



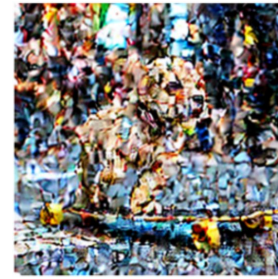
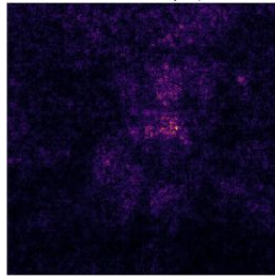
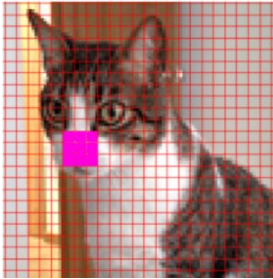
DeepLearning.AI

Transformers

Specialized Approaches to
Natural Language Processing in Pytorch

Module 2

Specialized Approaches to **Vision**



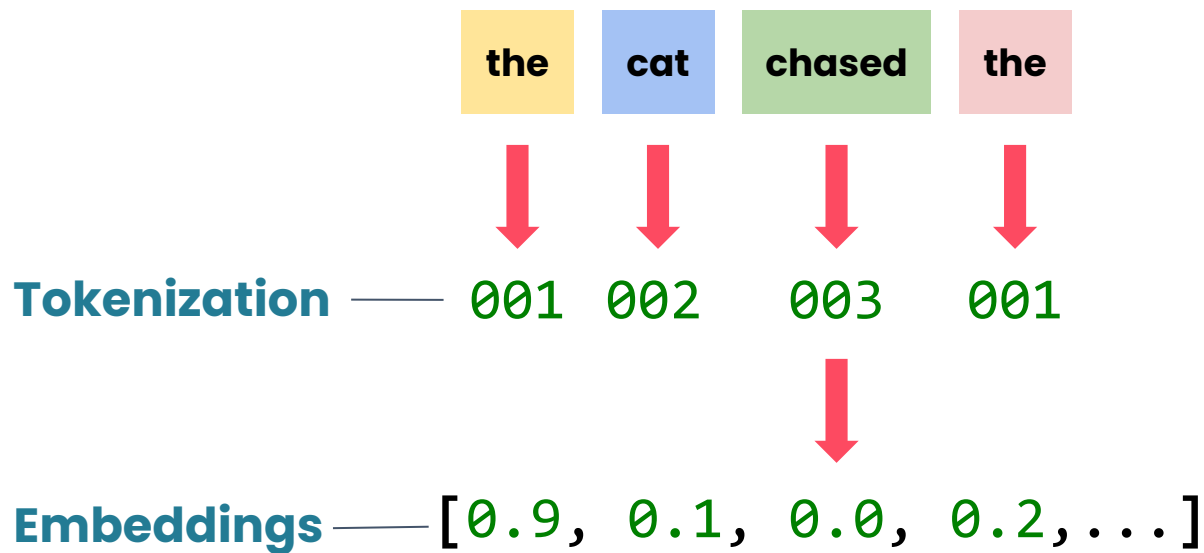
Module 3

Specialized Approaches to **Language**

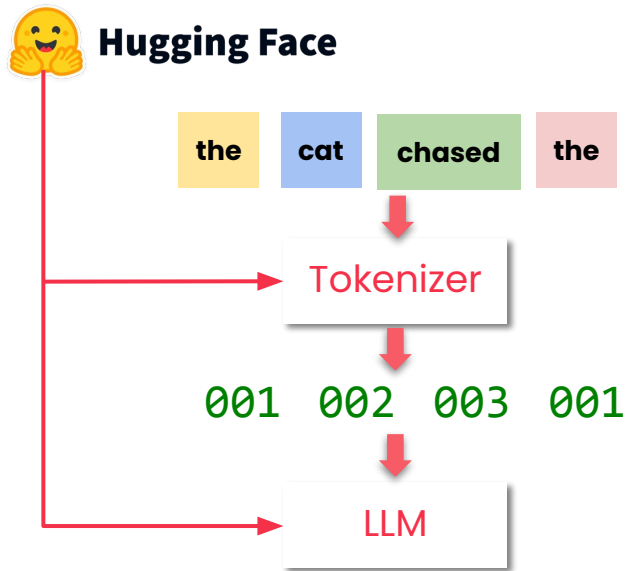
Transformers = Backbone of modern NLP



Tokenization and Embeddings



Pre-Trained Models



```
from transformers import AutoTokenizer,
AutoModelForCausalLM

# Load a pretrained GPT-2 model and its tokenizer
tokenizer = AutoTokenizer.from_pretrained("gpt2")
model = AutoModelForCausalLM.from_pretrained("gpt2")

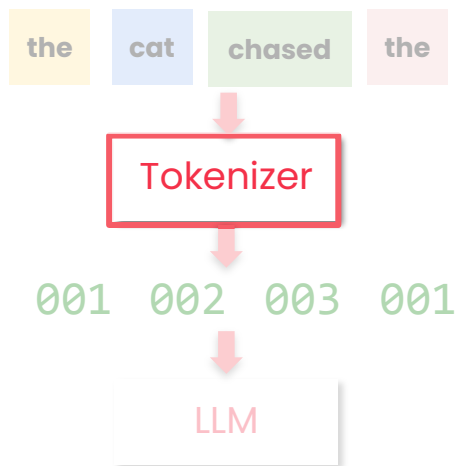
# Encode a short prompt
inputs = tokenizer("The cat chased the",
                  return_tensors="pt")

# Generate up to 10 tokens
outputs = model.generate(**inputs, max_length=10)
```

Pre-Trained Models



Hugging Face



```
from transformers import AutoTokenizer,
AutoModelForCausalLM

# Load a pretrained GPT-2 model and its tokenizer
tokenizer = AutoTokenizer.from_pretrained("gpt2")
model = AutoModelForCausalLM.from_pretrained("gpt2")

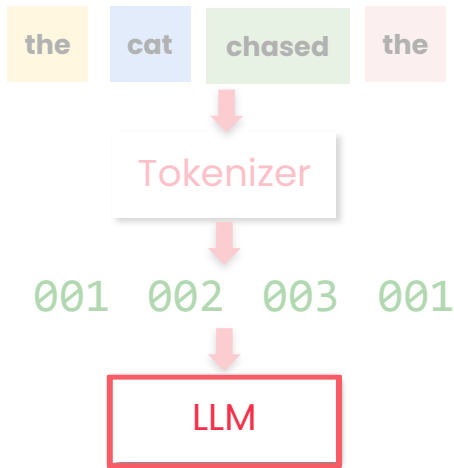
# Encode a short prompt
inputs = tokenizer("The cat chased the",
                  return_tensors="pt")

# Generate up to 10 tokens
outputs = model.generate(**inputs, max_length=10)
```

Pre-Trained Models



Hugging Face



```
from transformers import AutoTokenizer,  
AutoModelForCausalLM
```

```
# Load a pretrained GPT-2 model and its tokenizer  
tokenizer = AutoTokenizer.from_pretrained("gpt2")  
model = AutoModelForCausalLM.from_pretrained("gpt2")
```

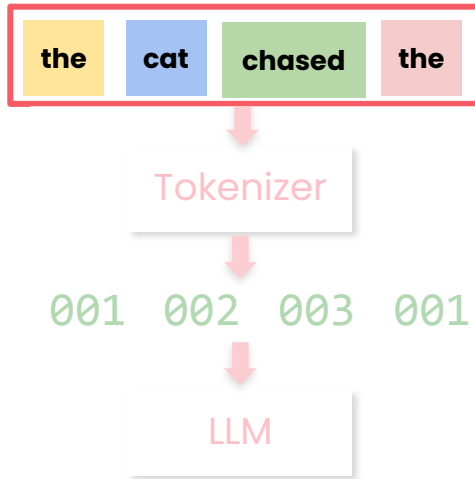
```
# Encode a short prompt  
inputs = tokenizer("The cat chased the",  
                  return_tensors="pt")
```

```
# Generate up to 10 tokens  
outputs = model.generate(**inputs, max_length=10)
```

