.Size([1, 1]), torch.Size([3, 320, 480]))
```

Now we have a way to create an image, and convert the label to noise.

```
norm.decode(diff_tuple).show(show_noise=True)
```

```
<AxesSubplot:title={'center':'TensorStep([[200]])'}>
```

Next, we need to go create a noised image, to pass to our model.

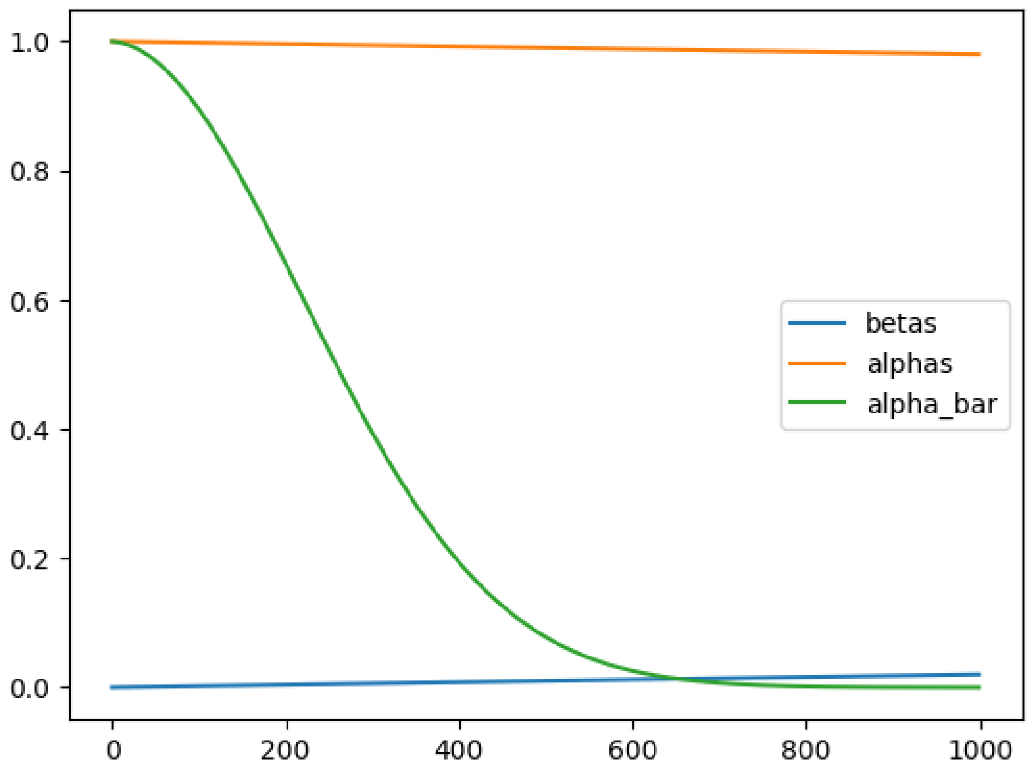
First, how much noise to apply to each step?

```python
class LinearNoiseSchedule:
    "Schedule like used in DDPM"
    def __init__(self,betas=None,n_steps=None,device='cuda'):
        if betas is not None: self.n_steps=betas.shape[0]
        if n_steps is None: self.n_steps=1000
        if betas is None: self.betas = torch.linspace(0.0001, 0.02, self.n_steps,device=de
        self.alphas = 1. - self.betas
        self.alpha_bar = torch.cumprod(self.alphas, dim=0)
```

Lets graph the various values here, in order to see what happens. Pay particularly close attention to `alpha_bar` as that controls the balance betwen our signal(image) and our noise.

```python
lns=LinearNoiseSchedule()
plt.plot((lns.betas).cpu())
plt.plot((lns.alphas).cpu())
plt.plot((lns.alpha_bar).cpu())
plt.legend(['betas', 'alphas','alpha_bar'])
```

```
<matplotlib.legend.Legend at 0x7f89721ee710>
```



Next is DDPM-style Q-sampling. This is pretty much used for all diffusion models, and is the process that takes us from and image to noise.

