Setup

```
import google.colab
IN_COLAB = True
print("Running as a Colab notebook")
%pip install git+https://github.com/neelnanda-io/Easy-Transformer.git@clean-tra
!curl -fsSL https://deb.nodesource.com/setup_16.x | sudo -E bash -; sudo apt-ge
%pip install git+https://github.com/neelnanda-io/PySvelte.git
%pip install fancy_einsum
%pip install einops
```

Show hidden output

```
import einops
from fancy_einsum import einsum
from dataclasses import dataclass
from easy_transformer import EasyTransformer
import torch
import torch.nn as nn
import numpy as np
import math
from easy_transformer.utils import get_corner, gelu_new, tokenize_and_concatena
import tqdm.auto as tqdm
```

```
reference_gpt2 = EasyTransformer.from_pretrained("gpt2-small", fold_ln=False, c
```

```
Moving model to device:  cuda
Finished loading pretrained model gpt2-small into EasyTransformer!
```

```
sorted_vocab = sorted(list(reference_gpt2.tokenizer.vocab.items()), key=lambda
print(sorted_vocab[:20])
print()
print(sorted_vocab[250:270])
print()
print(sorted_vocab[990:1010])
print()
```

```
[('!', 0), ('"', 1), ('#', 2), ('$', 3), ('%', 4), ('&', 5), ("'", 6), ('('

[('Ľ', 250), ('Ŀ', 251), ('ľ', 252), ('Ł', 253), ('ł', 254), ('Ń', 255), ('

[('Ġprodu', 990), ('Ġstill', 991), ('led', 992), ('ah', 993), ('Ġhere', 994
```

```
sorted_vocab[-20:]
```

```
[('Revolution', 50237),
 ('Ġsnipers', 50238),
 ('Ġreverted', 50239),
 ('Ġconglomerate', 50240),
 ('Terry', 50241),
 ('794', 50242),
 ('Ġharsher', 50243),
 ('Ġdesolate', 50244),
 ('ĠHitman', 50245),
 ('Commission', 50246),
 ('Ġ(/', 50247),
 ('âĢ¦."', 50248),
 ('Compar', 50249),
 ('Ġamplification', 50250),
 ('ominated', 50251),
 ('Ġregress', 50252),
 ('ĠCollider', 50253),
 ('Ġinformants', 50254),
 ('Ġgazed', 50255),
 ('<|endoftext|>', 50256)]
```

```
print(reference_gpt2.to_tokens("this is an input int the model"))
print(reference_gpt2.to_tokens("dhairya is fine, this is one more input in the
```

```
tensor([[50256,  5661,   318,   281,  5128,   493,   262,  2746]])
tensor([[   67, 27108,  3972,   318,  3734,    11,   428,   318,   530,   5
          5128,   287,   262,  2746]])
```

```
print(reference_gpt2.to_str_tokens("Dhairya Kantawala"))
print(reference_gpt2.to_str_tokens(" Dhairya Kantawala"))
print(reference_gpt2.to_str_tokens(" dhairya"))
print(reference_gpt2.to_str_tokens("dhairyA"))
```

```
['<|endoftext|>', 'D', 'hair', 'ya', ' Kant', 'aw', 'ala']
['<|endoftext|>', ' Dh', 'air', 'ya', ' Kant', 'aw', 'ala']
['<|endoftext|>', ' d', 'hair', 'ya']
['<|endoftext|>', 'dh', 'airy', 'A']
```

```
reference_gpt2.to_str_tokens("56873+3184623=123456789-1000000000")
```

```
['<|endoftext|>',
 '568',
 '73',
 '+',
 '318',
 '46',
 '23',
 '=',
 '123',
 '45',
 '67',
 '89',
 '-',
 '1',
 '000000',
 '000']
```

```
reference_text = "this is going to be an input to my model"
tokens = reference_gpt2.to_tokens(reference_text)
print(tokens)
print(tokens.shape) # this should be batch x position
print(reference_gpt2.to_str_tokens(tokens))
```

```
tensor([[50256,  5661,   318,  1016,   284,   307,   281,  5128,   284,   6
          2746]])
torch.Size([1, 11])
['<|endoftext|>', 'this', ' is', ' going', ' to', ' be', ' an', ' input', '
```

```
tokens = tokens.cuda()
logits, cache = reference_gpt2.run_with_cache(tokens) # batch x position x d_si
print(logits.shape)
print(cache['blocks.0.attn.hook_attn_scores'][0][0])
```

```
torch.Size([1, 11, 50257])
tensor([[ 3.4530e-01, -1.0000e+05, -1.0000e+05, -1.0000e+05, -1.0000e+05,
         -1.0000e+05, -1.0000e+05, -1.0000e+05, -1.0000e+05, -1.0000e+05,
         -1.0000e+05],
        [ 1.0486e+00, -1.6701e+00, -1.0000e+05, -1.0000e+05, -1.0000e+05,
         -1.0000e+05, -1.0000e+05, -1.0000e+05, -1.0000e+05, -1.0000e+05,
         -1.0000e+05],
        [ 5.8880e-01, -9.9642e-01, -2.4995e+00, -1.0000e+05, -1.0000e+05,
         -1.0000e+05, -1.0000e+05, -1.0000e+05, -1.0000e+05, -1.0000e+05,
         -1.0000e+05],
        [ 5.7708e-01, -7.1816e-01, -7.0905e-01, -1.1896e+00, -1.0000e+05,
         -1.0000e+05, -1.0000e+05, -1.0000e+05, -1.0000e+05, -1.0000e+05,
         -1.0000e+05],
        [ 4.4528e-02, -1.1793e+00, -1.7356e+00, -1.6333e+00, -2.9861e+00,
         -1.0000e+05, -1.0000e+05, -1.0000e+05, -1.0000e+05, -1.0000e+05,
         -1.0000e+05],
        [ 3.6138e-01, -1.0978e+00, -1.0631e+00, -1.0377e+00, -2.2664e+00,
         -2.4596e+00, -1.0000e+05, -1.0000e+05, -1.0000e+05, -1.0000e+05,
         -1.0000e+05],
        [ 1.3588e-01, -1.5954e+00, -2.1717e+00, -1.9635e+00, -3.0174e+00,
         -2.4444e+00, -2.6230e+00, -1.0000e+05, -1.0000e+05, -1.0000e+05,
         -1.0000e+05],
        [ 8.9044e-01, -1.5931e+00, -1.1096e+00, -1.1551e+00, -1.1997e+00,
         -1.0497e+00, -1.2446e+00, -1.9092e-01, -1.0000e+05, -1.0000e+05,
         -1.0000e+05],
        [-3.8368e-01, -1.5840e+00, -2.1704e+00, -2.0278e+00, -3.5256e+00,
         -2.4196e+00, -2.6276e+00, -2.0424e+00, -3.4488e+00, -1.0000e+05,
         -1.0000e+05],
        [ 3.1159e-01, -1.4761e+00, -2.4970e+00, -2.1014e+00, -2.9067e+00,
         -2.2389e+00, -2.6010e+00, -2.2527e+00, -2.9522e+00, -2.3705e+00,
         -1.0000e+05],
        [-1.9134e-01, -1.5350e+00, -2.1362e+00, -2.3336e+00, -2.3188e+00,
         -2.3512e+00, -1.2564e+00, -1.5041e+00, -2.2690e+00, -1.0262e+00,
         -5.7729e-01]], device='cuda:0')
```

```
log_probs = logits.log_softmax(dim=-1)
probs = logits.softmax(dim=-1)
print(log_probs.shape)
print(probs.shape)
```

```
torch.Size([1, 11, 50257])
torch.Size([1, 11, 50257])
```

```
list(zip(reference_gpt2.to_str_tokens(reference_text), reference_gpt2.tokenizer
```

```
[('<|endoftext|>', '\n'),
 ('this', ' is'),
 (' is', ' a'),
 (' going', ' to'),
 (' to', ' be'),
 (' be', ' a'),
 (' an', ' interesting'),
 (' input', ' for'),
 (' to', ' the'),
 (' my', ' next'),
 (' model', ',')]
```

