Unet

```
from fastai.basics import *
from fastai.vision.models.unet import *
from fastai.torch_basics import *
```

```
class SequentialExDict(nn.Sequential):
    "Like `nn.Sequential`, but has a dictionary passed along with x."
    def __init__(self, *layers,dict_names=['seq_dict']):
        super().__init__(*layers)
        self.dict_names=dict_names
    def forward(self, x,**kwargs):
        dicts = getattrs(x,*self.dict_names,default=kwargs)
        for module in self:
            for k,v in zip(self.dict_names,dicts): setattr(x,k,v)
            x = module(x)
        for k,v in zip(self.dict_names,dicts): setattr(x,k,v)
        return x
```

```
class TimeEmbedding(nn.Module):
    """
    ### Embeddings for $t$
    """

    def __init__(self, n_channels: int):
        """
        * `n_channels` is the number of dimensions in the embedding
        """
        super().__init__()
        self.n_channels = n_channels
        # First linear layer
        self.layers = nn.Sequential(
            nn.Linear(self.n_channels // 4, self.n_channels),
            nn.ReLU(True),
            nn.Linear(self.n_channels, self.n_channels)
        )

    def forward(self, x):
        # Create sinusoidal position embeddings
        # [same as those from the transformer](../../transformers/positional_encoding.html
        #
```

```
        # \begin{align}
        # PE^{(1)}_{t,i} &= sin\Bigg(\frac{t}{10000^{\frac{i}{d - 1}}}\Bigg) \\
        # PE^{(2)}_{t,i} &= cos\Bigg(\frac{t}{10000^{\frac{i}{d - 1}}}\Bigg)
        # \end{align}
        #
        # where $d$ is `half_dim`
        t=torch.tensor(x.seq_dict['t']) if isinstance(x.seq_dict['t'],int) else x.seq_dict
        t=t.view(t.shape[0])
        half_dim = self.n_channels // 8
        emb = math.log(10_000) / (half_dim - 1)
        emb = torch.exp(torch.arange(half_dim, device=t.device) * -emb)
        emb = t[:, None] * emb[None, :]
        emb = torch.cat((emb.sin(), emb.cos()), dim=1)

        # Transform with the MLP
        emb = self.layers(emb)
        x.seq_dict['time']=emb
        return x
```

```
class OnKey(nn.Module):
    def __init__(self,k_in,module,k_out=None):
        super().__init__()
        if(k_out is None): k_out=k_in+'_out'
        self.k_in=k_in
        self.k_out=k_out
        self.f=module
    def forward(self, x):
        x.seq_dict[self.k_out]=self.f(x.seq_dict[self.k_in])
        return x
```

```
class Stack(nn.Module):
    def __init__(self,key,f=lambda x:x):
        super().__init__()
        self.key,self.f=key,f
    def forward(self,x):
        if(self.key not in x.seq_dict): x.seq_dict[self.key]=[]
        x.seq_dict[self.key]+=[self.f(x)]
        return x
```

```
class Pop(nn.Module):
    def __init__(self,key,f,clear=True,**kwargs):
        super().__init__()
        self.key,self.clear,self.f,self.kwargs=key,clear,f,kwargs
    def forward(self, x):
        o=x.seq_dict[self.key]
        if(is_listy(o)):
            o =  x.seq_dict[self.key].pop(-1) if(self.clear) else o[-1]
        elif(self.clear): x.seq_dict[self.key]=None
        return self.f(x,o,**self.kwargs)
```

```
def merge(x,o,dense=False): return torch.cat((x,o),dim=1) if(dense) else x+o.view(o.shape+
```

```
class UnetTime(nn.Module):
    "A little Unet with time embeddings"
    def __init__(self,dims=[96, 192, 384, 768, 768],img_channels=3,ks=7,stem_stride=4,t_ch
        super().__init__()
        i_d=0
        h=dims[i_d]
        self.time_emb=TimeEmbedding(t_channels)
        # Not putting in for loop for ease of understanding arch
        self.down=SequentialExDict(
            nn.Conv2d(img_channels,h,ks,1,ks//2),
            Stack('u'),
            Stack('s',lambda x:x.shape[-2:]),
            self.down_sample(h,(h:=dims[(i_d:=i_d+1)]),2,stem_stride,1),
            nn.GroupNorm(1,h),
            Stack('u'),
            Stack('s',lambda x:x.shape[-2:]),
            self.basic_block(h,t_channels,ks=ks),
            self.down_sample(h,(h:=dims[(i_d:=i_d+1)]),2,2,1),
            Stack('u'),
            Stack('s',lambda x:x.shape[-2:]),
            self.basic_block(h,t_channels,ks=ks),
            self.down_sample(h,(h:=dims[(i_d:=i_d+1)]),2,2,1),
            Stack('u'),
            Stack('s',lambda x:x.shape[-2:]),
            self.basic_block(h,t_channels,ks=ks),
            self.down_sample(h,(h:=dims[(i_d:=i_d+1)]),2,2,1),
            Stack('u'),
```

```
        )
        self.middle=SequentialExDict(
            self.basic_block(h,t_channels)
        )
        self.up=SequentialExDict(
            Pop('u',merge,dense=True),
            self.up_sample(h*2,(h:=dims[(i_d:=i_d-1)]),4,1,1),
            self.basic_block(h,t_channels),
            Pop('u',merge,dense=True),
            self.up_sample(h*2,(h:=dims[(i_d:=i_d-1)]),4,1,1),
            self.basic_block(h,t_channels),
            Pop('u',merge,dense=True),
            self.up_sample(h*2,(h:=dims[(i_d:=i_d-1)]),4,1,1),
            self.basic_block(h,t_channels),
            Pop('u',merge,dense=True),
            self.up_sample(h*2,(h:=dims[(i_d:=i_d-1)]),4,1,1),
            self.basic_block(h,t_channels),
            Pop('u',merge,dense=True),
            self.down_sample(h*2,img_channels,5,1,2,bias=True),
            self.basic_block(img_channels,t_channels,bias=True),
        )
        self.layers=SequentialExDict(
            self.time_emb,
            self.down,
            self.middle,
            self.up
        )
    @delegates(nn.Conv2d.__init__)
    def up_sample(self,in_channels,out_channels,kernel_size,stride,padding,**kwargs):
        return SequentialExDict(
            Pop('s',lambda x,o:F.interpolate(x, size=[oi+1 for oi in o], mode='bilinear'))
            self.down_sample(in_channels,out_channels,kernel_size,stride,padding,**kwargs)
        )
    @delegates(nn.Conv2d.__init__)
    def down_sample(self,in_channels,out_channels,kernel_size,stride,padding,**kwargs):
        return SequentialExDict(
            nn.GroupNorm(1,in_channels),
            nn.Conv2d(in_channels,out_channels,kernel_size,stride,padding,**kwargs),
        )
    def basic_block(self,channels,time_channels,expansion=4,ks=7,stride=1,pad=None,bias=Fa
        if pad is None: pad=ks//2
```

```
        return SequentialExDict(
            Stack('r'),
            nn.Conv2d(channels,channels,ks,padding=pad,bias=bias,stride=stride),
            nn.GroupNorm(1,channels),
            nn.Conv2d(channels,channels*expansion,1,bias=bias),
            OnKey('time',nn.Linear(time_channels,channels*expansion)),
            Pop('time_out',merge),
            nn.GELU(),
            nn.Conv2d(channels*expansion,channels,1,bias=bias),
            Pop('r',merge),
        )
    def forward(self,x,x_o,t):
        return self.layers(x,t=t)
```

hello