```
class DDPM_Q_Sampling():
    def __init__(self,predicts_x=False,noise_schedule=LinearNoiseSchedule(),n_steps=1000,d
        self.device=device
        self.ns=noise_schedule
        self.n_steps=n_steps
        self.t_sched=torch.linspace(0,len(self.ns.alpha_bar)-1,n_steps,dtype=torch.long)[.
    def __call__(self,x,es,t):
        t=self.t_sched[t]
        a=self.ns.alpha_bar[t].to(device=x.device)
        signal = (a ** .5)*x
        noise = (1-a)**.5 * es
        return signal + noise
    def undo(self,z,es,t):
        "Goes back to the original image given noise. Only works if es is the original noi
        if(isinstance(es,TensorImage)):
            return es
```

```
        t=self.t_sched[t]
        a=TensorBase(self.ns.alpha_bar[t].to(device=z.device))
        noise=TensorBase((1-a)**.5 * es)
        return TensorImage((z-noise)/(a ** .5))
```

```
diff_trans = DiffusionSamplingTransform(DDPM_Q_Sampling(),lambda x:x)
```

```
norm.decode(diff_trans(diff_tuple)).show()
```

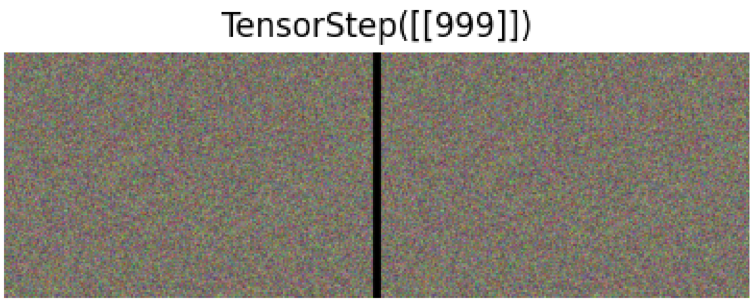
```
<AxesSubplot:title={'center':'TensorStep([[200]])'}>
```

Lets now test so make sure our noise is being generated correctly.

```
noise_tuple=LabelToNoise.encodes(DiffusionTuple(img[0].detach().clone(),TensorStep(torch.t
```

```
norm.decode(diff_trans(noise_tuple)).show(show_noise=True)
```

```
<AxesSubplot:title={'center':'TensorStep([[999]])'}>
```



These are not exactly the same as it is one noising step, but they are fairly close.

```
is_close(norm.decode(diff_trans(noise_tuple))[0],TensorImage(norm.decode(diff_trans(noise_
```

```
TensorImage(True, device='cuda:0')
```

Going from noise to and image, p_sampling

P sampling is going from noise to image!

```
class Diffusion_P_Sampler():
    def __init__(self,model,sampling_function):
        self.device=sampling_function.device
        self.model=model
        self.sampling_function=sampling_function
    # __call__ implemented, but not shown.
    def iter_noise(self,x_t,ts,t_start):
        i=0
        while((ts>0).any()):
            x,t=x_t[ts>0],ts[ts>0]
            with autocast(device_type=self.device, dtype=x.dtype):
                with torch.no_grad():
                    e = self.model(x,self.deconvert(t) if i!=0 else t_start)
                x_t[ts>0]=self.sampling_function(x,e,t,t=t_start if i==0 else None)
            ts[ts>0]-=1
            i+=1
            yield x_t
```

Notice here that we generate noise as random numbers during our sampling process. This makes DDPM sampling a stocastic process.

```
@patch
def __call__(self:DDPM_P_Sampling,x,es,ns_t,t=None):
    t= self.t_sched[ns_t] if(t is None) else t[...,None,None,None]
    n=torch.randn_like(x)
    e,a,b=self._noise_at_t(es,t),self.ns.alphas[t],self.ns.betas[t]
    signal = (x - e) / (a ** 0.5)
    noise = b**.5 * n
    return signal + noise
@patch
def _noise_at_t(self:DDPM_P_Sampling,es,t):
    eps_coef = (1 - self.ns.alphas[t]) / (1 - self.ns.alpha_bar[t]) ** .5
    return eps_coef* es
```

We implement DDIM sampling here, as it drastically reduces sampling time from 1000 steps to 50. DDIM is also a deterministic sampler, meaning we do not have random numbers generated as part of our sampling process. Just generally helps us keep our sanity when trying to show our results. https://arxiv.org/abs/2010.02502