Pre-Trained Models



Hugging Face



```
from transformers import AutoTokenizer,
AutoModelForCausalLM

# Load a pretrained GPT-2 model and its tokenizer
tokenizer = AutoTokenizer.from_pretrained("gpt2")
model = AutoModelForCausalLM.from_pretrained("gpt2")

# Encode a short prompt
inputs = tokenizer("The cat chased the",
                  return_tensors="pt")

# Generate up to 10 tokens
outputs = model.generate(**inputs, max_length=10)
```


Pre-Trained Models



Hugging Face

the cat chased the

Tokenizer

001 002 003 001

LLM

"The cat chased the mouse ..."

```
from transformers import AutoTokenizer,
AutoModelForCausalLM

# Load a pretrained GPT-2 model and its tokenizer
tokenizer = AutoTokenizer.from_pretrained("gpt2")
model = AutoModelForCausalLM.from_pretrained("gpt2")

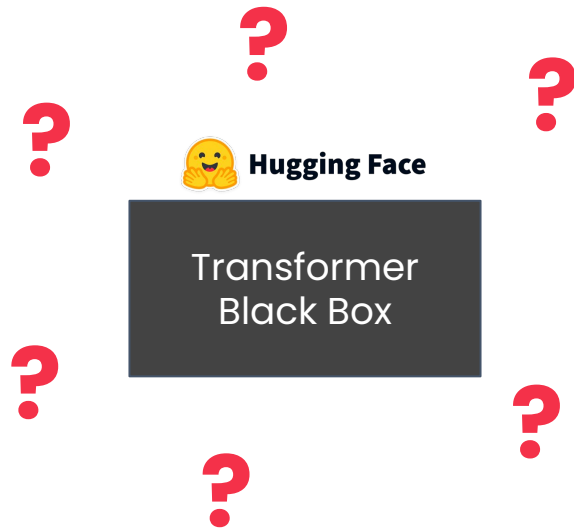
# Encode a short prompt
inputs = tokenizer("The cat chased the",
                  return_tensors="pt")

# Generate up to 10 tokens
outputs = model.generate(**inputs, max_length=10)
```

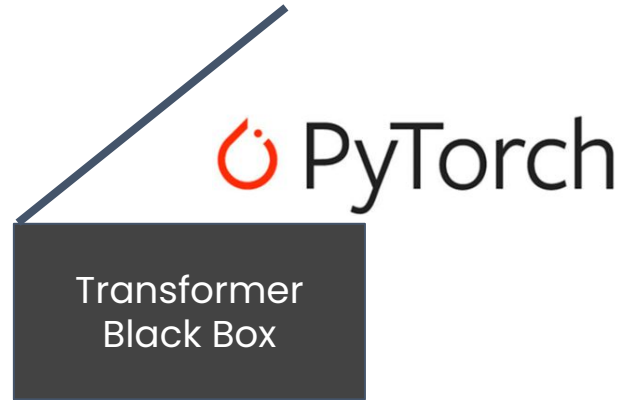
Transformers

You can't...

- Experiment
- Adapt
- Debug
- Explain



Transformers



Early Language Models

RNNs & LSTMs

The cat chased the mouse quickly across the yard

Transformers



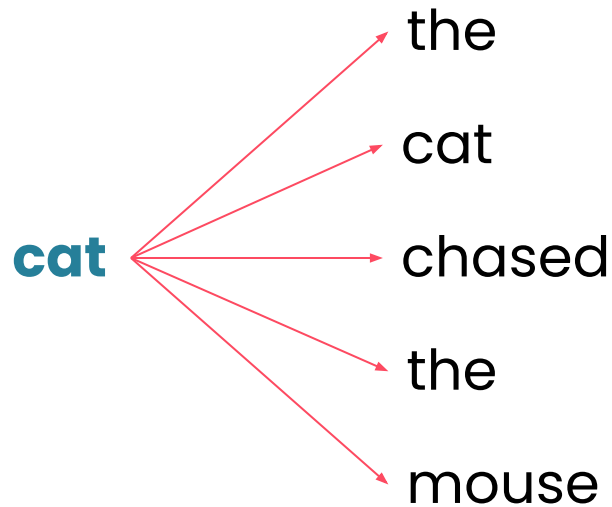
The cat chased the mouse

Attention

“the cat chased the mouse”

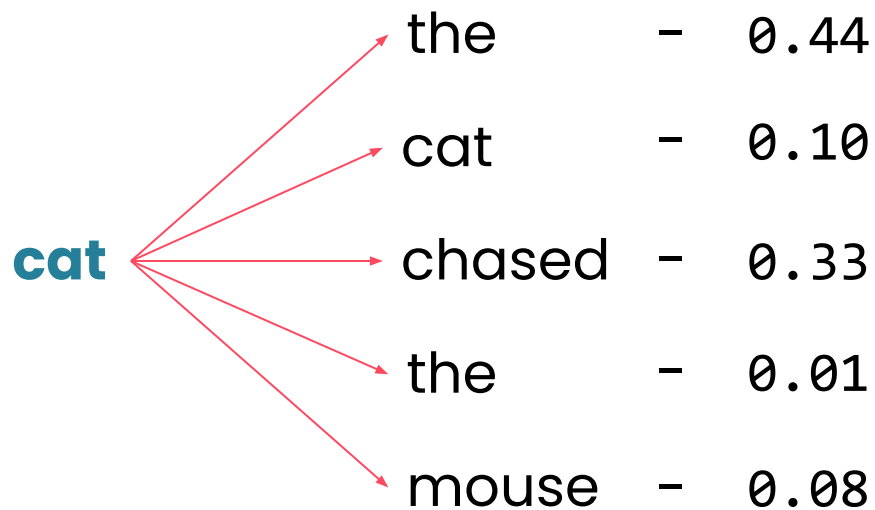
Attention

"the **cat** chased the mouse"



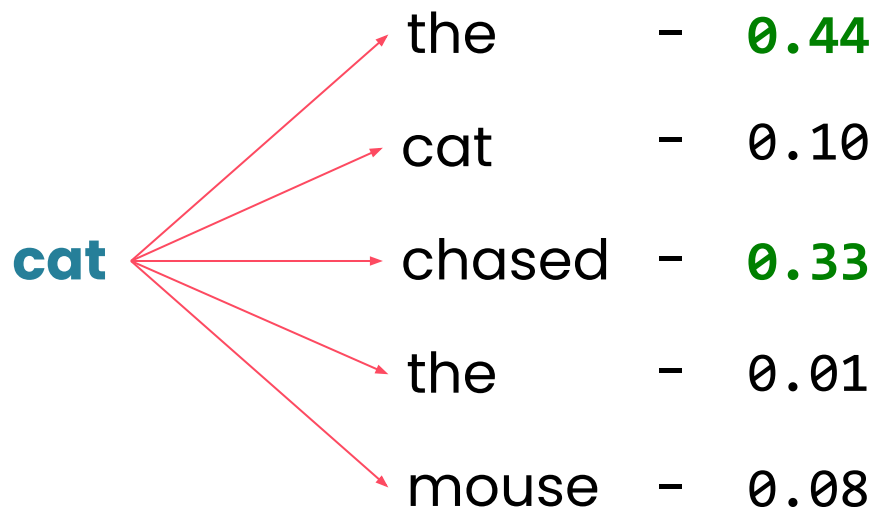
Attention

"the **cat** chased the mouse"



