In [1]:
```python
import torch
import torch.nn as nn
```

0. Prepare Input.

In [2]:
```python
audio = torch.rand(1, 160000)  # 10s audio over 16kHz SR.
x = audio[None].repeat(2, 1, 1)  # make it a batch.
x.shape
```

Out[2]:
```
torch.Size([2, 1, 160000])
```

1. Feature Extractor (Downsample with Conv).

In [3]:
```python
def conv_block(down_rate, c_in=512, c_out=512, kernal_size=3, **kwargs):
    assert kernal_size % 2, 'For simplicity, make sure kernal_size is odd.'
    return nn.Sequential(
        nn.Conv1d(
            in_channels=c_in,
            out_channels=c_out,
            stride=down_rate,
            kernel_size=kernal_size,
            padding=kernal_size // 2,
            **kwargs,
        ),
        nn.GELU()
    )


feature_extractor = torch.nn.Sequential(
    conv_block(5, 1, 512, kernal_size=9),
    *[conv_block(down_rate) for down_rate in [2,] * 6]
)

x = feature_extractor(x)
x.shape
```

Out[3]: torch.Size([2, 512, 500])

2.1 Vector-Quantization

Why VQ? Enfore continuous vector to be some limited representations (Codebook) and therefore prevent overfitting.

What is a Codebook? A trainable group of vectors.

In [4]:
```
from einops import rearrange, einsum


class VectorQuantizer(nn.Module):
    def __init__(self, n_group=2, group_len=320, n_dim=128, feat_in=512):
        super().__init__()
        self.n_group = n_group
        self.scorer = nn.Linear(feat_in, n_group * group_len)
        self.codebook = nn.Parameter(torch.randn(n_group, group_len, n_dim))

    def forward(self, x):
        x = self.scorer(x)  # 'b l (n_group group_len))
        score = rearrange(x, 'b l (g n) -> b l g n', g=self.n_group)
        score = score.softmax(dim=-1)[..., None]  # (b l g n 1)

        codebook = self.codebook[None, None]  # (1 1 g n d)

        q = rearrange((score * codebook).sum(dim=-2), 'b l g d -> b l (g d)')
        return q
```

In [5]:
```
_out = VectorQuantizer()(rearrange(x, 'b d l -> b l d'))
_out.shape
```

Out[5]: torch.Size([2, 500, 256])

Why Gumbel?

Probability Sampling. Here is a example.

In [6]:
```
p = torch.tensor([0.6, 0.3, 0.1])
for _ in range(10):
    print(p.argmax().item(), end=' ')
```

```
0 0 0 0 0 0 0 0 0 0
```

In [7]:
```
def gumbel(p):
    gumbel_noise = - torch.log(- torch.log(torch.rand_like(p)))
    return p + gumbel_noise

for _ in range(10):
    print(gumbel(p).argmax().item(), end=' ')
```

```
0 2 0 1 0 2 0 0 1 2
```

Gumbel Vector-Quantier

In [8]:
```
class GumbelVectorQuantizer(nn.Module):
    def __init__(self, n_group=2, group_len=320, n_dim=128, feat_in=512):
        super().__init__()
        self.n_group = n_group
        self.scorer = nn.Linear(feat_in, n_group * group_len)
        self.codebook = nn.Parameter(torch.randn(n_group, group_len, n_dim))

    def gumbel_softmax(self, p, dim, tau=0.1, eps=1e-8):
        gumbel_noise = - torch.log(- torch.log(torch.rand_like(p) + eps) + eps)
        return ((p + gumbel_noise) / tau).softmax(dim=dim)

    def forward(self, x):
        x = self.scorer(x)  # 'b l (n_group group_len))
        score = rearrange(x, 'b l (g n) -> b l g n', g=self.n_group)
        score = self.gumbel_softmax(score, dim=-1)[..., None]  # (b l g n 1)

        codebook = self.codebook[None, None]  # (1 1 g n d)

        q = rearrange((score * codebook).sum(dim=-2), 'b l g d -> b l (g d)')
        return q
```

In [9]:
```
gumbel_vq = GumbelVectorQuantizer()
q = gumbel_vq(rearrange(x, 'b d l -> b l d'))
```

```
q.shape
```

Out[9]:

```
torch.Size([2, 500, 256])
```

2.2 Mask Feature.

Mask Feature at time dimension.

In [10]:

```
class FeatureMasker(nn.Module):
    def __init__(self, feature_dim=512, n_masks=8, mask_len=10):
        super().__init__()
        # replacer is a leanable vector, not zero vector.
        self.vec_replacer = nn.Parameter(torch.randn(feature_dim))
        self.n_masks = n_masks
        self.mask_len = mask_len

    def random_mask(self, x, fill_value=None):
        b = x.shape[0]
        start_points = torch.randint(0, x.shape[1] - self.mask_len, (b, self.n_masks,))
        end_points = start_points + self.mask_len
        ref = torch.zeros(b, x.shape[1])
        for i in range(self.n_masks):
            s, e = start_points[:, i], end_points[:, i]
            for b_i in range(b):
                ref[b_i, s[b_i]: e[b_i]] = 1

        idx = (ref > 0).nonzero()
        x[idx[:, 0], idx[:, 1]] = self.vec_replacer

        return x, idx

    def forward(self, x):
        # x -> (b l d)
        return self.random_mask(x)
```