```
@patch
def __call__(self:DDIM_P_Sampling,z,es,ns_t,t=None):
    if(t is None): t=self.t_sched[ns_t]
    tp1=self.t_sched[ns_t-1]
    a,a_tp1=self.ns.alpha_bar[t][...,None,None,None],self.ns.alpha_bar[tp1][...,None,None,
    if self.predicts_x:
        xs=es
        es=(z - (a)**.5 * xs)/(1-a)**.5
    else: xs=(z - (1-a)**.5 * es)/ (a ** .5)
    signal = a_tp1**.5*(xs)
    noise = (1-a_tp1)**.5*es
    return signal + noise
```

Training a model

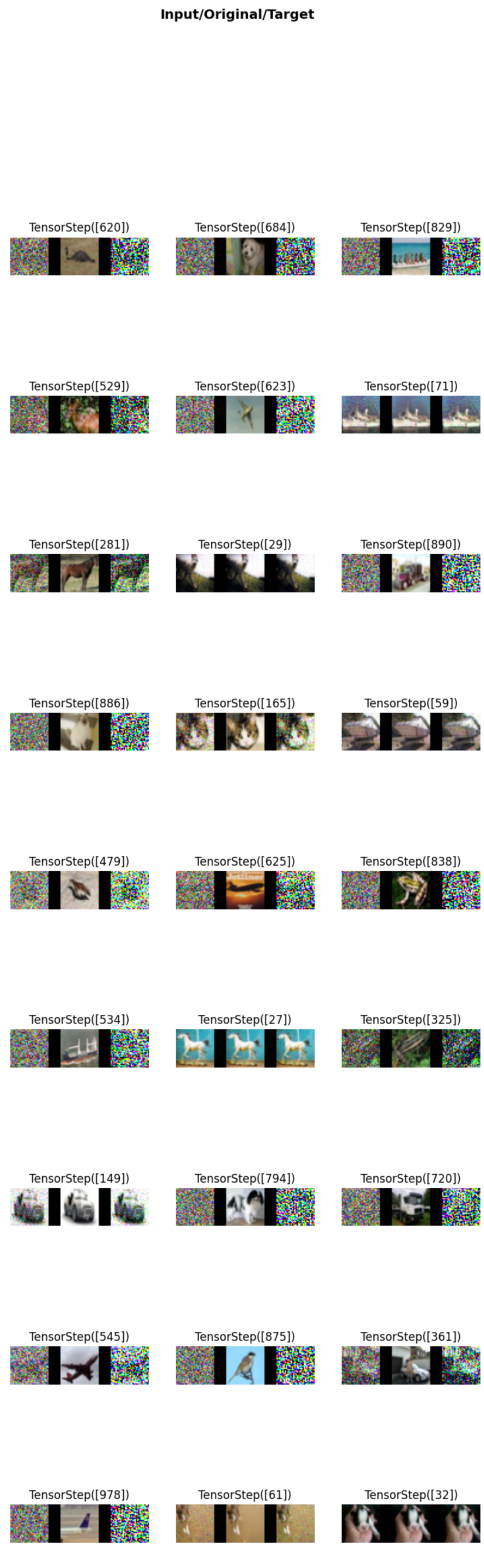
```
path = untar_data(URLs.CIFAR)
```

```
m=Unet(dim=192+192//8,channels=3,).cuda()
```

```
bs=128
n_steps=1000
```

```
diffusion_transform = DiffusionSamplingTransform(DDPM_Q_Sampling(),Diffusion_P_Sampler(m,D
dls=DataBlock((ImageBlock(),
               TransformBlock(type_tfms=[Transform(enc=lambda o: TensorStep(o))]),
               ImageBlock()),
          n_inp=2,
          item_tfms=[Resize(32)],
          batch_tfms=(Normalize.from_stats(*cifar_stats),ToDiffusionTuple,LabelToNoise,dif
          get_items=get_image_files,
          get_x=[lambda x:x,
                 lambda x: torch.randint(1, n_steps, (1,), dtype=torch.long)],
          splitter=IndexSplitter(range(bs)),
).dataloaders(path,bs=bs,val_bs=2*bs)
dls.show_batch()
```

