```python
from math import log2, ceil
from functools import partial
from itertools import zip_longest

import torch
import torch.nn.functional as F
from torch.cuda.amp import autocast
from torch import nn, einsum
from torch.nn import Module, ModuleList

from einops import rearrange, repeat
from einops.layers.torch import Rearrange

# from simple_hierarchical_transformer.attention import Attend

from typing import Tuple
from local_attention import LocalMHA
from zeta import SSM

# constants

Linear = partial(nn.Linear, bias=False)

LocalMHA = partial(LocalMHA, causal=True, prenorm=True)

# helper functions
```

```python
def exists(val):
    return val is not None


def is_power_of_two(n):
    return log2(n).is_integer()


def all_unique(arr):
    return len(set(arr)) == len(arr)


def apply_fns(fns, tensors):
    return [fn(tensor) for fn, tensor in zip(fns, tensors)]


def cast_tuple(t, length=1):
    return t if isinstance(t, tuple) else ((t,) * length)


def default(*vals):
    for val in vals:
        if exists(val):
            return val
```

```python
    return None



def eval_decorator(fn):
    def inner(model, *args, **kwargs):
        was_training = model.training
        model.eval()
        out = fn(model, *args, **kwargs)
        model.train(was_training)
        return out

    return inner



# tensor helpers



def l2norm(t):
    return F.normalize(t, dim=-1)



def cosine_sim_loss(x, y):
    x, y = map(l2norm, (x, y))
    return 1.0 - einsum("b n d, b n d -> b n", x, y).mean()
```

```
# sampling helpers


def log(t, eps=1e-20):
    return t.clamp(min=eps).log()


def gumbel_noise(t):
    noise = torch.zeros_like(t).uniform_(0, 1)
    return -log(-log(noise))


def gumbel_sample(t, temperature=1.0, dim=-1):
    return ((t / max(temperature, 1e-10)) + gumbel_noise(t)).argmax(
        dim=dim
    )


def top_k(logits, thres=0.9):
    k = int((1 - thres) * logits.shape[-1])
    val, ind = torch.topk(logits, k)
    probs = torch.full_like(logits, -torch.finfo(logits.dtype).max)
    probs.scatter_(1, ind, val)
    return probs
```

```
# rotary positional embedding w/ xpos
# https://arxiv.org/abs/2104.09864
# https://arxiv.org/abs/2212.10554v1


class RotaryEmbedding(Module):
    def __init__(self, dim, scale_base=512, use_xpos=True):
        super().__init__()
        inv_freq = 1.0 / (
            10000 ** (torch.arange(0, dim, 2).float() / dim)
        )
        self.register_buffer("inv_freq", inv_freq)

        self.use_xpos = use_xpos
        self.scale_base = scale_base
        scale = (torch.arange(0, dim, 2) + 0.4 * dim) / (1.4 * dim)
        self.register_buffer("scale", scale)

    @property
    def device(self):
        return next(self.buffers()).device

    @autocast(enabled=False)
    def forward(self, seq_len):
        device = self.device
        t = torch.arange(seq_len, device=device).type_as(
```

```python
            self.inv_freq
        )
        freqs = torch.einsum("i , j -> i j", t, self.inv_freq)
        freqs = torch.cat((freqs, freqs), dim=-1)

        if not self.use_xpos:
            return freqs, torch.ones(1, device=device)

        power = (t - (seq_len // 2)) / self.scale_base
        scale = self.scale ** rearrange(power, "n -> n 1")
        scale = torch.cat((scale, scale), dim=-1)

        return freqs, scale


def rotate_half(x):
    x1, x2 = x.chunk(2, dim=-1)
    return torch.cat((-x2, x1), dim=-1)


def apply_rotary_pos_emb(pos, t, scale=1.0):
    seq_len = t.shape[-2]

    pos = pos[..., -seq_len:, :]
    if not isinstance(scale, (int, float)):
        scale = scale[..., -seq_len:, :]
```

```python
    return (t * pos.cos() * scale) + (
        rotate_half(t) * pos.sin() * scale
    )


@autocast(enabled=False)
def apply_rotary_pos_emb_qk(rotary_emb, q, k):
    freqs, scale = rotary_emb
    q = apply_rotary_pos_emb(freqs, q, scale)
    k = apply_rotary_pos_emb(freqs, k, scale**-1)
    return q, k


# token shift, from Peng et al of RWKV


def token_shift(t):
    t, t_shift = t.chunk(2, dim=-1)
    t_shift = F.pad(t_shift, (0, 0, 1, -1))
    return torch.cat((t, t_shift), dim=-1)


# hierarchy related classes
```

```
def pad_seq_to_multiple(t, mult):
    seq_len = t.shape[-2]
    next_seq_len_mult = ceil(seq_len / mult) * mult
    remainder = next_seq_len_mult - seq_len

    if remainder == 0:
        return t, seq_len

