TransformerLens + PyTorch Quick Reference

ARENA Mech Interp Week - J Rosser

1 TransformerLens: Model Loading

1.1 Load Pretrained Model

```
from transformer_lens import HookedTransformer

model = HookedTransformer.from_pretrained(
    "gpt2-small",
    center_unembed=True,       # Center W_U
    center_writing_weights=True,
    fold_ln=True,              # Fold LayerNorm
    refactor_factored_attn_matrices=True,
    device=device
)
```

1.2 Create Custom Model

```
from transformer_lens import HookedTransformerConfig

cfg = HookedTransformerConfig(
    n_layers=2, n_heads=12, d_model=768,
    d_head=64, d_mlp=3072, d_vocab=50257,
    n_ctx=1024, act_fn="gelu",
    normalization_type="LN",  # or "LNPre", None
    attention_dir="causal",   # or "bidirectional"
    device=device
)
model = HookedTransformer(cfg)
```

1.3 Model Config Access

```
model.cfg.n_layers   # Number of layers
model.cfg.n_heads    # Heads per layer
model.cfg.d_model    # Model dimension
model.cfg.d_head     # Head dimension
model.cfg.d_mlp      # MLP dimension
model.cfg.d_vocab    # Vocabulary size
model.cfg.n_ctx      # Context length
model.cfg.device     # Device
```

2 TransformerLens: Tokenization

```
# Text -> Token IDs (prepends BOS by default)
tokens = model.to_tokens("Hello world")
tokens = model.to_tokens(text, prepend_bos=False)

# Text -> String tokens list
str_toks = model.to_str_tokens("Hello world")
# ['<|endoftext|>', 'Hello', ' world']

# Token IDs -> String
text = model.to_string(tokens)

# Single token operations
tok_id = model.to_single_token(" Paris")
tok_str = model.to_single_str_token(tok_id)

# Batch decode
texts = model.tokenizer.batch_decode(token_ids)
```

3 TransformerLens: Forward Pass

3.1 Basic Forward Pass

```
logits = model(tokens)  # [batch, seq, d_vocab]
loss = model(tokens, return_type="loss")
```

3.2 Forward with Cache

```
logits, cache = model.run_with_cache(tokens)

# Cache only specific activations
logits, cache = model.run_with_cache(
    tokens,
    names_filter=lambda n: n.endswith("pattern"),
    return_type=None  # Don't return logits
)
```

3.3 Forward with Hooks

```
def my_hook(activation, hook):
    # Modify activation
    return activation  # Must return if modifying

logits = model.run_with_hooks(
    tokens,
    fwd_hooks=[
        (utils.get_act_name("z", 0), my_hook),
        ("blocks.1.attn.hook_pattern", other_hook),
    ]
)
```

4 TransformerLens: Cache Access

4.1 Access Patterns

```
# Tuple syntax (preferred)
cache["pattern", layer]       # Attention patterns
cache["q", layer]             # Query vectors
cache["k", layer]             # Key vectors
cache["v", layer]             # Value vectors
cache["z", layer]             # Pre-projection output
cache["result", layer]        # Post-projection output
cache["resid_pre", layer]     # Residual before layer
cache["resid_mid", layer]     # After attn, before MLP
cache["resid_post", layer]    # After layer
cache["attn_out", layer]      # Attention output
cache["mlp_out", layer]       # MLP output
cache["post", layer]          # MLP post-activation
cache["pre", layer]           # MLP pre-activation

# String syntax
cache["blocks.0.attn.hook_pattern"]
```

4.2 Cache Methods

```
# Apply LayerNorm to stacked residuals
scaled = cache.apply_ln_to_stack(
    residual_stack, layer=-1, pos_slice=-1
)

# Get accumulated residual stream
accum, labels = cache.accumulated_resid(
    layer=-1, incl_mid=True,
    pos_slice=-1, return_labels=True
)

# Decompose by component
decomp, labels = cache.decompose_resid(
    layer=-1, pos_slice=-1, return_labels=True
)

# Stack all head outputs
heads, labels = cache.stack_head_results(
    layer=-1, pos_slice=-1, return_labels=True
)
```

5 TransformerLens: Weight Access

```
# Embeddings
model.W_E           # [d_vocab, d_model]
model.W_pos         # [n_ctx, d_model]
model.W_U           # [d_model, d_vocab]

# Attention weights (all have layer, head dims)
model.W_Q[layer, head]  # [d_model, d_head]
model.W_K[layer, head]  # [d_model, d_head]
model.W_V[layer, head]  # [d_model, d_head]
model.W_O[layer, head]  # [d_head, d_model]

# MLP weights
model.W_in[layer]   # [d_model, d_mlp]
model.W_out[layer]  # [d_mlp, d_model]
model.b_in[layer]   # [d_mlp]
model.b_out[layer]  # [d_model]

# Composed circuits
W_OV = model.W_V[l,h] @ model.W_O[l,h]  # OV circuit
W_QK = model.W_Q[l,h] @ model.W_K[l,h].T  # QK circuit
```