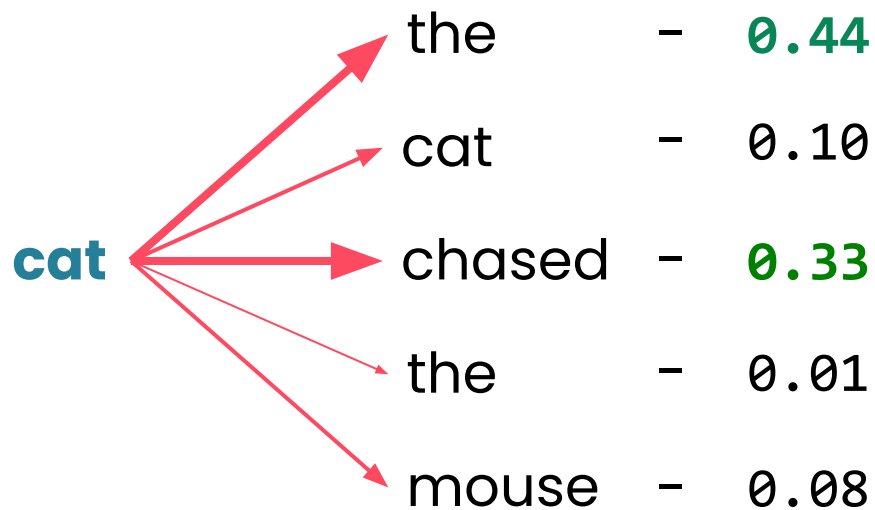
Attention

"the **cat** chased the mouse"



Attention

"the **cat** chased the mouse"



Attention

The animal didn't cross the street because it was too wide

Attention

The animal didn't cross the street because **it** was too wide

Attention

The animal didn't cross the street because it was too wide

Attention

The animal didn't cross the street because it was too wide



Attention

The animal didn't cross the street because it was too wide

Attention

The animal didn't cross the street because it was too wide

adjectives

A red curved line connects the word 'street' to the word 'wide', illustrating an attention mechanism. The word 'adjectives' is written in red above the line.

Attention

Subject / verb

The animal cross the street because it was too wide

A red curved line (arc) connects the word 'animal' in a red box to the word 'cross' in a red box, illustrating the subject-verb relationship.

Attention Heads

Attention Head 1

Attention Head 2

Attention Head 3

Attention Head 4

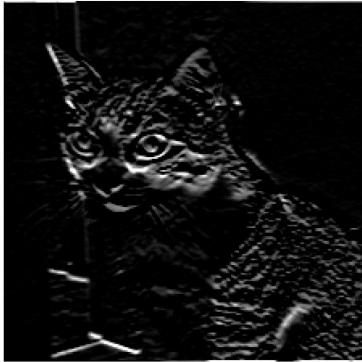
Attention Head 5

...

**Each head learns
different patterns**

CNN Filters

Horizontal Edge



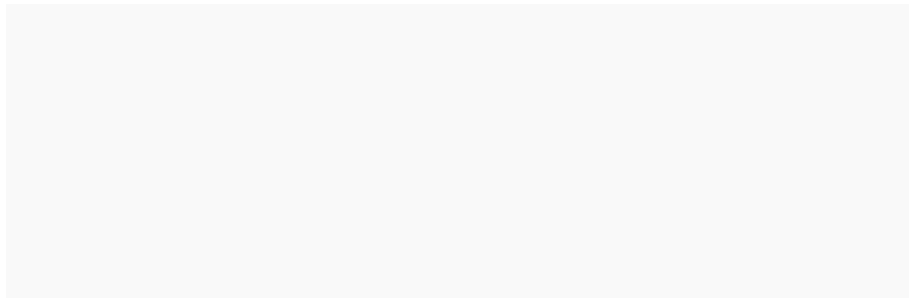
Vertical Edge



Ridge



Attention Heads



Attention Head 1

Attention Head 2

Attention Head 3

Attention Head 4

Attention Head 5

• • •

Attention Heads

```
import torch
import torch.nn as nn

mha = nn.MultiheadAttention(. . .)
```

Attention Head 1

Attention Head 2

Attention Head 3

Attention Head 4

Attention Head 5

• • •

Position



Position: RNNs

The → cat → chased → the → dog

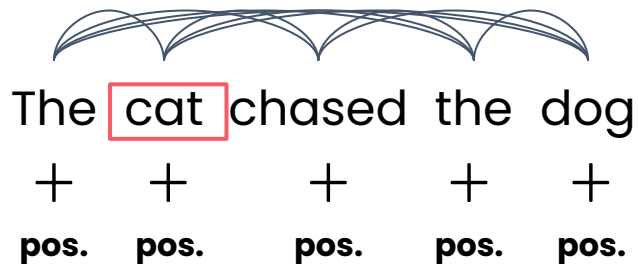
Position: Transformers



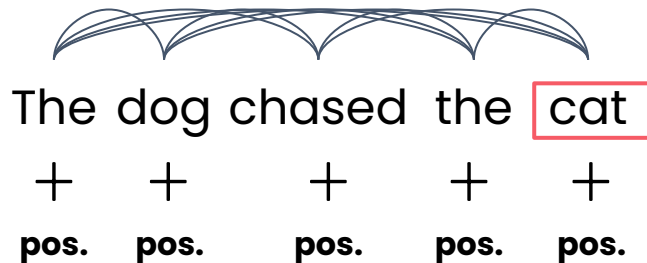
The cat chased the dog



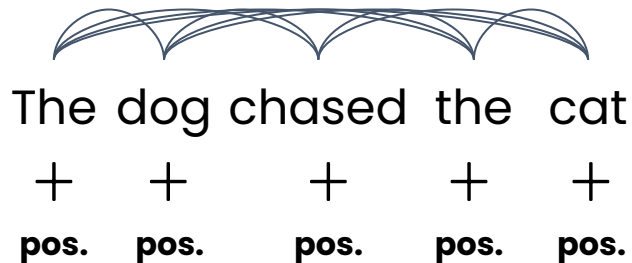
Position: Transformers



Position: Transformers



Position: Transformers



Positional info gives a sense of order

Transformer Architectures

- **Encoders**
- **Decoders**
- **Encoder-Decoders**

Encoders

- **Understanding**
- **Classification**
- **Named Entity**
- **BERT**



The cat chased the dog

Decoders



The cat chased the dog

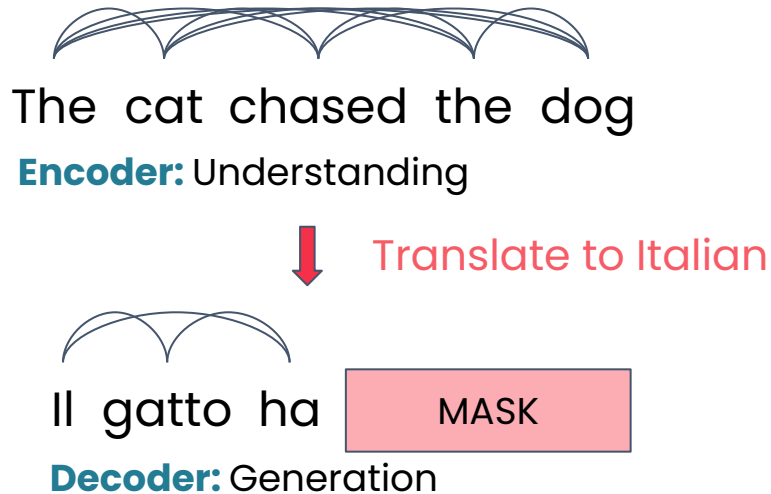
Decoders

- **Generation**
- **Chatbots**
- **GPT**



Encoder – Decoders

- **Translation**
- **Summarization**



Transformer Architectures

- **Encoders:** Understanding
- **Decoders:** Generation
- **Encoder-Decoders:** Transformation

Transformer Architectures

- **Encoders:** Understanding
- **Decoders:** Generation
- **Encoder-Decoders:** Transformation

Transformer Architectures

- **Encoders:** Understanding
- **Decoders:** Generation
- **Encoder-Decoders:** Transformation

Transformer Architectures

- **Encoders:** Understanding
- **Decoders:** Generation
- **Encoder-Decoders:** Transformation

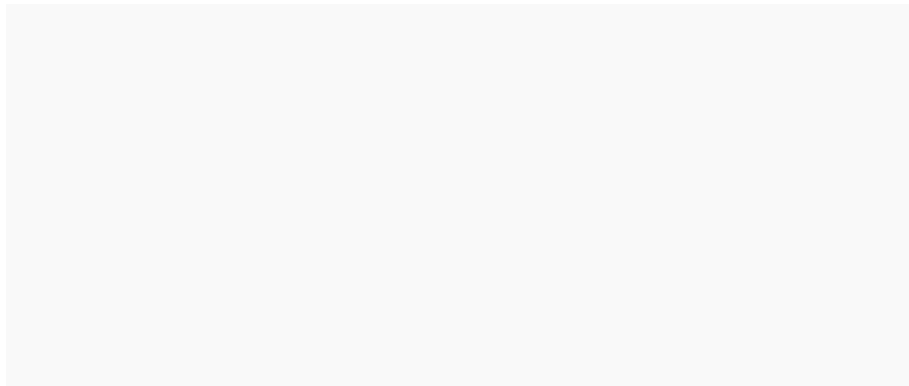


DeepLearning.AI

Attention

Specialized Approaches to
Natural Language Processing in Pytorch

Attention



Attention Head 1

Attention Head 2

Attention Head 3

Attention Head 4

Attention Head 5

...