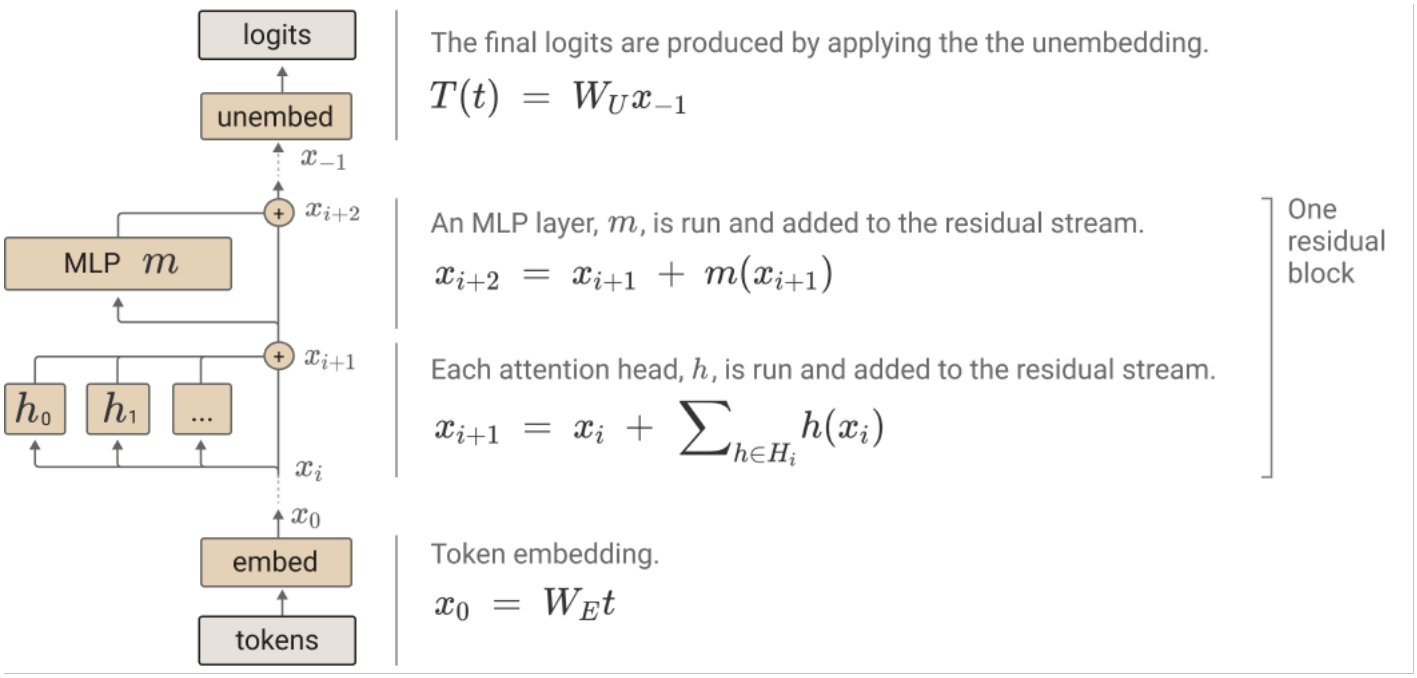
```
next_token = logits[0, -1].argmax(dim=-1)
print(next_token) #... ('+', 10), (',', 11), ('-', 12) ...
```

```
tensor(11, device='cuda:0')
```

```
next_tokens = torch.cat([tokens, torch.tensor(next_token, device='cuda', dtype=
new_logits = reference_gpt2(next_tokens)
print("New Input:", next_tokens)
print(next_tokens.shape)
print("New Input:", reference_gpt2.tokenizer.decode(next_tokens[0]))

print(new_logits.shape)
print(new_logits[-1, -1].argmax(-1))

print(reference_gpt2.tokenizer.decode(new_logits[-1, -1].argmax(-1)))
```

```
New Input: tensor([[50256,  5661,   318,  1016,   284,   307,   281,  5128,
          2746,    11]], device='cuda:0')
torch.Size([1, 12])
New Input: <|endoftext|>this is going to be an input to my model,
torch.Size([1, 12, 50257])
tensor(475, device='cuda:0')
 but
<ipython-input-49-d982d3e85e6a>:1: UserWarning: To copy construct from a te
  next_tokens = torch.cat([tokens, torch.tensor(next_token, device='cuda',
```