In [11]:

```
masker = FeatureMasker()
x, mask_idx = masker(x.permute(0, 2, 1))
x.shape, mask_idx.shape
```

Out[11]: (torch.Size([2, 500, 512]), torch.Size([157, 2]))

2.3 Transformer Encoder (Model Long-range Correlation)

In [12]:
```
class MHAttn(nn.Module):
    def __init__(self, dim=768, n_heads=12):
        super().__init__()
        self.n_heads = n_heads
        self.to_qkv = nn.Linear(dim, dim * 3)
        self.to_out = nn.Linear(dim, dim)
        self.d_root = dim ** 0.5
        self.to_mh = lambda x: rearrange(x, 'b l (h d) -> (b h) l d', h=self.n_heads)
        self.mh_to_d = lambda x: rearrange(x, '(b h) l d -> b l (h d)', h=self.n_heads)

    def forward(self, x):
        q, k, v = list(map(self.to_mh, self.to_qkv(x).chunk(3, dim=-1)))
        attn = (einsum(q, k, 'B i d, B j d -> B i j') / self.d_root).softmax(dim=1)
        return self.to_out(self.mh_to_d(attn @ v))


class FF(nn.Module):
    def __init__(self, dim=768):
        super().__init__()
        self.ln = nn.LayerNorm(normalized_shape=dim)
        self.up = nn.Sequential(
            nn.Linear(dim, dim * 4),
            nn.GELU()
        )
        self.down = nn.Sequential(
            nn.Linear(dim * 4, dim),
            nn.GELU()
        )

    def forward(self, x):
        return self.down(self.up(self.ln(x)))

class TransformerBlock(nn.Module):
    def __init__(self, dim=768):
        super().__init__()
```

```python
        self.attn = MHAttn(dim)
        self.ff = FF(dim)

    def forward(self, x):
        return x + self.ff(self.attn(x))
```

In [13]:
```python
to_transformer_dim = nn.Conv1d(512, 768, kernel_size=1)
pe = conv_block(1, c_in=768, c_out=768, kernal_size=127, groups=12)
transformer_encoder = nn.Sequential(*[TransformerBlock() for _ in range(12)])
```

In [14]:
```python
x = rearrange(x, 'b d l -> b l d')
c = to_transformer_dim(x)
c = c + pe(c)
c = rearrange(c, 'b d l -> b l d')
c = transformer_encoder(c)
c.shape
```

Out[14]:
```
torch.Size([2, 500, 768])
```

3. Self-Supervised Learning

In [15]:
```python
import torch.nn.functional as F
```

In [16]:
```python
to_vq_dim = nn.Linear(768, 256)
c = to_vq_dim(c)
```

In [17]:
```python
c_pred_masked = F.normalize(c[mask_idx[:,0], mask_idx[:, 1]], dim=-1)
q_masked = F.normalize(q[mask_idx[:,0], mask_idx[:, 1]], dim=-1)
c_pred_masked.shape, q_masked.shape
```

Out[17]:
```
(torch.Size([157, 256]), torch.Size([157, 256]))
```

3.1 Main Loss: Predict (from Transformer Encoder) v.s. Qutanized Vecotr

In [18]:
```python
# Similarity Matrix: [N_mask, N_mask]
logits = c_pred_masked @ q_masked.T
labels = torch.arange(logits.shape[0], device=logits.device)  # only pred in diognal is true.
```

```python
temperature = 1.0

loss_nce = torch.nn.functional.cross_entropy(logits / temperature, labels)
```

3.2 Extra Loss: Codebook Usuage; Feature Penalty; ...More

3.2.1 Codebook Usuage

In [19]:
```python
# Encourage Codebook Usuage

class GumbelVectorQuantizer_(GumbelVectorQuantizer):
    """re-implement it to obtain score (of each vec in the code book)"""
    def forward(self, x):
        x = self.scorer(x)  # 'b l (n_group group_len))
        score = rearrange(x, 'b l (g n) -> b l g n', g=self.n_group)
        score = self.gumbel_softmax(score, dim=-1)[..., None]  # (b l g n 1)

        codebook = self.codebook[None, None]  # (1 1 g n d)
        q = rearrange((score * codebook).sum(dim=-2), 'b l g d -> b l (g d)')

        return q, score
```

In [20]:
```python
vq_ = GumbelVectorQuantizer_()
q_, score = vq_(rearrange(x, 'b d l -> b l d'))
```

In [21]:
```python
prob_per_vec = score[..., 0].mean(dim=(0, 1, 2))
prob_per_vec.shape
```

Out[21]:
```
torch.Size([320])
```

In [22]:
```python
loss_codebook = (prob_per_vec * torch.log(prob_per_vec + 1e-9)).sum()
loss_codebook
```

Out[22]:
```
tensor(-5.6982, grad_fn=<SumBackward0>)
```

3.2.3 Feature Penalty

In [23]:
```
# `x` is obtained from (Conv) `feature_extractor(wave)`
loss_feature_pen = (x ** 2).mean()
loss_feature_pen
```

Out[23]:
```
tensor(0.1466, grad_fn=<MeanBackward0>)
```

3.3 Loss for Training.

In [24]:
```
total_loss = loss_nce + 0.1 * loss_codebook + 10 * loss_feature_pen
total_loss
```

Out[24]:
```
tensor(5.9539, grad_fn=<AddBackward0>)
```

4. Finally!

After self-supervised training, we can use trained CNN + Transformer-Encoder as a powerfull feature extractor for downstream tasks.