Attention

```
import torch
import torch.nn as nn

mha = nn.MultiheadAttention(embed_dim=4,
                             num_heads=2,
                             dropout=0.1,
                             batch_first=True)
```

Attention Head 1

Attention Head 2

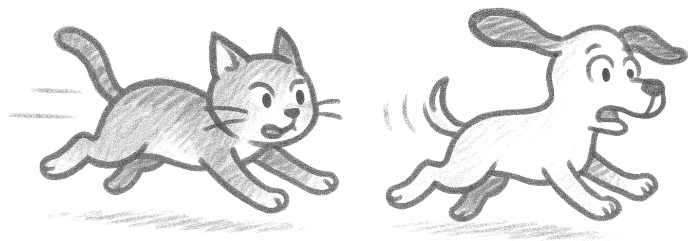
Attention Head 3

Attention Head 4

Attention Head 5

...

Attention



"The cat chased the dog"

Tokens

<s> The cat chased the dog

Tokens

<s> The cat chased the dog

Tokens

<s> The cat chased the dog

Tokens

<s> The cat chased the dog

Tokens

```
token_sequences = [['the', 'cat', 'chased', 'the', 'dog']]
```

Tokens

```
token_sequences = [['the', 'cat', 'chased', 'the', 'dog']]
```

```
token_sequences = torch.tensor([[0, 1, 2, 0, 3]])
```

Tokens

```
token_sequences = [['the', 'cat', 'chased', 'the', 'dog']]  
  
token_sequences = torch.tensor([[0, 1, 2, 0, 3]])  
  
tok_embed = nn.Embedding(num_embeddings=len(vocab), embedding_dim=4)
```

Tokens

```
token_sequences = [['the', 'cat', 'chased', 'the', 'dog']]  
  
token_sequences = torch.tensor([[0, 1, 2, 0, 3]])  
  
tok_embed = nn.Embedding(num_embeddings=len(vocab), embedding_dim=4)
```


Tokens

```
token_sequences = [['the', 'cat', 'chased', 'the', 'dog']]  
  
token_sequences = torch.tensor([[0, 1, 2, 0, 3]])  
  
tok_embed = nn.Embedding(num_embeddings=len(vocab), embedding_dim=4)
```

'cat'

[0.2, 0.8, -0.5, 0.1]

'dog'

[0.3, 0.7, -0.6, 0.2]

'carburetor'

[-0.9, 0.1, 0.8, -0.4]

Token Embeddings

```
torch.tensor([[[ 0.12, -0.55, 0.33, 0.10], # the  
               [-0.44, 0.91, -0.12, -0.77], # cat  
               [ 0.48, 0.02, 0.05, 0.39], # chased  
               [ 0.12, -0.55, 0.33, 0.10], # the  
               [-0.30, 0.14, -0.70, 0.81]]]) # dog
```

Token Embeddings

```
torch.tensor([[[ 0.12, -0.55, 0.33, 0.10], # the
                [-0.44, 0.91, -0.12, -0.77], # cat
                [ 0.48, 0.02, 0.05, 0.39], # chased
                [ 0.12, -0.55, 0.33, 0.10], # the
                [-0.30, 0.14, -0.70, 0.81]]]) # dog
```

Token Embeddings

```
torch.tensor([[[ 0.12, -0.55, 0.33, 0.10], # the  
               [-0.44, 0.91, -0.12, -0.77], # cat  
               [ 0.48, 0.02, 0.05, 0.39], # chased  
               [ 0.12, -0.55, 0.33, 0.10], # the  
               [-0.30, 0.14, -0.70, 0.81]]]) # dog
```

Token Embeddings

```
torch.tensor([[ [ 0.12, -0.55, 0.33, 0.10], # the  
                [-0.44, 0.91, -0.12, -0.77], # cat  
                [ 0.48, 0.02, 0.05, 0.39], # chased  
                [ 0.12, -0.55, 0.33, 0.10], # the  
                [-0.30, 0.14, -0.70, 0.81]]) # dog
```

Which word came first?

Positional Embeddings

```
torch.tensor([[[ 0.12, -0.55, 0.33, 0.10], # the  
                [-0.44, 0.91, -0.12, -0.77], # cat  
                [ 0.48, 0.02, 0.05, 0.39], # chased  
                [ 0.12, -0.55, 0.33, 0.10], # the  
                [-0.30, 0.14, -0.70, 0.81]]]) # dog
```

} 5

Positional Embeddings

```
torch.tensor([[[[ 0.12, -0.55, 0.33, 0.10], # the 0  
                 [-0.44, 0.91, -0.12, -0.77], # cat 1  
                 [ 0.48, 0.02, 0.05, 0.39], # chased 2  
                 [ 0.12, -0.55, 0.33, 0.10], # the 3  
                 [-0.30, 0.14, -0.70, 0.81]]]]) # dog 4
```

Positional Embeddings

```
torch.tensor([[[ 0.12, -0.55,  0.33,  0.10], # the 0  
                [-0.44,  0.91, -0.12, -0.77], # cat 1  
                [ 0.48,  0.02,  0.05,  0.39], # chased 2  
                [ 0.12, -0.55,  0.33,  0.10], # the 3  
                [-0.30,  0.14, -0.70,  0.81]]]) # dog 4
```

```
positions = torch.arange(seq_len, device=word_vecs.device)
```


Positional Embeddings

```
torch.tensor([[[ 0.12, -0.55,  0.33,  0.10],    # the
                [-0.44,  0.91, -0.12, -0.77],    # cat
                [ 0.48,  0.02,  0.05,  0.39],    # chased
                [ 0.12, -0.55,  0.33,  0.10],    # the
                [-0.30,  0.14, -0.70,  0.81]]]) # dog
```

```
positions = torch.arange(seq_len, device=word_vecs.device)
```

Positional Embeddings

```
torch.tensor([[[ 0.12, -0.55,  0.33,  0.10], # the
                [-0.44,  0.91, -0.12, -0.77], # cat
                [ 0.48,  0.02,  0.05,  0.39], # chased
                [ 0.12, -0.55,  0.33,  0.10], # the
                [-0.30,  0.14, -0.70,  0.81]]]) # dog
```

```
positions = torch.arange(seq_len, device=word_vecs.device)
```