Clean Transformer Implementation



Key:

```
batch = 1
position = 20
d_model = 768
n_heads = 12
n_layers = 12
d_mlp = 3072 (4 * d_model)
d_head = 64 (d_model / n_heads)
```

```
reference_text = "this is going to be an input to my model, i want it as big as
tokens = reference_gpt2.to_tokens(reference_text)
tokens = tokens.cuda()
print(f"input token shape: {tokens.shape}")
logits, cache = reference_gpt2.run_with_cache(tokens) # batch x position x d_si
print(f"output logit shape: {logits.shape}")

for activation_name, activation in cache.cache_dict.items():
      print(activation_name, activation.shape)
```

```
blocks.8.hook_resid_mid torch.Size([1, 20, 768])
blocks.8.ln2.hook_scale torch.Size([1, 20, 1])
blocks.8.ln2.hook_normalized torch.Size([1, 20, 768])
blocks.8.mlp.hook_pre torch.Size([1, 20, 3072])
blocks.8.mlp.hook_post torch.Size([1, 20, 3072])
blocks.8.hook_mlp_out torch.Size([1, 20, 768])
```

```
blocks.8.hook_resid_post torch.Size([1, 20, 768])
blocks.9.hook_resid_pre torch.Size([1, 20, 768])
blocks.9.ln1.hook_scale torch.Size([1, 20, 1])
blocks.9.ln1.hook_normalized torch.Size([1, 20, 768])
blocks.9.attn.hook_q torch.Size([1, 20, 12, 64])
blocks.9.attn.hook_k torch.Size([1, 20, 12, 64])
blocks.9.attn.hook_v torch.Size([1, 20, 12, 64])
blocks.9.attn.hook_attn_scores torch.Size([1, 12, 20, 20])
blocks.9.attn.hook_attn torch.Size([1, 12, 20, 20])
blocks.9.attn.hook_z torch.Size([1, 20, 12, 64])
blocks.9.hook_attn_out torch.Size([1, 20, 768])
blocks.9.hook_resid_mid torch.Size([1, 20, 768])
blocks.9.ln2.hook_scale torch.Size([1, 20, 1])
blocks.9.ln2.hook_normalized torch.Size([1, 20, 768])
blocks.9.mlp.hook_pre torch.Size([1, 20, 3072])
blocks.9.mlp.hook_post torch.Size([1, 20, 3072])
blocks.9.hook_mlp_out torch.Size([1, 20, 768])
blocks.9.hook_resid_post torch.Size([1, 20, 768])
blocks.10.hook_resid_pre torch.Size([1, 20, 768])
blocks.10.ln1.hook_scale torch.Size([1, 20, 1])
blocks.10.ln1.hook_normalized torch.Size([1, 20, 768])
blocks.10.attn.hook_q torch.Size([1, 20, 12, 64])
blocks.10.attn.hook_k torch.Size([1, 20, 12, 64])
blocks.10.attn.hook_v torch.Size([1, 20, 12, 64])
blocks.10.attn.hook_attn_scores torch.Size([1, 12, 20, 20])
blocks.10.attn.hook_attn torch.Size([1, 12, 20, 20])
blocks.10.attn.hook_z torch.Size([1, 20, 12, 64])
blocks.10.hook_attn_out torch.Size([1, 20, 768])
blocks.10.hook_resid_mid torch.Size([1, 20, 768])
blocks.10.ln2.hook_scale torch.Size([1, 20, 1])
blocks.10.ln2.hook_normalized torch.Size([1, 20, 768])
blocks.10.mlp.hook_pre torch.Size([1, 20, 3072])
blocks.10.mlp.hook_post torch.Size([1, 20, 3072])
blocks.10.hook_mlp_out torch.Size([1, 20, 768])
blocks.10.hook_resid_post torch.Size([1, 20, 768])
blocks.11.hook_resid_pre torch.Size([1, 20, 768])
blocks.11.ln1.hook_scale torch.Size([1, 20, 1])
blocks.11.ln1.hook_normalized torch.Size([1, 20, 768])
blocks.11.attn.hook_q torch.Size([1, 20, 12, 64])
blocks.11.attn.hook_k torch.Size([1, 20, 12, 64])
blocks.11.attn.hook_v torch.Size([1, 20, 12, 64])
blocks.11.attn.hook_attn_scores torch.Size([1, 12, 20, 20])
blocks.11.attn.hook_attn torch.Size([1, 12, 20, 20])
blocks.11.attn.hook_z torch.Size([1, 20, 12, 64])
blocks.11.hook_attn_out torch.Size([1, 20, 768])
blocks.11.hook_resid_mid torch.Size([1, 20, 768])
blocks.11.ln2.hook_scale torch.Size([1, 20, 1])
blocks.11.ln2.hook_normalized torch.Size([1, 20, 768])
blocks.11.mlp.hook_pre torch.Size([1, 20, 3072])
blocks.11.mlp.hook_post torch.Size([1, 20, 3072])
blocks.11.hook_mlp_out torch.Size([1, 20, 768])
blocks.11.hook_resid_post torch.Size([1, 20, 768])
ln_final.hook_scale torch.Size([1, 20, 1])
```

```
for name, param in reference_gpt2.named_parameters():
    # Only print for first layer
    if ".0." in name or "blocks" not in name:
        print(name, param.shape)
```

```
embed.W_E torch.Size([50257, 768])
pos_embed.W_pos torch.Size([1024, 768])
blocks.0.ln1.w torch.Size([768])
blocks.0.ln1.b torch.Size([768])
blocks.0.ln2.w torch.Size([768])
blocks.0.ln2.b torch.Size([768])
blocks.0.attn.W_Q torch.Size([12, 768, 64])
blocks.0.attn.W_K torch.Size([12, 768, 64])
blocks.0.attn.W_V torch.Size([12, 768, 64])
blocks.0.attn.W_O torch.Size([12, 64, 768])
blocks.0.attn.b_Q torch.Size([12, 64])
blocks.0.attn.b_K torch.Size([12, 64])
blocks.0.attn.b_V torch.Size([12, 64])
blocks.0.attn.b_O torch.Size([768])
blocks.0.mlp.W_in torch.Size([768, 3072])
blocks.0.mlp.b_in torch.Size([3072])
blocks.0.mlp.W_out torch.Size([3072, 768])
blocks.0.mlp.b_out torch.Size([768])
ln_final.w torch.Size([768])
ln_final.b torch.Size([768])
unembed.W_U torch.Size([768, 50257])
unembed.b_U torch.Size([50257])
```

```
print(reference_gpt2.cfg)
```

```
_by_inverse_layer_idx=False, positional_embedding_type='standard', final_rms
```

```
@dataclass
class Config:
    d_model: int = 768
    debug: bool = True
    layer_norm_eps: float = 1e-5
    d_vocab: int = 50257
    init_range: float = 0.02
    n_ctx: int = 1024
    d_head: int = 64
    d_mlp: int = 3072
    n_heads: int = 12
    n_layers: int = 12

cfg = Config()
print(cfg)
```

```
Config(d_model=768, debug=True, layer_norm_eps=1e-05, d_vocab=50257, init_r
```

Tests

```
def rand_float_test(cls, shape):
    cfg = Config(debug=True)
    layer = cls(cfg).cuda()
    random_input = torch.randn(shape).cuda()
    print("Input shape:", random_input.shape)
    output = layer(random_input)
    print("Output shape:", output.shape)
    print()
    return output

def rand_int_test(cls, shape):
    cfg = Config(debug=True)
    layer = cls(cfg).cuda()
    random_input = torch.randint(100, 1000, shape).cuda()
    print("Input shape:", random_input.shape)
    output = layer(random_input)
    print("Output shape:", output.shape)
    print()
    return output

def load_gpt2_test(cls, gpt2_layer, input_name, cache_dict=cache.cache_dict):
    cfg = Config(debug=True)
    layer = cls(cfg).cuda()
    layer.load_state_dict(gpt2_layer.state_dict(), strict=False)
    # Allow inputs of strings or tensors
    if isinstance(input_name, str):
        reference_input = cache_dict[input_name]
    else:
        reference_input = input_name
    print("Input shape:", reference_input.shape)
    output = layer(reference_input)
    print("Output shape:", output.shape)
    reference_output = gpt2_layer(reference_input)
    print("Reference output shape:", reference_output.shape)

    comparison = torch.isclose(output, reference_output, atol=1e-4, rtol=1e-3)
    print(f"{comparison.sum()/comparison.numel():.2%} of the values are correct
    return output
```

LayerNorm

we want to make mean 0 Normalize to have variance 1 Scale with learned weights Translate with learned bias

```
class LayerNorm(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.cfg = cfg
        self.w = nn.Parameter(torch.ones(cfg.d_model))
        self.b = nn.Parameter(torch.zeros(cfg.d_model))

    def forward(self, residual):
        # residual: [batch, position, d_model]
        if cfg.debug:
          print("LayerNorm input shape:", residual.shape)
        residual = residual - einops.reduce(residual, "batch position d_model -
        scale = (einops.reduce(residual.pow(2), "batch position d_model -> batc
        normalized = residual / scale
        normalized = normalized * self.w + self.b
        if cfg.debug:
          print("LayerNorm output shape:", normalized.shape)
        return normalized
```

```
_ = rand_float_test(LayerNorm, [2, 4, 768])
_ = load_gpt2_test(LayerNorm, reference_gpt2.ln_final, "blocks.0.hook_resid_pos
```

```
Input shape: torch.Size([2, 4, 768])
LayerNorm input shape: torch.Size([2, 4, 768])
LayerNorm output shape: torch.Size([2, 4, 768])
Output shape: torch.Size([2, 4, 768])

Input shape: torch.Size([1, 20, 768])
LayerNorm input shape: torch.Size([1, 20, 768])
LayerNorm output shape: torch.Size([1, 20, 768])
Output shape: torch.Size([1, 20, 768])
Reference output shape: torch.Size([1, 20, 768])
100.00% of the values are correct
```