    t = F.pad(t, (0, 0, 0, remainder), value=0.0)
    return t, seq_len


def curtail_seq_to_multiple(t, mult):
    seq_len = t.shape[-2]
    prev_seq_len_mult = (seq_len // mult) * mult
    remainder = seq_len - prev_seq_len_mult

    if remainder == 0:
        return t

    t = t[..., :prev_seq_len_mult, :]
    return t


def hierarchical_cat(tokens, strides: Tuple[int, ...]):
    assert len(tokens) == len(strides)
```

```python
    if all([s == 1 for s in strides]):
        return torch.cat(tokens, dim=-1)

    tokens = [
        repeat(t, "b n d -> b (n s) d", s=s)
        for t, s in zip(tokens, strides)
    ]
    min_seq_len = min([t.shape[-2] for t in tokens])
    tokens = [t[..., :min_seq_len, :] for t in tokens]
    return torch.cat(tokens, dim=-1)


class CausalConv(Module):
    def __init__(self, dim_in, dim_out, kernel_size, stride=1):
        super().__init__()
        self.causal_padding = kernel_size - 1
        self.conv = nn.Conv1d(
            dim_in, dim_out, kernel_size, stride=stride
        )

    def forward(self, x):
        x = F.pad(x, (self.causal_padding, 0))
        return self.conv(x)
```

```
class Compress(Module):
    def __init__(
        self,
        *,
        dim,
        dim_out,
        num_tokens=None,
        stride=1,
        compress_factor=1,
        expansion_factor=4,
        dim_head=64,
        heads=8,
        ignore_index=0,
        should_recon=False,
    ):
        super().__init__()
        assert compress_factor > 0 and is_power_of_two(
            compress_factor
        )

        self.stride = stride
        self.no_compress = compress_factor == 1
        self.compress_factor = compress_factor

        self.should_recon = should_recon
```

```
if self.no_compress:
    self.compress_fn = (
        Linear(dim, dim_out)
        if dim != dim_out
        else nn.Identity()
    )
    return

dim_inner = int(dim * expansion_factor)

self.compress_fn = nn.Sequential(
    Rearrange("b n d -> b d n"),
    CausalConv(
        dim, dim_inner, compress_factor, stride=stride
    ),
    nn.SiLU(),
    nn.Conv1d(dim_inner, dim_out, 1),
    Rearrange("b d n -> b n d"),
)

if should_recon:
    assert exists(num_tokens)
    self.to_recon = Linear(
        dim_out, compress_factor * num_tokens
    )
```

```
        self.ignore_index = ignore_index

    def recon(self, h, ids):
        assert self.should_recon

        if self.no_compress:
            return torch.zeros((), device=h.device).requires_grad_()

        c = self.compress_factor
        seq_len = ids.shape[-1]

        recon_logits = self.to_recon(h)
        recon_logits = rearrange(
            recon_logits, "b n (c d) -> (b c) d n", c=c
        )

        recon_ids = F.pad(ids, (c - 1, 0), value=self.ignore_index)
        recon_ids = tuple(
            recon_ids[:, i : (seq_len + i)] for i in range(c)
        )
        recon_ids = torch.stack(recon_ids, dim=1)
        recon_ids = rearrange(recon_ids, "b c n -> (b c) n")

        if self.stride > 1:
            recon_ids = recon_ids[..., :: self.stride]
```

```
        recon_loss = F.cross_entropy(
            recon_logits, recon_ids, ignore_index=self.ignore_index
        )
        return recon_loss

    def forward(self, x):
        return self.compress_fn(x)


class HierarchicalMerge(Module):
    def __init__(self, dims: Tuple[int, ...], dim_out, h_strides=1):
        super().__init__()
        dim = sum(dims)

        strides = cast_tuple(h_strides, len(dims))
        assert len(strides) == len(dims)

        self.strides = strides

        self.net = nn.Sequential(
            RMSNorm(dim),
            nn.Linear(dim, dim_out * 2),
            nn.SiLU(),
            nn.Linear(dim_out * 2, dim_out),
        )
```

```
    def forward(self, tokens):
        x = hierarchical_cat(tokens, self.strides)
        return self.net(x)


# classes


class RMSNorm(Module):
    def __init__(self, dim):
        super().__init__()
        self.scale = dim**0.5
        self.gamma = nn.Parameter(torch.ones(dim))

    def forward(self, x):
        return F.normalize(x, dim=-1) * self.scale * self.gamma


class FeedForward(Module):
    def __init__(self, dim, mult=4):
        super().__init__()
        dim_inner = int(dim * mult)

        self.net = nn.Sequential(
            RMSNorm(dim),
            Linear(dim, dim_inner),
```

```