6 TransformerLens: Hook Names

```
from transformer_lens import utils

# Get hook name for activation type
utils.get_act_name("pattern", layer)
utils.get_act_name("z", layer)
utils.get_act_name("result", layer)
utils.get_act_name("resid_pre", layer)
utils.get_act_name("resid_post", layer)
utils.get_act_name("q", layer)
utils.get_act_name("k", layer)
utils.get_act_name("v", layer)
utils.get_act_name("mlp_out", layer)
utils.get_act_name("post", layer)  # MLP post-act

# Common hook name patterns
"hook_embed"
"hook_pos_embed"
"blocks.{L}.hook_resid_pre"
"blocks.{L}.attn.hook_pattern"
"blocks.{L}.attn.hook_z"
"blocks.{L}.attn.hook_result"
"blocks.{L}.hook_mlp_out"
"ln_final.hook_normalized"
```

7 TransformerLens: FactoredMatrix

```
from transformer_lens import FactoredMatrix

# Create from two matrices (stores A, B separately)
OV = FactoredMatrix(model.W_V[l,h], model.W_O[l,h])

# Chain operations (stays factored)
full_OV = model.W_E @ OV @ model.W_U

# Materialize only when needed
full_matrix = OV.AB

# Efficient properties
OV.eigenvalues  # Only non-zero eigenvalues
OV.S            # Singular values
OV.norm()       # Frobenius norm
OV.shape        # Shape of full matrix

# Efficient indexing (returns FactoredMatrix)
submatrix = full_OV[indices, indices]
```

8 TransformerLens: Hooks

8.1 Hook Function Patterns

```
from transformer_lens.hook_points import HookPoint
from functools import partial

# Access hook (no return = no modification)
def access_hook(act, hook: HookPoint):
    store[hook.layer()] = act.mean()

# Modify hook (must return tensor)
def modify_hook(act, hook: HookPoint):
    act[:, :, head_idx, :] = 0.0
    return act

# Hook with extra arguments
def param_hook(act, hook, head_idx, scale):
    act[:, :, head_idx, :] *= scale
    return act

hook_fn = partial(param_hook, head_idx=4, scale=0.5)
```

8.2 Hook Management

```
# Reset all hooks
model.reset_hooks()
model.reset_hooks(including_permanent=True)

# Permanent hooks
model.add_hook(hook_name, hook_fn, is_permanent=True)

# Hook context dictionary
hook.ctx["key"] = value
stored = model.hook_dict["hook_name"].ctx["key"]
```

9 SAE-Lens Integration

```
from sae_lens import SAE, HookedSAETransformer

# Load model with SAE support
model = HookedSAETransformer.from_pretrained(
    "gpt2-small", device=device
)

# Load pretrained SAE
sae, cfg, sparsity = SAE.from_pretrained(
    release="gpt2-small-res-jb",
    sae_id="blocks.7.hook_resid_pre",
    device=str(device),
)

# Run with SAE (context manager)
with model.saes(saes=[sae]):
    logits = model(tokens)

# Run with SAE (explicit)
model.add_sae(sae)
logits = model(tokens)
model.reset_saes()  # Don't forget!

# Cache SAE activations
_, cache = model.run_with_cache_with_saes(
    tokens, saes=[sae]
)
sae_acts = cache[f"{sae.cfg.hook_name}.hook_sae_acts_post"]

# SAE weights
sae.W_enc  # [d_in, d_sae] encoder
sae.W_dec  # [d_sae, d_in] decoder
```

10 PyTorch: Device Management

```
import torch as t

# Device selection
device = t.device(
    "mps" if t.backends.mps.is_available()
    else "cuda" if t.cuda.is_available()
    else "cpu"
)
```

```
# Move tensors/models
tensor = tensor.to(device)
model = model.to(device)

# Disable gradients
t.set_grad_enabled(False)

# Inference mode context
with t.inference_mode():
    logits = model(tokens)

# No grad context
with t.no_grad():
    logits = model(tokens)
```