Embedding

Basically a lookup table from tokens to residual stream vectors.

```
class Embed(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.cfg = cfg
        self.W_E = nn.Parameter(torch.empty((cfg.d_vocab, cfg.d_model)))
        nn.init.normal_(self.W_E, std=self.cfg.init_range)

    def forward(self, tokens):
        # tokens: [batch, position]
        if cfg.debug: print("Tokens:", tokens.shape)
        embed = self.W_E[tokens, :] # [batch, position, d_model]
        return embed

rand_int_test(Embed, [2, 4])
load_gpt2_test(Embed, reference_gpt2.embed, tokens)
```

```
Input shape: torch.Size([2, 4])
Tokens: torch.Size([2, 4])
Output shape: torch.Size([2, 4, 768])

Input shape: torch.Size([1, 20])
Tokens: torch.Size([1, 20])
Output shape: torch.Size([1, 20, 768])
Reference output shape: torch.Size([1, 20, 768])
100.00% of the values are correct
tensor([[[ 0.0514, -0.0277,  0.0499,  ...,  0.0070,  0.1552,  0.1207],
         [-0.0788, -0.0764,  0.1948,  ..., -0.1088,  0.0170, -0.1547],
         [-0.0097,  0.0101,  0.0556,  ...,  0.1145, -0.0380, -0.0254],
         ...,
         [ 0.0499, -0.0448,  0.0323,  ...,  0.1662,  0.1075, -0.0307],
         [ 0.1144, -0.0443,  0.1429,  ...,  0.0916, -0.0164,  0.2492],
         [-0.0810,  0.0021,  0.1231,  ..., -0.1585, -0.3604,  0.1450]]],
       device='cuda:0', grad_fn=<IndexBackward0>)
```

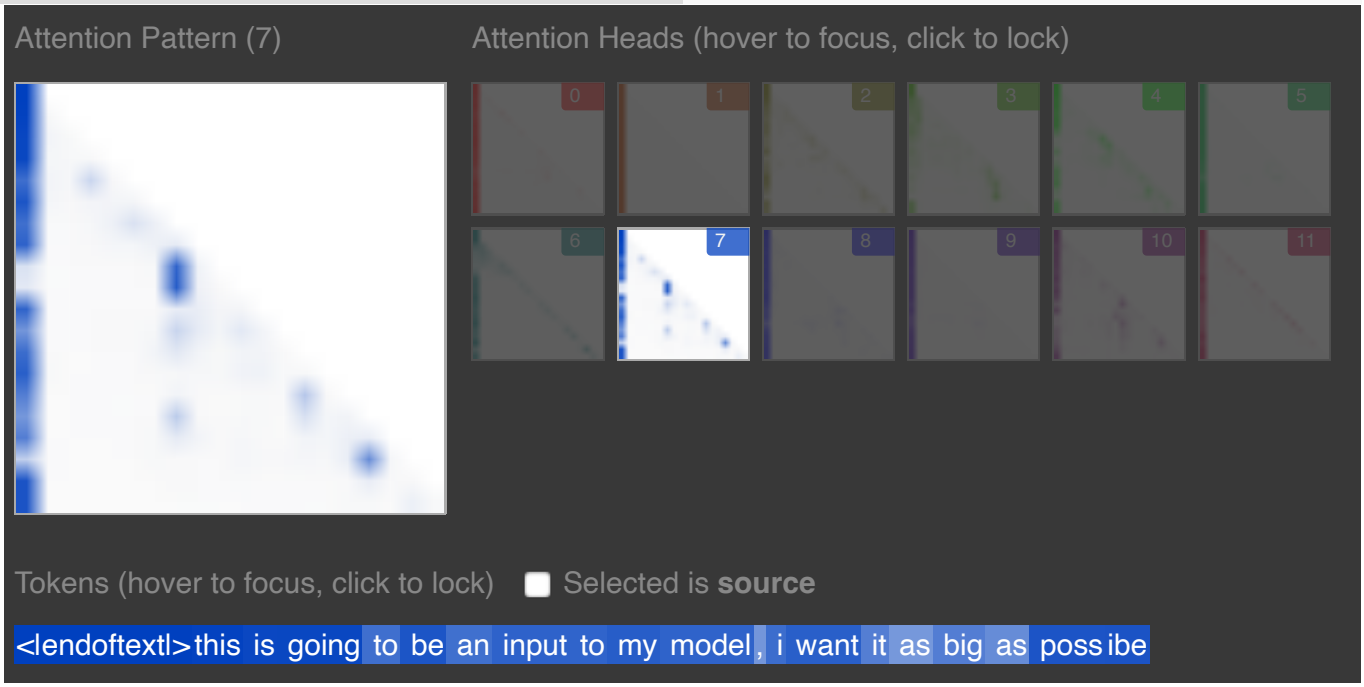
Positional Embedding

```
class PosEmbed(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.cfg = cfg
        self.W_pos = nn.Parameter(torch.empty((cfg.n_ctx, cfg.d_model)))
        nn.init.normal_(self.W_pos, std=self.cfg.init_range)

    def forward(self, tokens):
      #tokens : [batch, position]
      if cfg.debug: print("Tokens:", tokens.shape)
      pos_embed = self.W_pos[:tokens.size(1), :] # [position, d_model]
      pos_embed = einops.repeat(pos_embed, "position d_model -> batch position
      if cfg.debug: print("Pos Embed:", pos_embed.shape)
      return pos_embed

rand_int_test(PosEmbed, [2, 4])
load_gpt2_test(PosEmbed, reference_gpt2.pos_embed, tokens)
```

```
Input shape: torch.Size([2, 4])
Tokens: torch.Size([2, 4])
Pos Embed: torch.Size([2, 4, 768])
Output shape: torch.Size([2, 4, 768])

Input shape: torch.Size([1, 20])
Tokens: torch.Size([1, 20])
Pos Embed: torch.Size([1, 20, 768])
Output shape: torch.Size([1, 20, 768])
Reference output shape: torch.Size([1, 20, 768])
100.00% of the values are correct
tensor([[[-1.8821e-02, -1.9742e-01,  4.0267e-03,  ..., -4.3044e-02,
           2.8267e-02,  5.4490e-02],
         [ 2.3959e-02, -5.3792e-02, -9.4879e-02,  ...,  3.4170e-02,
           1.0172e-02, -1.5573e-04],
         [ 4.2161e-03, -8.4764e-02,  5.4515e-02,  ...,  1.9745e-02,
           1.9325e-02, -2.1424e-02],
         ...,
         [-4.1693e-03,  3.0046e-02,  7.8318e-02,  ..., -4.6164e-03,
          -6.3801e-03, -1.4911e-03],
         [ 7.4972e-04,  2.8626e-02,  7.5494e-02,  ..., -3.7352e-03,
          -2.5456e-03, -2.7157e-03],
         [-6.7148e-03,  3.1997e-02,  8.2699e-02,  ..., -4.1213e-03,
          -4.8707e-03, -1.1040e-03]]], device='cuda:0',
       grad_fn=<ExpandBackward0>)
```