            nn.GELU(),
            Linear(dim_inner, dim),
        )

    def forward(self, x):
        return self.net(x)


class HierarchicalBlock(Module):
    def __init__(
        self,
        dim,
        dim_head=64,
        heads=8,
        window_size=None,
        compress_factor=1,
        stride=1,
        ff_mult=4,
    ):
        super().__init__()
        self.stride = stride

        assert is_power_of_two(compress_factor)
        self.compress_factor = compress_factor
        self.no_compress = compress_factor == 1
```

```
        assert not exists(window_size) or window_size >= 0
        self.has_attn = window_size != 0

        self.attn = None

        if self.has_attn:
            self.attn = SSM(dim, dim_head, dim, dim)

        self.ff = FeedForward(dim=dim, mult=ff_mult)

    def forward(self, x):
        c = self.compress_factor
        axial_dim = c // self.stride

        x, orig_seq_len = pad_seq_to_multiple(x, axial_dim)

        # hierarchical attention is performed with a simple axial attention

        # this, and using a convolution for compressing at the beginning
        # is one of the improvements on top of hourglass transformer
        # the downside is that the savings are only O(c) instead of O(c ** 2) as in hourglass transformer
          # you can get the O(c ** 2) saving by setting the hierarchical stride == c, but you'll see that performance is much worse, as some tokens will have a c - 1 token gap to the last hierarchical token

        if not self.no_compress:
            x = rearrange(x, "b (n c) d -> (b c) n d", c=axial_dim)
```

```
        if exists(self.attn):
            x = self.attn(token_shift(x)) + x

        x = self.ff(token_shift(x)) + x

        if not self.no_compress:
            x = rearrange(x, "(b c) n d -> b (n c) d", c=axial_dim)

        return x[:, :orig_seq_len]


class HierarchicalTransformer(Module):
    def __init__(
        self,
        *,
        num_tokens,
        dim,
        depth,
        seq_len=2048,
        dim_head=64,
        heads=8,
        ff_mult=4,
        hierarchies=1,
        window_sizes=None,
        hierarchical_stride=1,
```

```
        hierarchy_merge_all=False,  # whether to pass the pooled hierarchical information back to all hierarchies or just one doing the prediction
        predict_hierarchy=None,
        predict_use_all_hierarchy=False,
        recon_loss_weight=0.1,
        hierarchical_ar_loss_weight=0.25,
        ignore_index=0,
        use_flash_attn=False,
    ):
        super().__init__()
        self.seq_len = seq_len

        hierarchies = cast_tuple(hierarchies)
        assert all_unique(
            hierarchies
        ), "hierarchies compression factors must be all unique integers"
        assert all(
            [*map(is_power_of_two, hierarchies)]
        ), "only powers of two allowed for hierarchies"

        self.hierarchies = hierarchies

        # just use a simple tuple list per hyperparameter to customize each hierarchy

        num_hierarchies = len(hierarchies)
```

```
dims = cast_tuple(dim, num_hierarchies)
assert len(dims) == num_hierarchies

window_sizes = cast_tuple(window_sizes, num_hierarchies)
assert len(window_sizes) == num_hierarchies

dim_head = cast_tuple(dim_head, num_hierarchies)
assert len(dim_head) == num_hierarchies

heads = cast_tuple(heads, num_hierarchies)
assert len(heads) == num_hierarchies

ff_mult = cast_tuple(ff_mult, num_hierarchies)
assert len(ff_mult) == num_hierarchies

hierarchical_stride = cast_tuple(
    hierarchical_stride, num_hierarchies
)

assert all(
    [*map(is_power_of_two, hierarchical_stride)]
), "all hierarchical strides must be power of two"
assert all(
    [s <= h for s, h in zip(hierarchical_stride, hierarchies)]
), "all strides must be less than the compression factor of the hierarchy"
```

```
        self.h_strides = hierarchical_stride

        assert len(hierarchical_stride) == num_hierarchies

        # this determines to which hierarchy is everything pooled into for final prediction
        # however, final next token prediction can also use all hierarchies with `predict_use_all_hierarchy`

        predict_hierarchy = default(
            predict_hierarchy, min(hierarchies)
        )
        self.predict_hierarchy_index = hierarchies.index(
            predict_hierarchy
        )