11 PyTorch: Tensor Creation

```
# From data
t.tensor([1, 2, 3])
t.tensor(data, device=device, dtype=t.float32)

# Zeros/Ones
t.zeros(batch, seq, d_model)
t.ones(shape)
t.zeros_like(tensor)
t.ones_like(tensor)

# Random
t.rand(shape)         # Uniform [0, 1)
t.randn(shape)        # Normal(0, 1)
t.randint(0, 100, (batch, seq))
t.randperm(n)         # Random permutation

# Sequences
t.arange(start, end, step)
t.linspace(start, end, steps)

# Special
t.empty(shape)        # Uninitialized
t.full(shape, value)
t.eye(n)              # Identity matrix
```

12 PyTorch: Tensor Operations

```
# Shape operations
tensor.shape           # Get shape
tensor.size(dim)       # Size of dimension
tensor.numel()         # Total elements
tensor.squeeze(dim)    # Remove dim of size 1
tensor.unsqueeze(dim)  # Add dim of size 1
tensor.flatten()       # Flatten to 1D
tensor.flatten(start, end)  # Flatten range
tensor.reshape(shape)
tensor.view(shape)     # Must be contiguous
tensor.T               # Transpose (2D)
tensor.transpose(d1, d2)
tensor.permute(dims)

# Stacking/Concatenation
t.stack([t1, t2], dim=0)  # New dimension
t.cat([t1, t2], dim=0)    # Existing dim
t.concat([t1, t2])        # Alias for cat

# Splitting
t.split(tensor, size, dim)
t.chunk(tensor, chunks, dim)
tensor.unbind(dim)     # Returns tuple
```

13 PyTorch: Math Operations

```
# Elementwise
tensor.abs()
tensor.exp()
tensor.log()
tensor.sqrt()
tensor.pow(n)
tensor.clip(min, max)

# Reductions
tensor.sum(dim)
tensor.mean(dim)
```

```
tensor.std(dim)
tensor.var(dim, unbiased=False)
tensor.max(dim)        # Returns (values, indices)
tensor.min(dim)
tensor.argmax(dim)
tensor.argmin(dim)
tensor.norm(dim=dim, keepdim=True)

# keepdim=True preserves dimension

# Softmax family
tensor.softmax(dim=-1)
tensor.log_softmax(dim=-1)

# Matrix operations
tensor @ other         # Matrix multiply
t.matmul(a, b)
t.bmm(a, b)            # Batched matmul
t.einsum("ij,jk->ik", a, b)

# Comparison
tensor > 0             # Returns bool tensor
tensor.any(dim)
tensor.all(dim)
t.where(cond, x, y)    # Conditional select
```

14 PyTorch: Indexing

```
# Basic slicing
tensor[:, -1]          # Last position
tensor[:, :-1]         # All but last
tensor[..., idx]       # Ellipsis for batch dims

# Gather (select along dim)
values = tensor.gather(dim=-1, index=indices)

# Advanced indexing
tensor[bool_mask]      # Boolean indexing
tensor[idx_tensor]     # Integer indexing
tensor[rows, cols]     # Multi-dim indexing

# Topk
values, indices = tensor.topk(k, dim=-1)

# Sorting
sorted_vals, indices = tensor.sort(dim=-1)
indices = tensor.argsort(descending=True)

# Unique
unique, counts = t.unique(tensor, return_counts=True)

# Nonzero
indices = t.nonzero(tensor > 0)
indices = t.argwhere(tensor > 0)

# Diagonal
tensor.diagonal(offset=0)
tensor.diag(offset)
```

15 PyTorch: In-Place & Masking

```
# In-place operations (end with _)
tensor.zero_()
tensor.fill_(value)
tensor.masked_fill_(mask, value)

# Masking patterns
mask = t.triu(t.ones(seq, seq), diagonal=1).bool()
tensor.masked_fill_(mask, float('-inf'))

# Clone vs reference
new = tensor.clone()  # Copy data
new = tensor.detach() # Detach from graph
new = tensor.detach().clone()  # Both
```

16 PyTorch: Type Conversion

```
tensor.float()         # to float32
tensor.half()          # to float16
tensor.long()          # to int64
tensor.int()           # to int32
tensor.bool()          # to bool
tensor.to(dtype)       # explicit dtype

# To Python
tensor.item()          # Scalar -> Python number
tensor.tolist()        # To Python list
tensor.numpy()         # To numpy (CPU only)
tensor.cpu().numpy()   # Move to CPU first

# From numpy
t.from_numpy(array)
t.tensor(array)        # Copies data
```