Attention

```
import pysvelte
pysvelte.AttentionMulti(tokens=reference_gpt2.to_str_tokens(reference_text), at
```




```
class Attention(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.cfg = cfg
        self.W_Q = nn.Parameter(torch.empty((cfg.n_heads, cfg.d_model, cfg.d_he
        nn.init.normal_(self.W_Q, std=self.cfg.init_range)
        self.b_Q = nn.Parameter(torch.zeros((cfg.n_heads, cfg.d_head)))
        self.W_K = nn.Parameter(torch.empty((cfg.n_heads, cfg.d_model, cfg.d_he
        nn.init.normal_(self.W_K, std=self.cfg.init_range)
        self.b_K = nn.Parameter(torch.zeros((cfg.n_heads, cfg.d_head)))
        self.W_V = nn.Parameter(torch.empty((cfg.n_heads, cfg.d_model, cfg.d_he
        nn.init.normal_(self.W_V, std=self.cfg.init_range)
        self.b_V = nn.Parameter(torch.zeros((cfg.n_heads, cfg.d_head)))

        self.W_O = nn.Parameter(torch.empty((cfg.n_heads, cfg.d_head, cfg.d_moc
        nn.init.normal_(self.W_O, std=self.cfg.init_range)
        self.b_O = nn.Parameter(torch.zeros((cfg.d_model)))

        self.register_buffer("IGNORE", torch.tensor(-1e5, dtype=torch.float32,

    def forward(self, normalized_resid_pre):
        # normalized_resid_pre: [batch, position, d_model]
        if self.cfg.debug: print("Normalized_resid_pre:", normalized_resid_pre.

        q = einsum("batch query_pos d_model, n_heads d_model d_head -> batch qu
        k = einsum("batch key_pos d_model, n_heads d_model d_head -> batch key_
```

```
        attn_scores = einsum("batch query_pos n_heads d_head, batch key_pos n_h
        attn_scores = attn_scores / math.sqrt(self.cfg.d_head)
        attn_scores = self.apply_causal_mask(attn_scores)

        pattern = attn_scores.softmax(dim=-1) # [batch, n_head, query_pos, key_

        v = einsum("batch key_pos d_model, n_heads d_model d_head -> batch key_

        z = einsum("batch n_heads query_pos key_pos, batch key_pos n_heads d_he

        attn_out = einsum("batch query_pos n_heads d_head, n_heads d_head d_moc
        return attn_out

    def apply_causal_mask(self, attn_scores):
        # attn_scores: [batch, n_heads, query_pos, key_pos]
        mask = torch.triu(torch.ones(attn_scores.size(-2), attn_scores.size(-1)
        attn_scores = attn_scores.masked_fill_(mask, self.IGNORE)
        return attn_scores

rand_float_test(Attention, [2, 4, 768])
load_gpt2_test(Attention, reference_gpt2.blocks[0].attn, cache["blocks.0.ln1.ho
```

```
Input shape: torch.Size([2, 4, 768])
Normalized_resid_pre: torch.Size([2, 4, 768])
Output shape: torch.Size([2, 4, 768])

Input shape: torch.Size([1, 20, 768])
Normalized_resid_pre: torch.Size([1, 20, 768])
Output shape: torch.Size([1, 20, 768])
Reference output shape: torch.Size([1, 20, 768])
100.00% of the values are correct
tensor([[[ 0.7966,  0.0170,  0.0348,  ...,  0.0331, -0.0231,  0.1810],
         [-0.3979,  0.3826, -0.3536,  ...,  0.0148, -0.0343,  0.1501],
         [ 0.2815, -0.2621,  0.0496,  ...,  0.0075, -0.0182,  0.1001],
         ...,
         [ 0.2625,  0.0709, -0.0099,  ..., -0.0388, -0.0109,  0.0391],
         [ 0.1534, -0.5234, -0.6443,  ...,  0.0168, -0.0016,  0.0861],
         [-0.3092, -0.1213, -0.7693,  ...,  0.0361, -0.0422, -0.0036]]],
       device='cuda:0', grad_fn=<AddBackward0>)
```

MLP

```
class MLP(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.cfg = cfg
        self.W_in = nn.Parameter(torch.empty((cfg.d_model, cfg.d_mlp)))
        nn.init.normal_(self.W_in, std=self.cfg.init_range)
        self.b_in = nn.Parameter(torch.zeros((cfg.d_mlp)))
        self.W_out = nn.Parameter(torch.empty((cfg.d_mlp, cfg.d_model)))
        nn.init.normal_(self.W_out, std=self.cfg.init_range)
        self.b_out = nn.Parameter(torch.zeros((cfg.d_model)))

    def forward(self, normalized_resid_mid):
        # normalized_resid_mid: [batch, position, d_model]
        if cfg.debug: print("normalized_resid_mid:", normalized_resid_mid.shape
        pre = einsum("batch position d_model, d_model d_mlp -> batch position d
        post = gelu_new(pre)
        mlp_out = einsum("batch position d_mlp, d_mlp d_model -> batch positior
        return mlp_out

rand_float_test(MLP, [2, 4, 768])
load_gpt2_test(MLP, reference_gpt2.blocks[0].mlp, cache["blocks.0.ln2.hook_norm
```

```
Input shape: torch.Size([2, 4, 768])
normalized_resid_mid: torch.Size([2, 4, 768])
Output shape: torch.Size([2, 4, 768])

Input shape: torch.Size([1, 20, 768])
normalized_resid_mid: torch.Size([1, 20, 768])
Output shape: torch.Size([1, 20, 768])
Reference output shape: torch.Size([1, 20, 768])
100.00% of the values are correct
tensor([[[-0.4380,  0.3624,  0.5117,  ...,  1.7227,  1.5761,  0.0368],
         [-0.2730,  0.9005,  0.3005,  ..., -0.1584, -0.0700,  1.4430],
         [-1.6772,  0.4180, -0.8356,  ...,  0.4434, -0.0948,  1.4874],
         ...,
         [-0.4461,  0.3206,  0.3844,  ...,  0.4620, -0.6832,  0.0261],
         [-1.3384, -0.7578, -0.8362,  ..., -1.1623, -1.1000,  3.5730],
         [-0.6864, -0.2602,  0.4513,  ..., -1.8726,  0.2134,  0.8488]]],
       device='cuda:0', grad_fn=<AddBackward0>)
```