Positional Embeddings

```
# In practice setup
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
model = AutoModel.from_pretrained("bert-base-uncased")
```

```
positions = torch.tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255, 256, 257, 258, 259, 260, 261, 262, 263, 264, 265, 266, 267, 268, 269, 270, 271, 272, 273, 274, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290, 291, 292, 293, 294, 295, 296, 297, 298, 299, 300, 301, 302, 303, 304, 305, 306, 307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 318, 319, 320, 321, 322, 323, 324, 325, 326, 327, 328, 329, 330, 331, 332, 333, 334, 335, 336, 337, 338, 339, 340, 341, 342, 343, 344, 345, 346, 347, 348, 349, 350, 351, 352, 353, 354, 355, 356, 357, 358, 359, 360, 361, 362, 363, 364, 365, 366, 367, 368, 369, 370, 371, 372, 373, 374, 375, 376, 377, 378, 379, 380, 381, 382, 383, 384, 385, 386, 387, 388, 389, 390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 401, 402, 403, 404, 405, 406, 407, 408, 409, 410, 411, 412, 413, 414, 415, 416, 417, 418, 419, 420, 421, 422, 423, 424, 425, 426, 427, 428, 429, 430, 431, 432, 433, 434, 435, 436, 437, 438, 439, 440, 441, 442, 443, 444, 445, 446, 447, 448, 449, 450, 451, 452, 453, 454, 455, 456, 457, 458, 459, 460, 461, 462, 463, 464, 465, 466, 467, 468, 469, 470, 471, 472, 473, 474, 475, 476, 477, 478, 479, 480, 481, 482, 483, 484, 485, 486, 487, 488, 489, 490, 491, 492, 493, 494, 495, 496, 497, 498, 499, 500, 501, 502, 503, 504, 505, 506, 507, 508, 509, 510, 511, 512, 513, 514, 515, 516, 517, 518, 519, 520, 521, 522, 523, 524, 525, 526, 527, 528, 529, 530, 531, 532, 533, 534, 535, 536, 537, 538, 539, 540, 541, 542, 543, 544, 545, 546, 547, 548, 549, 550, 551, 552, 553, 554, 555, 556, 557, 558, 559, 560, 561, 562, 563, 564, 565, 566, 567, 568, 569, 570, 571, 572, 573, 574, 575, 576, 577, 578, 579, 580, 581, 582, 583, 584, 585, 586, 587, 588, 589, 590, 591, 592, 593, 594, 595, 596, 597, 598, 599, 600, 601, 602, 603, 604, 605, 606, 607, 608, 609, 610, 611, 612, 613, 614, 615, 616, 617, 618, 619, 620, 621, 622, 623, 624, 625, 626, 627, 628, 629, 630, 631, 632, 633, 634, 635, 636, 637, 638, 639, 640, 641, 642, 643, 644, 645, 646, 647, 648, 649, 650, 651, 652, 653, 654, 655, 656, 657, 658, 659, 660, 661, 662, 663, 664, 665, 666, 667, 668, 669, 670, 671, 672, 673, 674, 675, 676, 677, 678, 679, 680, 681, 682, 683, 684, 685, 686, 687, 688, 689, 690, 691, 692, 693, 694, 695, 696, 697, 698, 699, 700, 701, 702, 703, 704, 705, 706, 707, 708, 709, 710, 711, 712, 713, 714, 715, 716, 717, 718, 719, 720, 721, 722, 723, 724, 725, 726, 727, 728, 729, 730, 731, 732, 733, 734, 735, 736, 737, 738, 739, 740, 741, 742, 743, 744, 745, 746, 747, 748, 749, 750, 751, 752, 753, 754, 755, 756, 757, 758, 759, 760, 761, 762, 763, 764, 765, 766, 767, 768, 769, 770, 771, 772, 773, 774, 775, 776, 777, 778, 779, 780, 781, 782, 783, 784, 785, 786, 787, 788, 789, 790, 791, 792, 793, 794, 795, 796, 797, 798, 799, 800, 801, 802, 803, 804, 805, 806, 807, 808, 809, 810, 811, 812, 813, 814, 815, 816, 817, 818, 819, 820, 821, 822, 823, 824, 825, 826, 827, 828, 829, 830, 831, 832, 833, 834, 835, 836, 837, 838, 839,
```

Positional Embeddings

```
torch.tensor([[ 0.12, -0.55,  0.33,  0.10], # the  
              [-0.44,  0.91, -0.12, -0.77], # cat  
              [ 0.12, -0.55,  0.33,  0.10],  
              [-0.44,  0.91, -0.12, -0.77],
```

```
# In practice setup  
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")  
model = AutoModel.from_pretrained("bert-base-uncased")
```

```
positions = torch.tensor([0, 1, 2, 3])  
  
# Tokenize with max_length=512  
inputs = tokenizer(  
    "Your text here",  
    max_length=512,  
    truncation=True,  
    padding="max_length",  
    return_tensors="pt"  
)
```

Positional Embeddings

```
torch.tensor([[[ 0.12, -0.55, 0.33, 0.10], # the  
               [-0.44, 0.91, -0.12, -0.77], # cat  
               [ 0.48, 0.02, 0.05, 0.39], # chased  
               [ 0.12, -0.55, 0.33, 0.10], # the  
               [-0.30, 0.14, -0.70, 0.81]]]) # dog
```

```
positions = torch.arange(seq_len, device=word_vecs.device)
```

```
#positions  
tensor([0, 1, 2, 3, 4])
```

Positional Embeddings

```
torch.tensor([[[ 0.12, -0.55, 0.33, 0.10], # the  
               [-0.44, 0.91, -0.12, -0.77], # cat  
               [ 0.48, 0.02, 0.05, 0.39], # chased  
               [ 0.12, -0.55, 0.33, 0.10], # the  
               [-0.30, 0.14, -0.70, 0.81]]]) # dog
```

```
positions = torch.arange(seq_len, device=word_vecs.device)  
positions = positions.unsqueeze(0).expand(batch_size, seq_len)
```

Positional Embeddings

```
torch.tensor([[[ 0.12, -0.55,  0.33,  0.10],    # the  
                [-0.44,  0.91, -0.12, -0.77],    # cat  
                [ 0.48,  0.02,  0.05,  0.39],    # chased  
                [ 0.12, -0.55,  0.33,  0.10],    # the  
                [-0.30,  0.14, -0.70,  0.81]]]) # dog
```

```
positions = torch.arange(seq_len, device=word_vecs.device)
```

```
positions = positions.unsqueeze(0).expand(batch_size, seq_len)
```

Positional Embeddings

```
torch.tensor([[[ 0.12, -0.55,  0.33,  0.10],   # the  
               [-0.44,  0.91, -0.12, -0.77],   # cat  
               [ 0.48,  0.02,  0.05,  0.39],   # chased  
               [ 0.12, -0.55,  0.33,  0.10],   # the  
               [-0.30,  0.14, -0.70,  0.81]]]) # dog
```

```
positions = torch.arange(seq_len, device=word_vecs.device)  
positions = positions.unsqueeze(0).expand(batch_size, seq_len)
```


Positional Embeddings

```
torch.tensor([[[ 0.12, -0.55, 0.33, 0.10], # the
                [-0.44, 0.91, -0.12, -0.77], # cat
                [ 0.48, 0.02, 0.05, 0.39], # chased
                [ 0.12, -0.55, 0.33, 0.10], # the
                [-0.30, 0.14, -0.70, 0.81]]]) # dog
```

```
positions = torch.arange(seq_len, device=word_vecs.device)
positions = positions.unsqueeze(0).expand(batch_size, seq_len)
```

Positional Embeddings

```
torch.tensor([[[ 0.12, -0.55,  0.33,  0.10],   # the
                [-0.44,  0.91, -0.12, -0.77],   # cat
                [ 0.48,  0.02,  0.05,  0.39],   # chased
                [ 0.12, -0.55,  0.33,  0.10],   # the
                [-0.30,  0.14, -0.70,  0.81]]]) # dog
```

```
positions = torch.arange(seq_len, device=word_vecs.device)
positions = positions.unsqueeze(0).expand(batch_size, seq_len)
```

```
pos_embed = nn.Embedding(num_embeddings=seq_len, embedding_dim=emb_dim)
```

Token and Positional Embeddings

```
pos_vecs = pos_embed(positions)
word_vecs = tok_embed(token_ids)
```

Token and Positional Embeddings

```
pos_vecs = pos_embed(positions)
word_vecs = tok_embed(token_ids)
```

```
# add them together
```

```
input_vecs = word_vecs + pos_vecs
```

meaning position

Attention



The cat chased the dog

Pay attention to what?