```
/home/molly/miniconda3/envs/fastai/lib/python3.10/site-packages/torch/_tensor.py:1121: UserWa
  ret = func(*args, **kwargs)
```



Input/Original/Target
TensorStep([620])
TensorStep([684])
TensorStep([829])
TensorStep([529])
TensorStep([623])
TensorStep([71])
TensorStep([281])
TensorStep([29])
TensorStep([890])
TensorStep([886])
TensorStep([165])
TensorStep([59])
TensorStep([479])
TensorStep([625])
TensorStep([838])
TensorStep([534])
TensorStep([27])
TensorStep([325])
TensorStep([149])
TensorStep([794])
TensorStep([720])
TensorStep([545])
TensorStep([875])
TensorStep([361])
TensorStep([978])
TensorStep([61])
TensorStep([32])

```
def mse_loss_weighted(ys,targ):
    return torch.mean(targ.w_sched[...,None] * ((ys - targ).flatten(start_dim=1) ** 2))
```

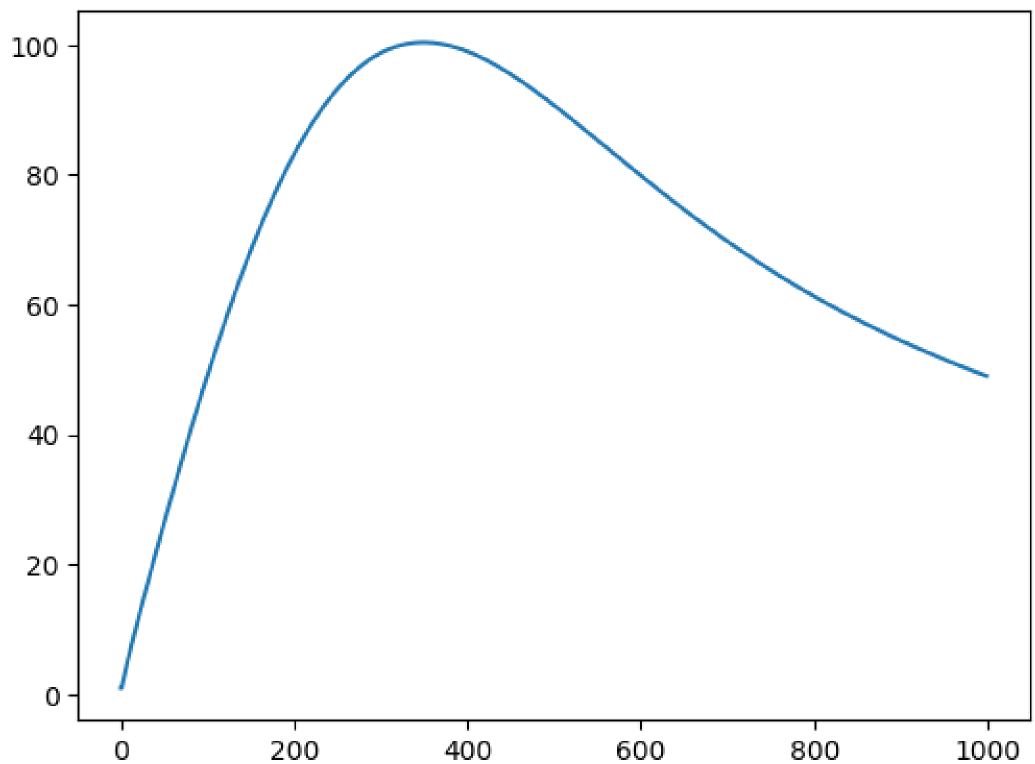
```
def snr(at): return at/(1-at)
```

Taken from a paper called “Perception Prioritized Training of Diffusion Models, this is a continuous version of the weights. Which becomes our signal=to-noise ration, over the change in our signal to noise ratio.”https://arxiv.org/abs/2204.00227

```
def continuous_weights(at):
    weights = -snr(at[1:])/(snr(at[1:])-snr(at[:-1]))
    return torch.cat((weights[0:1],weights))
```

This is what the weights look like. Notice I clip the weights at 1.