Transformer Block

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

```
        self.ln1 = LayerNorm(cfg)
        self.attn = Attention(cfg)
        self.ln2 = LayerNorm(cfg)
        self.mlp = MLP(cfg)

    def forward(self, resid_pre):
        # resid_pre [batch, position, d_model]
        normalized_resid_pre = self.ln1(resid_pre)
        attn_out = self.attn(normalized_resid_pre)
        resid_mid = resid_pre + attn_out
        normalized_resid_mid = self.ln2(resid_mid)
        mlp_out = self.mlp(normalized_resid_mid)
        resid_post = resid_mid + mlp_out
        return resid_post


rand_float_test(TransformerBlock, [2, 4, 768])
load_gpt2_test(TransformerBlock, reference_gpt2.blocks[0], cache["resid_pre", 0
```

```
Input shape: torch.Size([2, 4, 768])
LayerNorm input shape: torch.Size([2, 4, 768])
LayerNorm output shape: torch.Size([2, 4, 768])
Normalized_resid_pre: torch.Size([2, 4, 768])
LayerNorm input shape: torch.Size([2, 4, 768])
LayerNorm output shape: torch.Size([2, 4, 768])
normalized_resid_mid: torch.Size([2, 4, 768])
Output shape: torch.Size([2, 4, 768])

Input shape: torch.Size([1, 20, 768])
LayerNorm input shape: torch.Size([1, 20, 768])
LayerNorm output shape: torch.Size([1, 20, 768])
Normalized_resid_pre: torch.Size([1, 20, 768])
LayerNorm input shape: torch.Size([1, 20, 768])
LayerNorm output shape: torch.Size([1, 20, 768])
normalized_resid_mid: torch.Size([1, 20, 768])
Output shape: torch.Size([1, 20, 768])
Reference output shape: torch.Size([1, 20, 768])
100.00% of the values are correct
tensor([[[ 0.3911,  0.1543,  0.6005,  ...,  1.7198,  1.7365,  0.3930],
         [-0.7257,  1.1529,  0.0468,  ..., -0.2183, -0.0771,  1.4383],
         [-1.4012,  0.0812, -0.6760,  ...,  0.5851, -0.1317,  1.5406],
         ...,
         [-0.1379,  0.3768,  0.4851,  ...,  0.5847, -0.5930,  0.0330],
         [-1.0698, -1.2969, -1.2621,  ..., -1.0576, -1.1205,  3.9056],
         [-1.0834, -0.3474, -0.1122,  ..., -1.9991, -0.1941,  0.9891]]],
       device='cuda:0', grad_fn=<AddBackward0>)
```

Unembedding

```
class Unembed(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.cfg = cfg
        self.W_U = nn.Parameter(torch.empty((cfg.d_model, cfg.d_vocab)))
        nn.init.normal_(self.W_U, std=self.cfg.init_range)
        self.b_U = nn.Parameter(torch.zeros((cfg.d_vocab), requires_grad=False)

    def forward(self, normalized_resid_final):
        # normalized_resid_final [batch, position, d_model]
        if cfg.debug: print("normalized_resid_final:", normalized_resid_final.s
        logits = einsum("batch position d_model, d_model d_vocab -> batch posit
        return logits

rand_float_test(Unembed, [2, 4, 768])
load_gpt2_test(Unembed, reference_gpt2.unembed, cache["ln_final.hook_normalized
```

```
Input shape: torch.Size([2, 4, 768])
normalized_resid_final: torch.Size([2, 4, 768])
Output shape: torch.Size([2, 4, 50257])

Input shape: torch.Size([1, 20, 768])
normalized_resid_final: torch.Size([1, 20, 768])
Output shape: torch.Size([1, 20, 50257])
Reference output shape: torch.Size([1, 20, 50257])
100.00% of the values are correct
tensor([[[ -43.4317,  -39.8364,  -43.0660,  ...,  -54.0877,  -54.3452,
           -42.3645],
         [ -67.6395,  -67.3921,  -70.2382,  ...,  -76.1376,  -73.9554,
           -68.8402],
         [ -90.5815,  -91.7345,  -93.3314,  ...,  -99.3449,  -98.5336,
           -92.6821],
         ...,
         [ -87.5023,  -87.8412,  -91.5994,  ...,  -95.5364,  -95.3084,
           -88.4299],
         [ -76.3971,  -76.9564,  -78.3513,  ...,  -86.9177,  -85.4052,
           -77.6207],
         [ -90.8475,  -91.4120,  -94.8759,  ..., -102.0653,  -99.6136,
           -91.7585]]], device='cuda:0', grad_fn=<AddBackward0>)
```

Full Transformer

```
class DemoTransformer(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.cfg = cfg
        self.embed = Embed(cfg)
```

```
        self.pos_embed = PosEmbed(cfg)
        self.blocks = nn.ModuleList([TransformerBlock(cfg) for _ in range(cfg.r
        self.ln_final = LayerNorm(cfg)
        self.unembed = Unembed(cfg)

    def forward(self, tokens):
        # tokens [batch, position]
        embed = self.embed(tokens)
        pos_embed = self.pos_embed(tokens)
        resid_pre = embed + pos_embed
        for block in self.blocks:
            resid_pre = block(resid_pre)
        normalized_resid_final = self.ln_final(resid_pre)
        logits = self.unembed(normalized_resid_final)
        return logits

rand_int_test(DemoTransformer, [2, 4])
load_gpt2_test(DemoTransformer, reference_gpt2, tokens)
```

```
LayerNorm output shape: torch.Size([2, 4, 768])
normalized_resid_mid: torch.Size([2, 4, 768])
LayerNorm input shape: torch.Size([2, 4, 768])
LayerNorm output shape: torch.Size([2, 4, 768])
Normalized_resid_pre: torch.Size([2, 4, 768])
LayerNorm input shape: torch.Size([2, 4, 768])
LayerNorm output shape: torch.Size([2, 4, 768])
normalized_resid_mid: torch.Size([2, 4, 768])
LayerNorm input shape: torch.Size([2, 4, 768])
LayerNorm output shape: torch.Size([2, 4, 768])
normalized_resid_final: torch.Size([2, 4, 768])
Output shape: torch.Size([2, 4, 50257])

Input shape: torch.Size([1, 20])
Tokens: torch.Size([1, 20])
Tokens: torch.Size([1, 20])
Pos Embed: torch.Size([1, 20, 768])
LayerNorm input shape: torch.Size([1, 20, 768])
LayerNorm output shape: torch.Size([1, 20, 768])
Normalized_resid_pre: torch.Size([1, 20, 768])
LayerNorm input shape: torch.Size([1, 20, 768])
LayerNorm output shape: torch.Size([1, 20, 768])
normalized_resid_mid: torch.Size([1, 20, 768])
LayerNorm input shape: torch.Size([1, 20, 768])
LayerNorm output shape: torch.Size([1, 20, 768])
Normalized_resid_pre: torch.Size([1, 20, 768])
LayerNorm input shape: torch.Size([1, 20, 768])
LayerNorm output shape: torch.Size([1, 20, 768])
normalized_resid_mid: torch.Size([1, 20, 768])
LayerNorm input shape: torch.Size([1, 20, 768])
LayerNorm output shape: torch.Size([1, 20, 768])
Normalized_resid_pre: torch.Size([1, 20, 768])
LayerNorm input shape: torch.Size([1, 20, 768])
```

```
LayerNorm output shape: torch.Size([1, 20, 768])
normalized_resid_mid: torch.Size([1, 20, 768])
LayerNorm input shape: torch.Size([1, 20, 768])
LayerNorm output shape: torch.Size([1, 20, 768])
Normalized_resid_pre: torch.Size([1, 20, 768])
LayerNorm input shape: torch.Size([1, 20, 768])
LayerNorm output shape: torch.Size([1, 20, 768])
normalized_resid_mid: torch.Size([1, 20, 768])
LayerNorm input shape: torch.Size([1, 20, 768])
LayerNorm output shape: torch.Size([1, 20, 768])
Normalized_resid_pre: torch.Size([1, 20, 768])
LayerNorm input shape: torch.Size([1, 20, 768])
LayerNorm output shape: torch.Size([1, 20, 768])
normalized_resid_mid: torch.Size([1, 20, 768])
LayerNorm input shape: torch.Size([1, 20, 768])
LayerNorm output shape: torch.Size([1, 20, 768])
Normalized_resid_pre: torch.Size([1, 20, 768])
LayerNorm input shape: torch.Size([1, 20, 768])
LayerNorm output shape: torch.Size([1, 20, 768])
normalized_resid_mid: torch.Size([1, 20, 768])
LayerNorm input shape: torch.Size([1, 20, 768])
LayerNorm output shape: torch.Size([1, 20, 768])
Normalized_resid_pre: torch.Size([1, 20, 768])
LayerNorm input shape: torch.Size([1, 20, 768])
LayerNorm output shape: torch.Size([1, 20, 768])
normalized_resid_mid: torch.Size([1, 20, 768])
LayerNorm input shape: torch.Size([1, 20, 768])
```