        hierarchy_predict_dim = dims[self.predict_hierarchy_index]

        self.hierarchy_merge_all = hierarchy_merge_all
        assert (
            hierarchy_merge_all
            or self.h_strides[self.predict_hierarchy_index] == 1
        ), "the hierarchy level being used for final next token prediction must have compression stride of 1"

        # training related loss weights

        self.recon_loss_weight = recon_loss_weight
```

```
        should_recon = recon_loss_weight > 0

        self.should_recon = should_recon

        # token embedding

        dim_token_emb = max(dims)
        self.token_emb = nn.Embedding(num_tokens, dim_token_emb)

        # hierarchy ar loss - following the same scheme as done in mirasol paper - cosine sim of prediction to next embedding

        self.hierarchical_ar_loss_weight = hierarchical_ar_loss_weight
        self.has_hierarchical_ar_loss = (
            hierarchical_ar_loss_weight > 0.0
        )

        self.to_hierarchical_preds = ModuleList([])

        for dim, hierarchy in zip(dims, hierarchies):
            linear_pred = (
                nn.Linear(dim, dim) if hierarchy > 1 else None
            )

            self.to_hierarchical_preds.append(linear_pred)
```

```
# hierarchy compressions - 1x just uses the base token_emb weights

self.compressors = ModuleList([])

for dim, hierarchy, stride in zip(
    dims, hierarchies, hierarchical_stride
):
    self.compressors.append(
        Compress(
            dim=dim_token_emb,
            dim_out=dim,
            num_tokens=num_tokens,
            compress_factor=hierarchy,
            stride=stride,
            should_recon=should_recon,
        )
    )

# post token embedding norms

self.post_token_emb_norms = ModuleList(
    [nn.LayerNorm(dim) for dim in dims]
)

# layers
```

```
self.layers = ModuleList([])

self.dims = dims

self.hierarchical_merges = ModuleList([])
self.need_hierarchical_merge = num_hierarchies > 1

for _ in range(depth):
    hierarchical_layer = ModuleList([])

    # add a transformer block for each layer in the hierarchy

    for (
        hierarchy,
        h_stride,
        h_dim,
        h_window_size,
        h_dim_head,
        h_heads,
        h_ff_mult,
    ) in zip(
        hierarchies,
        hierarchical_stride,
        dims,
        window_sizes,
```

```
            dim_head,
            heads,
            ff_mult,
        ):

            # make sure the window size never exceeds the effective sequence length

            effective_seq_len = seq_len // hierarchy

            if (
                exists(h_window_size)
                and h_window_size > effective_seq_len
            ):
                print(
                    f"window size for hierarchy {hierarchy}x is greater than effective sequence length - setting window size to None (which would use normal full attention)"
                )
                h_window_size = None

            # add attention and feedforward

            hierarchical_layer.append(
                HierarchicalBlock(
                    dim=h_dim,
                    dim_head=h_dim_head,
                    heads=h_heads,
```

```
                window_size=h_window_size,
                compress_factor=hierarchy,
                stride=h_stride,
                ff_mult=h_ff_mult,
            )
        )

        self.layers.append(hierarchical_layer)

        # for merging the information across hierarchies
        # for now, only one direction, from all hierarchies to the hierarchy that is being used to make
predictions on, set by predict_hierarchy_index above

        if not self.need_hierarchical_merge:
            continue

        merge = HierarchicalMerge(
            dims=dims,
            dim_out=(
                hierarchy_predict_dim
                if not self.hierarchy_merge_all
                else sum(dims)
            ),
            h_strides=hierarchical_stride,
        )
```

```
        self.hierarchical_merges.append(merge)

    # final post-transformer norms, for all hierarchies

    self.norms = ModuleList([nn.LayerNorm(dim) for dim in dims])

    # to logit, for hierarchy set at predict_hierarchy_index, or all hierarchies

    self.predict_use_all_hierarchy = predict_use_all_hierarchy
    logit_dim_in = (
        sum(dims)
        if predict_use_all_hierarchy
        else hierarchy_predict_dim
    )