... and how much attention?

Attention Heads

Attention Head 1

Attention Head 2

Attention Head 3

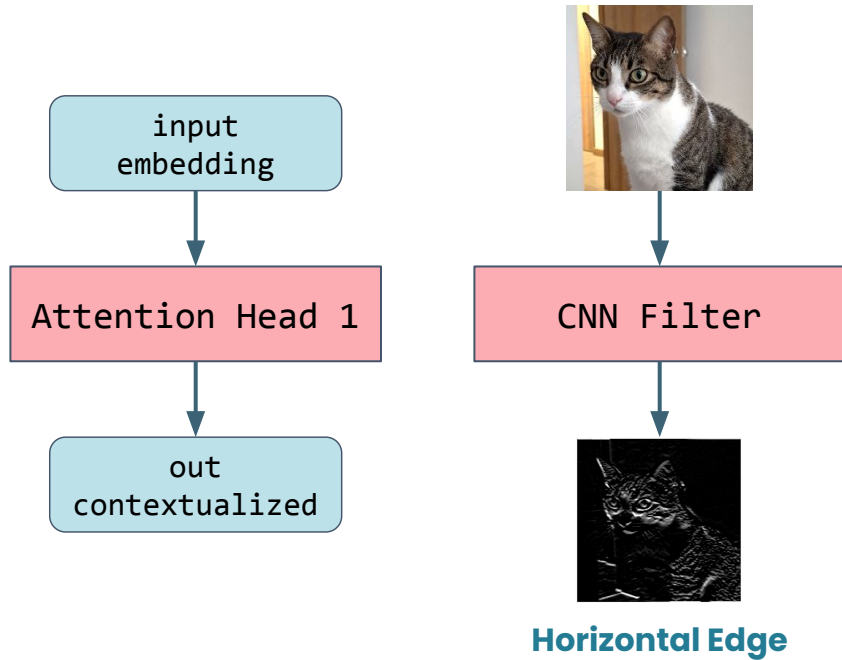
Attention Head 4

Attention Head 5

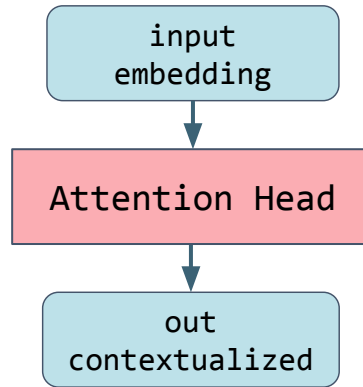
• • •

**Each head learns
different patterns**

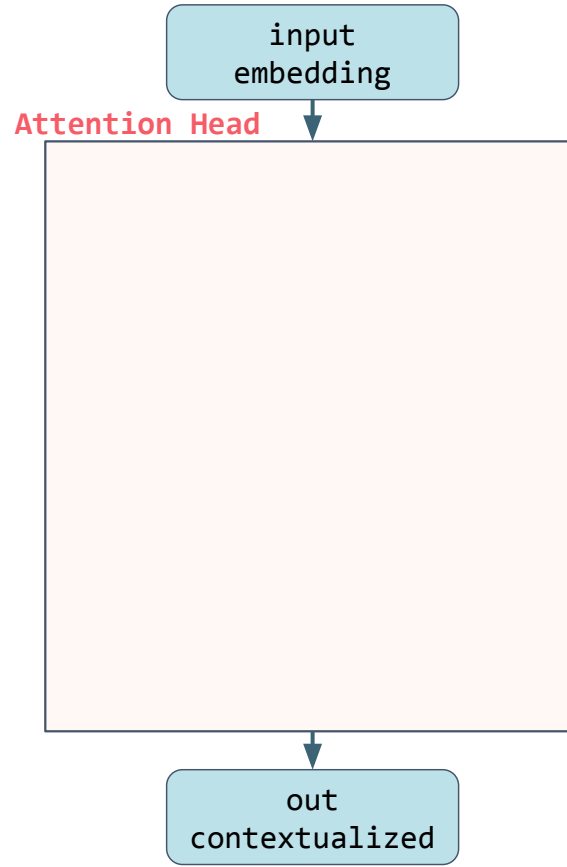
Attention Heads



Attention Heads

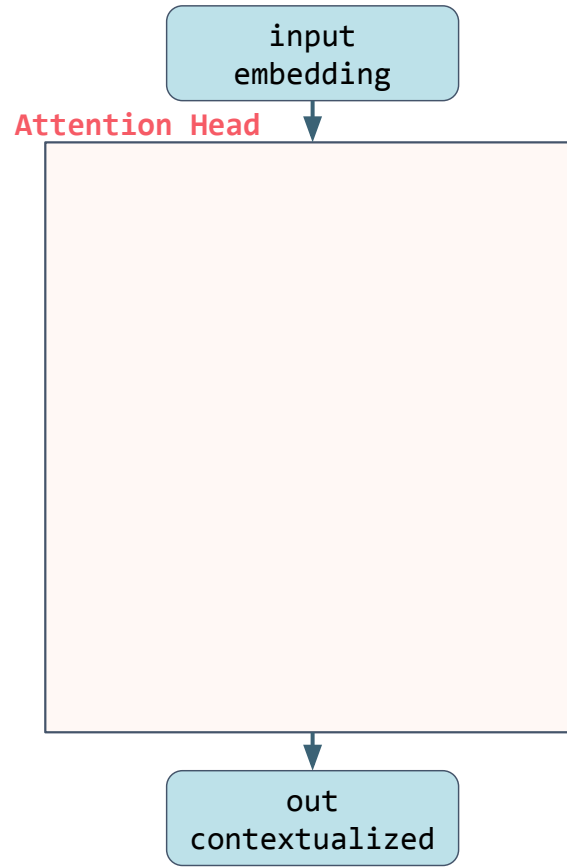


Attention Heads



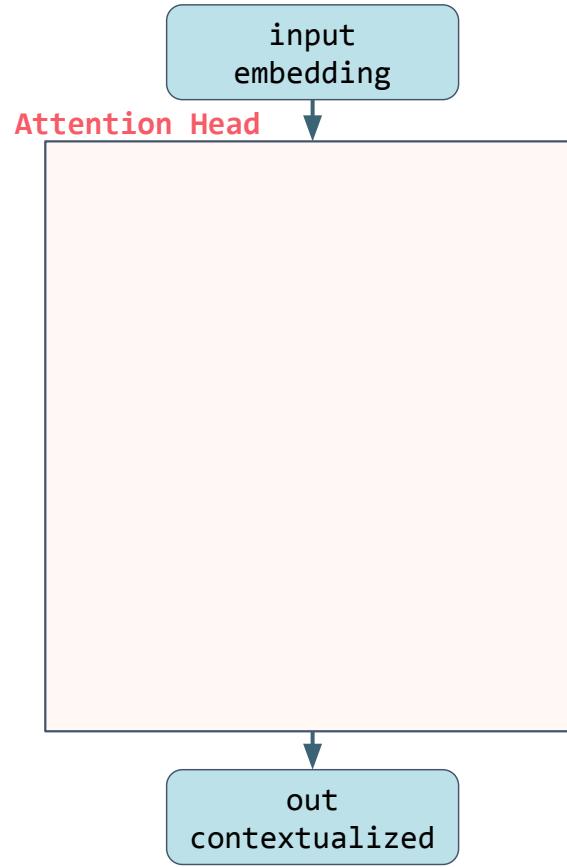
Attention Heads

```
to_k = nn.Linear(4, 4)
```