Let's try it out

```
demo_gpt2 = DemoTransformer(Config(debug=False))
demo_gpt2.load_state_dict(reference_gpt2.state_dict(), strict=False)
demo_gpt2.cuda()
```

```
DemoTransformer(
  (embed): Embed()
  (pos_embed): PosEmbed()
  (blocks): ModuleList(
    (0-11): 12 x TransformerBlock(
      (ln1): LayerNorm()
      (attn): Attention()
      (ln2): LayerNorm()
      (mlp): MLP()
    )
  )
  (ln_final): LayerNorm()
  (unembed): Unembed()
)
```

```
test_string = """Mini scule is a species of microhylid frog endemic to Madagasc
```

```
test_tokens = reference_gpt2.to_tokens(test_string).cuda()
demo_logits = demo_gpt2(test_tokens)
```

```
LayerNorm output shape: torch.Size([1, 237, 768])
normalized_resid_mid: torch.Size([1, 237, 768])
LayerNorm input shape: torch.Size([1, 237, 768])
LayerNorm output shape: torch.Size([1, 237, 768])
LayerNorm input shape: torch.Size([1, 237, 768])
LayerNorm output shape: torch.Size([1, 237, 768])
normalized_resid_mid: torch.Size([1, 237, 768])
LayerNorm input shape: torch.Size([1, 237, 768])
LayerNorm output shape: torch.Size([1, 237, 768])
LayerNorm input shape: torch.Size([1, 237, 768])
LayerNorm output shape: torch.Size([1, 237, 768])
normalized_resid_mid: torch.Size([1, 237, 768])
LayerNorm input shape: torch.Size([1, 237, 768])
LayerNorm output shape: torch.Size([1, 237, 768])
LayerNorm input shape: torch.Size([1, 237, 768])
LayerNorm output shape: torch.Size([1, 237, 768])
normalized_resid_mid: torch.Size([1, 237, 768])
LayerNorm input shape: torch.Size([1, 237, 768])
LayerNorm output shape: torch.Size([1, 237, 768])
LayerNorm input shape: torch.Size([1, 237, 768])
LayerNorm output shape: torch.Size([1, 237, 768])
normalized_resid_mid: torch.Size([1, 237, 768])
LayerNorm input shape: torch.Size([1, 237, 768])
LayerNorm output shape: torch.Size([1, 237, 768])
LayerNorm input shape: torch.Size([1, 237, 768])
LayerNorm output shape: torch.Size([1, 237, 768])
normalized_resid_mid: torch.Size([1, 237, 768])
LayerNorm input shape: torch.Size([1, 237, 768])
LayerNorm output shape: torch.Size([1, 237, 768])
LayerNorm input shape: torch.Size([1, 237, 768])
LayerNorm output shape: torch.Size([1, 237, 768])
normalized_resid_mid: torch.Size([1, 237, 768])
LayerNorm input shape: torch.Size([1, 237, 768])
LayerNorm output shape: torch.Size([1, 237, 768])
LayerNorm input shape: torch.Size([1, 237, 768])
LayerNorm output shape: torch.Size([1, 237, 768])
normalized_resid_mid: torch.Size([1, 237, 768])
LayerNorm input shape: torch.Size([1, 237, 768])
LayerNorm output shape: torch.Size([1, 237, 768])
LayerNorm input shape: torch.Size([1, 237, 768])
LayerNorm output shape: torch.Size([1, 237, 768])
normalized_resid_mid: torch.Size([1, 237, 768])
LayerNorm input shape: torch.Size([1, 237, 768])
LayerNorm output shape: torch.Size([1, 237, 768])
LayerNorm input shape: torch.Size([1, 237, 768])
LayerNorm output shape: torch.Size([1, 237, 768])
normalized_resid_mid: torch.Size([1, 237, 768])
LayerNorm input shape: torch.Size([1, 237, 768])
```

```
    LayerNorm output shape: torch.Size([1, 237, 768])
    LayerNorm input shape: torch.Size([1, 237, 768])
    LayerNorm output shape: torch.Size([1, 237, 768])
    normalized_resid_mid: torch.Size([1, 237, 768])
    LayerNorm input shape: torch.Size([1, 237, 768])
    LayerNorm output shape: torch.Size([1, 237, 768])
    LayerNorm input shape: torch.Size([1, 237, 768])
    LayerNorm output shape: torch.Size([1, 237, 768])
    normalized_resid_mid: torch.Size([1, 237, 768])
    LayerNorm input shape: torch.Size([1, 237, 768])
    LayerNorm output shape: torch.Size([1, 237, 768])
    normalized_resid_final: torch.Size([1, 237, 768])
```

```
def lm_cross_entropy_loss(logits, tokens):
    # Measure next token loss
    # Logits have shape [batch, position, d_vocab]
    # Tokens have shape [batch, position]
    log_probs = logits.log_softmax(dim=-1)
    pred_log_probs = log_probs[:, :-1].gather(dim=-1, index=tokens[:, 1:].unsqu
    return -pred_log_probs.mean()
loss = lm_cross_entropy_loss(demo_logits, test_tokens)
print(loss)
print("Loss as average prob", (-loss).exp())
print("Loss as 'uniform over this many variables'", (loss).exp())
print("Uniform loss over the vocab", math.log(demo_gpt2.cfg.d_vocab))
```

```
tensor(3.7186, device='cuda:0', grad_fn=<NegBackward0>)
Loss as average prob tensor(0.0243, device='cuda:0', grad_fn=<ExpBackward0>
Loss as 'uniform over this many variables' tensor(41.2079, device='cuda:0',
Uniform loss over the vocab 10.82490511970208
```

We can also greedily generate text:

```
test_string = "hi my name is dhairya, what is"
for i in tqdm.tqdm(range(3)):
    test_tokens = reference_gpt2.to_tokens(test_string).cuda()
    demo_logits = demo_gpt2(test_tokens)
    test_string += reference_gpt2.tokenizer.decode(demo_logits[-1, -1].argmax())
```

Show hidden output

```
print(test_string)
```

```
hi my name is dhairya, what is your name?
```

Training a new model

```
if IN_COLAB:
    %pip install datasets
    %pip install transformers
import datasets
import transformers
import plotly.express as px
```