    self.to_logits = Linear(logit_dim_in, num_tokens)

    # training related loss parameters

    self.ignore_index = ignore_index

    self.register_buffer(
        "zeros", torch.tensor(0.0), persistent=False
    )

@torch.no_grad()
```

```python
@eval_decorator
def generate(
    self,
    prompt,
    seq_len,
    temperature=1.0,
    filter_thres=0.9,
    **kwargs,
):
    b, t, device = *prompt.shape, prompt.device

    out = prompt

    for _ in range(seq_len):
        logits = self.forward(out[:, -self.seq_len :], **kwargs)[
            :, -1
        ]
        filtered_logits = top_k(logits, thres=filter_thres)
        sample = gumbel_sample(
            filtered_logits, temperature=temperature
        )
        sample = rearrange(sample, "b -> b 1")
        out = torch.cat((out, sample), dim=-1)

    return out[:, t:]
```

```
@property
def device(self):
    return next(self.parameters()).device


def forward(
    self,
    ids,
    return_loss=False,
    return_hierarchical_token_embeds=False,
    return_hierarchical_embeds=False,
    ablate_hierarchical_merge=False,
):
    """
    einops notation:

    b - batch
    n - sequence length
    c - compression factor
    d - dimension
    """


    # if training, predict next token in sequence


    if return_loss:
        ids, labels = ids[:, :-1], ids[:, 1:]
```

```
# assert seq len

assert ids.shape[-1] <= self.seq_len

# get token embeddings, and pad to multiple of compression factor

x = self.token_emb(ids)

    # for every hierarchy, compress token embeddings appropriately to the hierarchical embeddings

tokens = []

for compress in self.compressors:
    tokens.append(compress(x))

# post embedding norms

tokens = apply_fns(self.post_token_emb_norms, tokens)

# if one wants all the compressed token embeds
# just to investigate the space

if return_hierarchical_token_embeds:
    return tokens
```

```
# layers

for layer, merge in zip_longest(
    self.layers, self.hierarchical_merges
):

    tokens = apply_fns(layer, tokens)

    # pool the information all hierarchies
    # and then update the tokens that will be used to make the final autoregressive prediction

    if (
        not self.need_hierarchical_merge
        or ablate_hierarchical_merge
    ):
        continue

    pooled = merge(tokens)

    if self.hierarchy_merge_all:
        tokens = [
            (t + p[..., ::s, :])
            for t, p, s in zip(
                tokens,
                pooled.split(self.dims, dim=-1),
                self.h_strides,
```

```
            )
        ]
    else:
        predict_tokens = tokens[self.predict_hierarchy_index]
        predict_tokens = predict_tokens + pooled
        tokens[self.predict_hierarchy_index] = predict_tokens

# final normalized embeddings

embeds = apply_fns(self.norms, tokens)

# if one wants all the normalized hierarchical embeds

if return_hierarchical_embeds:
    return embeds

# select the hierarchical embeddings that will be doing the predicting

if self.predict_use_all_hierarchy:
    predict_embed = hierarchical_cat(embeds, self.h_strides)
else:
    predict_embed = embeds[self.predict_hierarchy_index]

# logits for predicting next token

logits = self.to_logits(predict_embed)
```

```
if not return_loss:
    return logits

# autoregressive loss (predictive coding)

logits = rearrange(logits, "b n c -> b c n")
ce_loss = F.cross_entropy(
    logits, labels, ignore_index=self.ignore_index
)

# reconstruction losses for hierarchy tokens

recon_losses = self.zeros.requires_grad_()

if self.should_recon:
    for compress, t in zip(self.compressors, embeds):
        recon_loss = compress.recon(t, ids)
        recon_losses = recon_losses + recon_loss

# hierarchical ar loss

hierarchical_ar_losses = self.zeros.requires_grad_()

for h_embed, maybe_h_pred_linear in zip(
    embeds, self.to_hierarchical_preds
```

```
):
    if not exists(maybe_h_pred_linear):
        continue

    h_pred = maybe_h_pred_linear(h_embed)
    h_ar_loss = cosine_sim_loss(
        h_pred[:, :-1], h_embed[:, 1:]
    )

    hierarchical_ar_losses = (
        hierarchical_ar_losses + h_ar_loss
    )

# total loss

total_loss = (
    ce_loss
    + recon_losses * self.recon_loss_weight
    + hierarchical_ar_losses
    * self.hierarchical_ar_loss_weight
)

return total_loss, (
    ce_loss,
    recon_losses,
    hierarchical_ar_losses,
```

)