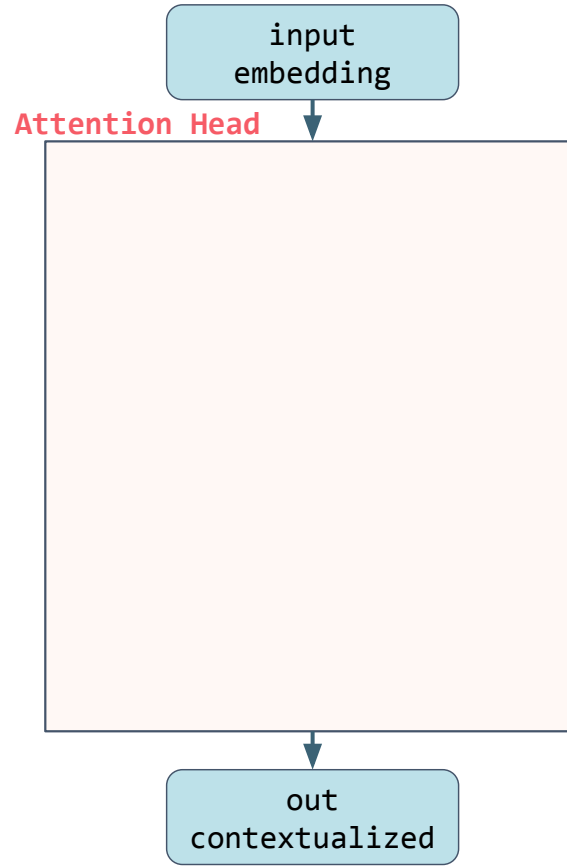
Attention Heads

```
to_k = nn.Linear(4, 4)
```



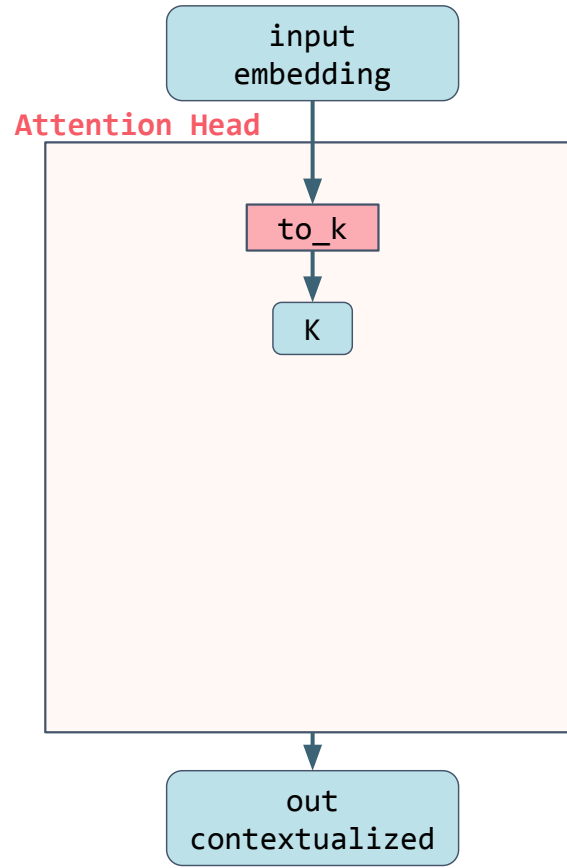
Attention Heads

```
to_k = nn.Linear(4, 4)  
K = to_k(input_embedding)
```



Attention Heads

```
to_k = nn.Linear(4, 4)
K = to_k(input_embedding)
```



Key

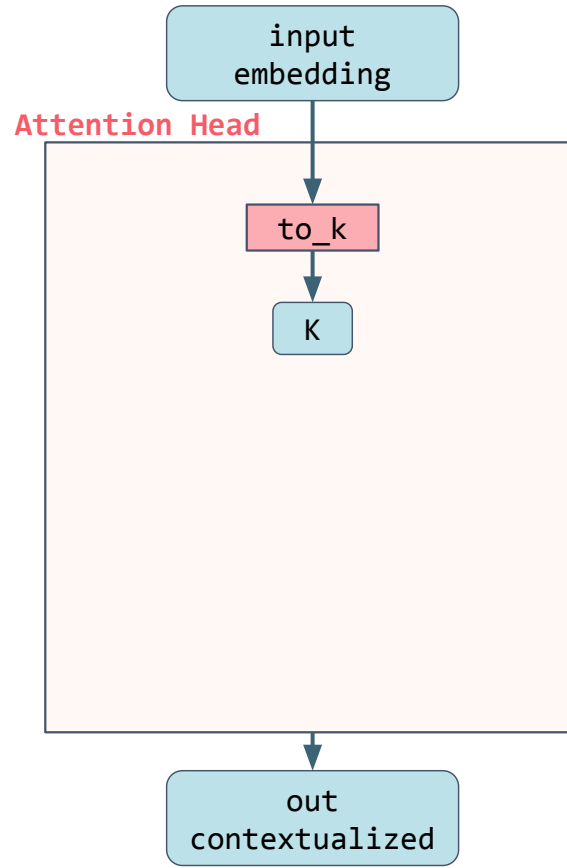
What I look like in this space

*But who should I
pay attention to?*

What am I looking for?

Attention Heads

```
to_k = nn.Linear(4, 4)
K = to_k(input_embedding)
```



Query

What I'm looking for

Key

What I look like in this space

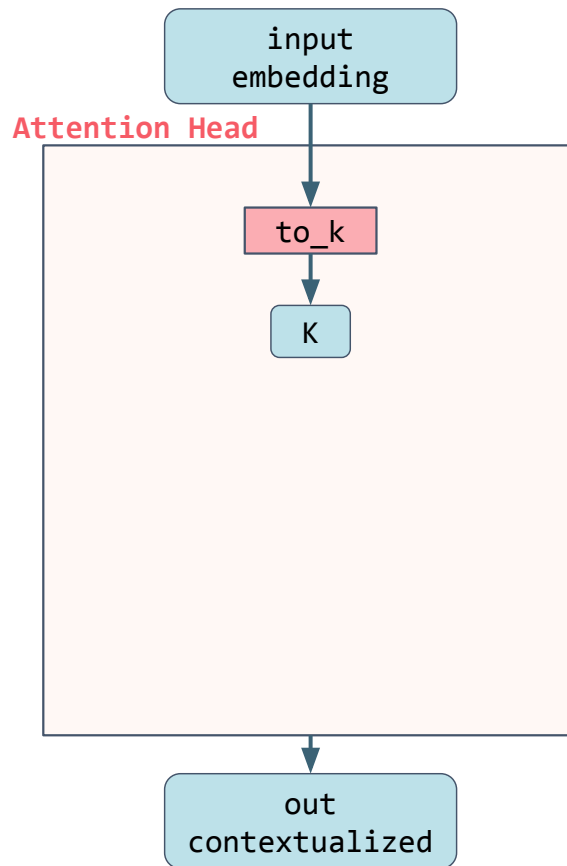
Attention Heads

```
to_q = nn.Linear(4, 4)
```

```
to_k = nn.Linear(4, 4)
```

```
Q = to_q(input_embedding)
```

```
K = to_k(input_embedding)
```



Query

What I'm looking for

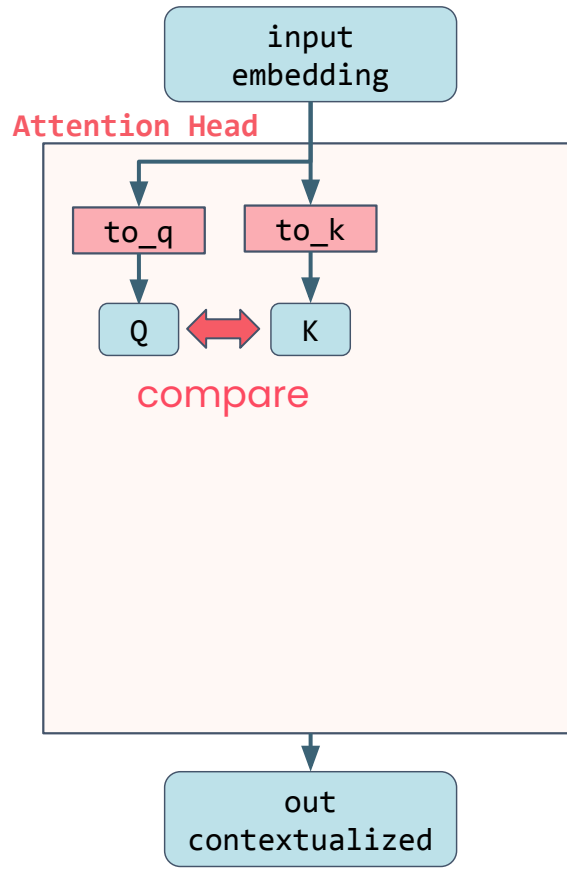
Key

What I look like in this space

Attention Heads

```
to_q = nn.Linear(4, 4)
to_k = nn.Linear(4, 4)

Q = to_q(input_embedding)
K = to_k(input_embedding)
```