Show hidden output

Config

```
batch_size = 8
num_epochs = 1
max_steps = 1000
log_every = 10
lr = 1e-3
weight_decay = 1e-2
model_cfg = Config(debug=False, d_model=256, n_heads=4, d_head=64, d_mlp=1024,
```

Create Data

We load in a tiny dataset I made, with the first 10K entries in the Pile (inspired by Stas' version for OpenWebText!)

```
dataset = datasets.load_dataset("NeelNanda/pile-10k", split="train")
print(dataset)
print(dataset[0]['text'][:100])
tokens_dataset = tokenize_and_concatenate(dataset, reference_gpt2.tokenizer, st
data_loader = torch.utils.data.DataLoader(tokens_dataset, batch_size=batch_size
```

```
README.md: 100% 373/373 [00:00<00:00, 39.0kB/s]
dataset_infos.json: 100% 921/921 [00:00<00:00, 71.5kB/s]
(…)-00000-of-00001-4746b8785c874cc7.parquet: 100% 33.3M/33.3M [00:00<00:00, 257MB/s]
Generating train split: 100% 10000/10000 [00:00<00:00, 21575.80 examples/s]
Dataset({
    features: ['text', 'meta'],
    num_rows: 10000
})
It is done, and submitted. You can play “Survival of the Tastiest” on Andro
Map (num_proc=4): 100% 10000/10000 [00:59<00:00, 250.25 examples/s]
Token indices sequence length is longer than the specified maximum sequence
Token indices sequence length is longer than the specified maximum sequence
Token indices sequence length is longer than the specified maximum sequence
```

Create Model

```
model = DemoTransformer(model_cfg)
model.cuda()
```

```
DemoTransformer(
  (embed): Embed()
  (pos_embed): PosEmbed()
  (blocks): ModuleList(
    (0-1): 2 x TransformerBlock(
      (ln1): LayerNorm()
      (attn): Attention()
      (ln2): LayerNorm()
      (mlp): MLP()
    )
  )
  (ln_final): LayerNorm()
  (unembed): Unembed()
)
```

Create Optimizer

We use AdamW - it's a pretty standard optimizer.

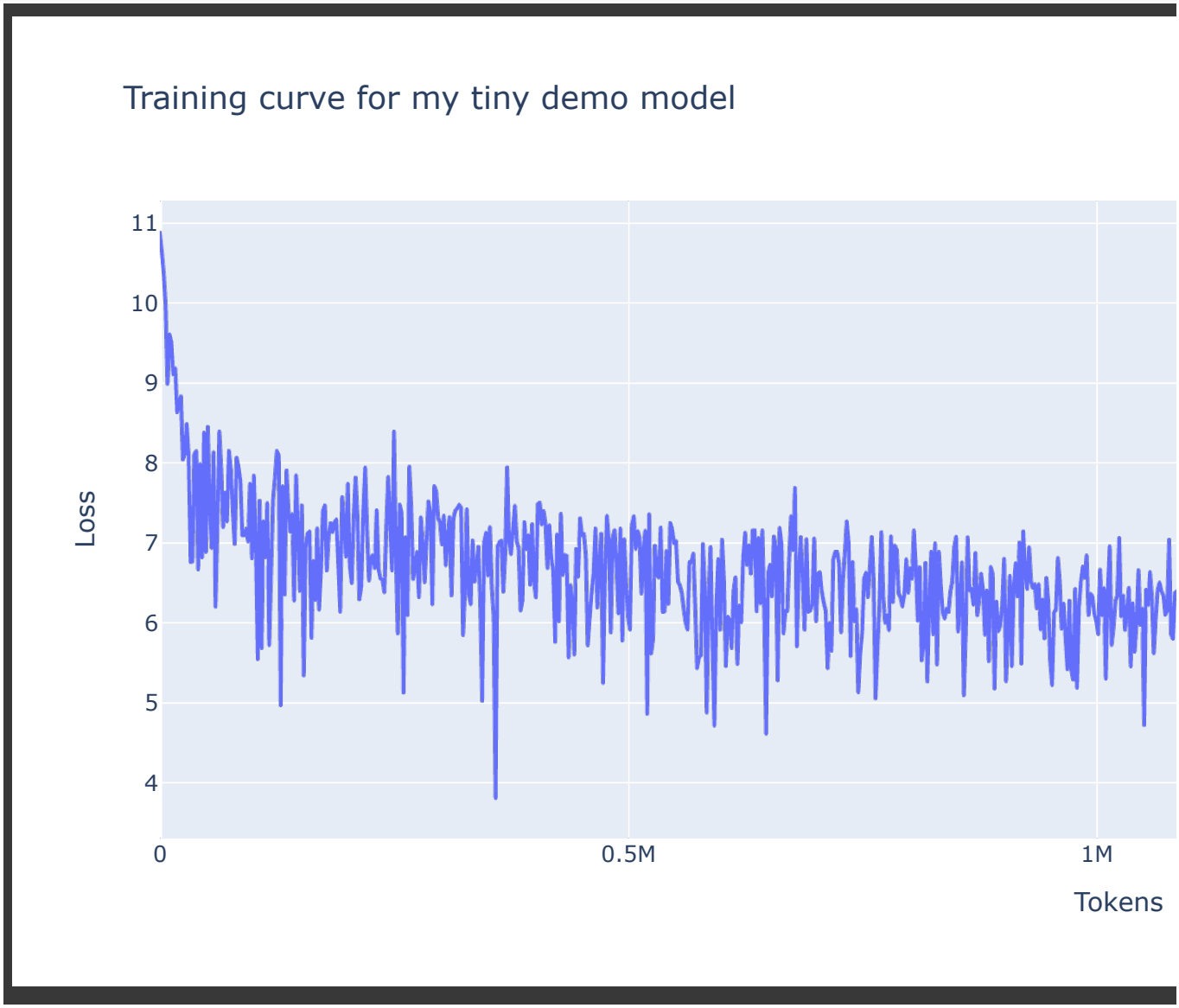
```
optimizer = torch.optim.AdamW(model.parameters(), lr=lr, weight_decay=weight_de
```

Run Training Loop

```
losses = []
print("Number of batches:", len(data_loader))
for epoch in range(num_epochs):
    for c, batch in tqdm.tqdm(enumerate(data_loader)):
        tokens = batch['tokens'].cuda()
        logits = model(tokens)
        loss = lm_cross_entropy_loss(logits, tokens)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
        losses.append(loss.item())
        if c % log_every == 0:
            print(f"Step: {c}, Loss: {loss.item():.4f}")
        if c > max_steps:
            break
```

Show hidden output

```
import plotly.express as px
import numpy as np

x_vals = np.arange(len(losses)) * (model_cfg.n_ctx * batch_size)

fig = px.line(
    x=x_vals,
    y=losses,
    labels={"x": "Tokens", "y": "Loss"},
    title="Training curve for my tiny demo model"
)

fig.show()
```



```
test_string = "CNN is a"
for i in tqdm.tqdm(range(10)):
    test_tokens = reference_gpt2.to_tokens(test_string).cuda()
    demo_logits = model(test_tokens)
    test_string += reference_gpt2.tokenizer.decode(demo_logits[-1, -1].argmax())
```

Show hidden output

```
print(test_string)
```

```
CNN is a the other of the other of the other of the
```