17 Einops Patterns

```
import einops

# Rearrange (reshape with named dims)
einops.rearrange(x, "b s h d -> b s (h d)")
einops.rearrange(x, "b (h w) c -> b h w c", h=28)
einops.rearrange(x, "(b1 b2) ... -> b1 b2 ...", b1=2)

# Reduce (aggregation)
einops.reduce(x, "b s d -> b d", "mean")
einops.reduce(x, "b s h d -> b h", "sum")

# Repeat (broadcast)
einops.repeat(x, "d -> b d", b=batch_size)
einops.repeat(x, "h d -> h n d", n=seq_len)

# Einsum (tensor contraction)
einops.einsum(q, k,
    "b s h d, b t h d -> b h s t")
einops.einsum(attn, v,
    "b h s t, b t h d -> b s h d")
einops.einsum(resid, W_U,
    "b s d, d v -> b s v")

# Common attention pattern
einops.einsum(q, k,
    "batch posQ heads dhead, batch posK heads dhead -> batch heads posQ posK")
```

18 Common Patterns: Logit Diff

```
def logits_to_ave_logit_diff(
    logits, answer_tokens, per_prompt=False
):
    """Compute logit difference metric."""
    final_logits = logits[:, -1, :]
    answer_logits = final_logits.gather(
        dim=-1, index=answer_tokens
    )
    correct, incorrect = answer_logits.unbind(dim=-1)
    logit_diff = correct - incorrect
    if per_prompt:
        return logit_diff
    return logit_diff.mean()
```

19 Common Patterns: Activation Patching

```
def patch_hook(
    corrupted_act, hook, pos, clean_cache
):
    """Patch clean activation into corrupted run."""
    corrupted_act[:, pos, :] = clean_cache[hook.name][:, pos, :]
    return corrupted_act

from functools import partial

hook_fn = partial(
    patch_hook, pos=end_pos, clean_cache=clean_cache
)
patched_logits = model.run_with_hooks(
    corrupted_tokens,
    fwd_hooks=[(utils.get_act_name("resid_pre", layer), hook_fn)]
)
```

20 Common Patterns: Ablation

```
# Zero ablation
def zero_ablate_head(z, hook, head_idx):
    z[:, :, head_idx, :] = 0.0
    return z

# Mean ablation
def mean_ablate_head(z, hook, head_idx):
    z[:, :, head_idx, :] = z[:, :, head_idx, :].mean(0)
    return z

# Ablate specific heads
def ablate_heads(z, hook, heads_to_ablate):
    for head in heads_to_ablate:
        z[:, :, head, :] = 0.0
    return z
```

21 Common Patterns: Direct Logit Attribution

```
# Get direction in residual stream
logit_dir = model.W_U[:, correct_tok] - model.W_U[:, incorrect_tok]

# Attribute head outputs
head_outputs = cache["result", layer]  # [batch, seq, heads, d_model]
dla = einops.einsum(
    head_outputs[:, -1], logit_dir,
    "batch heads d_model, d_model -> batch heads"
)

# For SAE latents
sae_acts = cache[sae_acts_hook]  # [batch, seq, d_sae]
dla = sae_acts[:, -1] * (sae.W_dec @ logit_dir)
```

22 CircuitsVis

```
import circuitsvis as cv

# Attention patterns
cv.attention.attention_patterns(
    tokens=model.to_str_tokens(tokens),
    attention=cache["pattern", layer][0],
    attention_head_names=[f"L{l}H{h}" for h in range(n_heads)]
)

# Attention heads view
cv.attention.attention_heads(
    tokens=str_tokens,
    attention=attention_pattern
)
```

23 Jaxtyping Annotations

```
from jaxtyping import Float, Int, Bool
from torch import Tensor

# Common annotations
Float[Tensor, "batch seq d_model"]
Float[Tensor, "batch heads seq_q seq_k"]
Int[Tensor, "batch seq"]
Bool[Tensor, "batch"]

# With variable dims
Float[Tensor, "... d_model"]
Float[Tensor, "*batch seq d_model"]
```

24 Training Loop Patterns

```
optimizer = t.optim.AdamW(
    model.parameters(), lr=1e-4, weight_decay=0.01
)

for epoch in range(n_epochs):
    for batch in dataloader:
        optimizer.zero_grad()

        logits = model(batch)
        loss = compute_loss(logits, targets)

        loss.backward()
        optimizer.step()

        # Optional: learning rate scheduling
        for group in optimizer.param_groups:
            group["lr"] = new_lr
```

25 Useful Imports

```
import torch as t
from torch import Tensor
import einops
from functools import partial
from tqdm import tqdm

from transformer_lens import (
    HookedTransformer,
    HookedTransformerConfig,
    ActivationCache,
    FactoredMatrix,
    utils,
)
from transformer_lens.hook_points import HookPoint

from sae_lens import SAE, HookedSAETransformer

from jaxtyping import Float, Int, Bool

import circuitsvis as cv
import plotly.express as px
```