```
plt.plot(continuous_weights(LinearNoiseSchedule().alpha_bar).cpu().clip(min=1))
```



```
class WeightedLinSched(Callback):
    def after_pred(self):
        if(not hasattr(self,'ws')):
            self.ws = continuous_weights(LinearNoiseSchedule().alpha_bar).clip(min=1)
            self.ws /= self.ws.mean()
        ts=self.learn.xb[1].flatten()
        self.learn.yb[0].w_sched=self.ws[ts]
```

```
learn = Learner(dls,m,mse_loss_weighted,opt_func=Adam,cbs=[FlattenCallback,WeightedLinSche
learn = learn.to_fp16()
learn.fit_flat_cos(10,lr=2e-4,wd=1e-4)
```

```
<IPython.core.display.HTML object>
```

epoch	train_loss	valid_loss	time
0	0.049204	0.042475	04:47
1	0.043398	0.044983	04:48
2	0.042191	0.034068	04:49
3	0.040477	0.038086	04:48
4	0.039220	0.038858	04:49
5	0.038735	0.032376	04:49
6	0.038864	0.029669	04:49
7	0.038684	0.041698	04:49
8	0.037839	0.031916	04:50
9	0.037525	0.041718	04:50

next check show_results

```
learn.show_results()
```

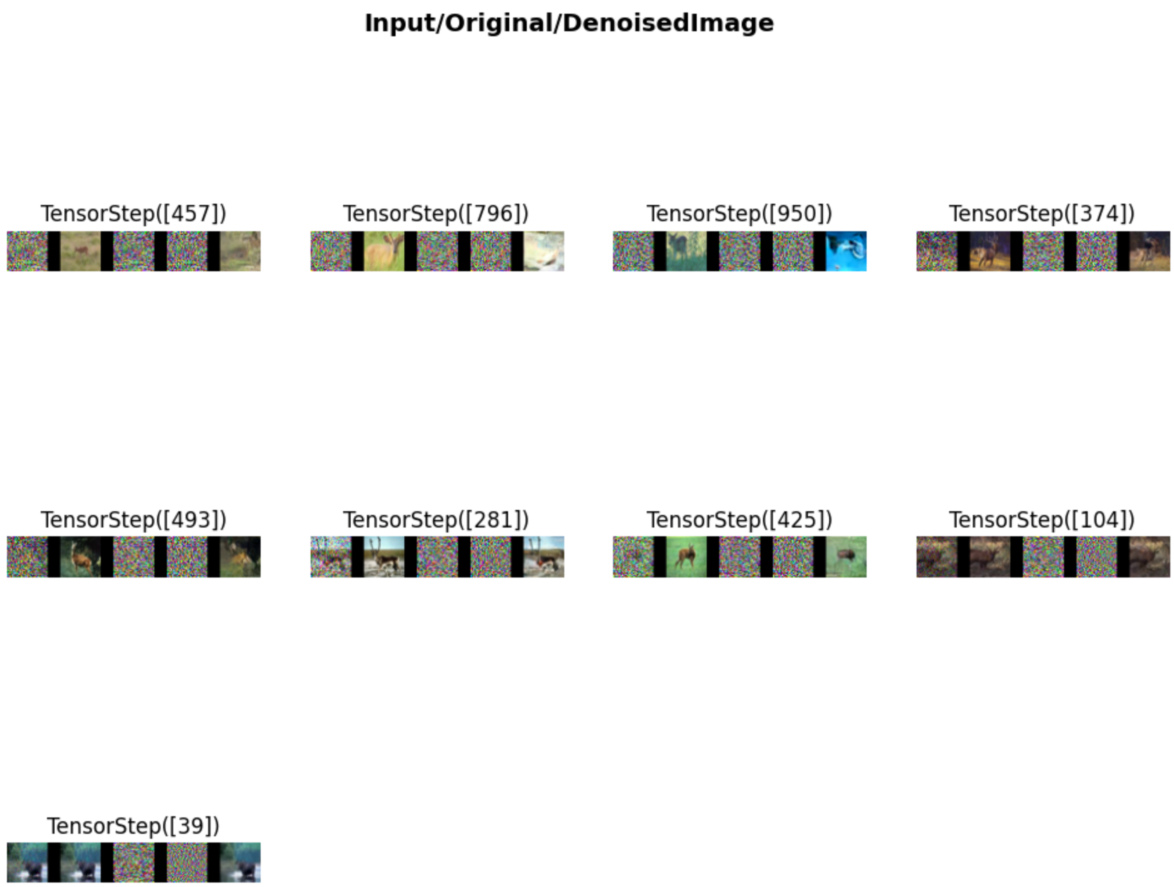
```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```


```
learn.show_results(show_noise=True)
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```



channels last, fused optimizers(foreach benjamin's), jit model.