Query

What I'm looking for

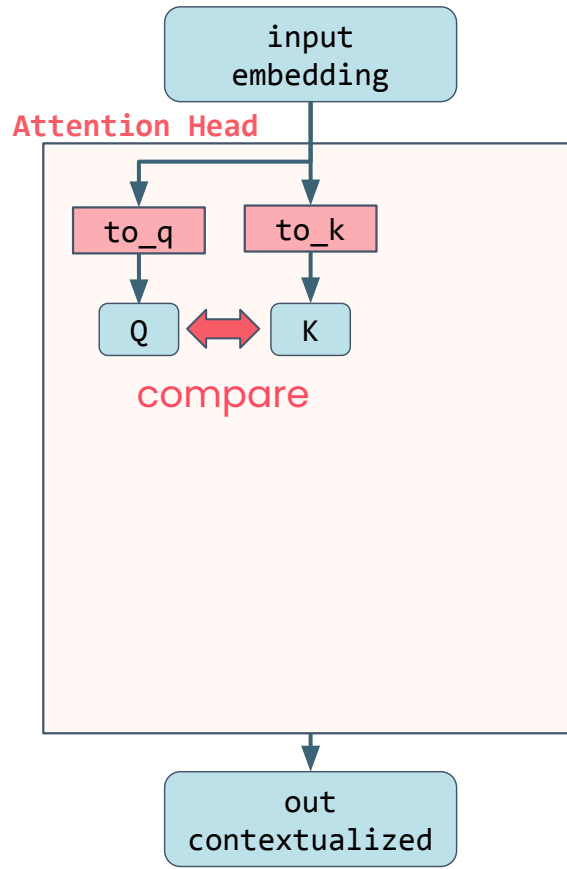
Key

What I look like in this space

Attention Heads

```
to_q = nn.Linear(4, 4)
to_k = nn.Linear(4, 4)

Q = to_q(input_embedding)
K = to_k(input_embedding)
```



Query

What I'm looking for

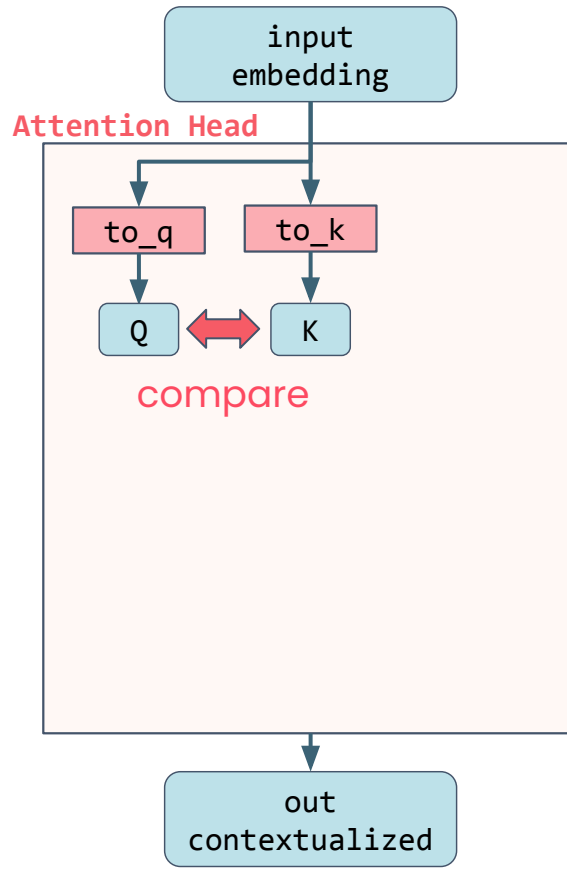
Key

What I look like in this space

Attention Heads

```
to_q = nn.Linear(4, 4)
to_k = nn.Linear(4, 4)

Q = to_q(input_embedding)
K = to_k(input_embedding)
```



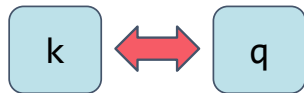
Query

What I'm looking for

Key

What I look like in this space

Calculating Attention Weights



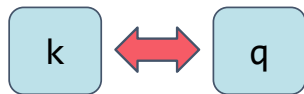
compare

scaled dot product

“the **cat** chased the dog”

“**cat**” Query = [0.2, -0.1, 0.5, 0.3] what ‘cat’ is looking for

Calculating Attention Weights



compare

scaled dot product

"the cat chased the dog"

"cat" Query = [0.2, -0.1, 0.5, 0.3] what 'cat' is looking for

* * * *

"the" Key = [0.1, 0.5, 0.4, 0.2] what 'the' looks like

↓ ↓ ↓ ↓

0.02 -0.05 0.2 0.06

Calculating Attention Weights



compare

scaled dot product

“the cat chased the dog”

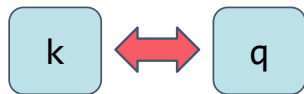
“cat” Query = [0.2, -0.1, 0.5, 0.3] what 'cat' is looking for

* * * *

“the” Key = [0.1, 0.5, 0.4, 0.2] what 'the' looks like

sum([0.02, -0.05, 0.2, 0.06])

Calculating Attention Weights



compare

scaled dot product

"**the cat** chased the dog"

"**cat**" Query = [0.2, -0.1, 0.5, 0.3] what 'cat' is looking for

* * * *

"**the**" Key = [0.1, 0.5, 0.4, 0.2] what 'the' looks like

0.23 attention score

Calculating Attention Weights



compare

scaled dot product

```
K_t = K.transpose(-2, 1)
scale = math.sqrt(Q.size(-1))

# KQ compare: scaled dot product
scores = torch.matmul(Q, K_t)/scale
```

0.23 attention score

Calculating Attention Weights



compare

scaled dot product

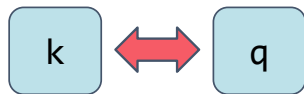
```
K_t = K.transpose(-2, 1)
scale = math.sqrt(Q.size(-1))

# KQ compare: scaled dot product
scores = torch.matmul(Q, K_t)/scale
```

0.23 attention score

scale

Calculating Attention Weights



compare

scaled dot product

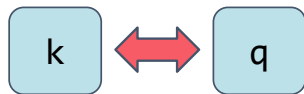
```
K_t = K.transpose(-2, 1)
scale = math.sqrt(Q.size(-1))

# KQ compare: scaled dot product
scores = torch.matmul(Q, K_t)/scale
```

Attention Scores

	the	cat	chased	the	dog
the	1.21	-0.259	-0.762	-0.995	0.054
cat	-0.216	0.604	-1.650	-1.061	1.876
chased	1.462	-0.410	-1.009	-0.990	0.822
the	1.506	-1.180	0.659	-1.810	0.521
dog	0.995	0.521	0.627	-0.511	0.315

Calculating Attention Weights



compare

scaled dot product

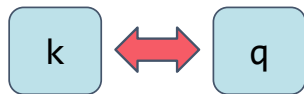
```
K_t = K.transpose(-2, 1)
scale = math.sqrt(Q.size(-1))

# KQ compare: scaled dot product
scores = torch.matmul(Q, K_t)/scale
```

Attention Scores

	the	cat	chased	the	dog
the	1.21	-0.259	-0.762	-0.995	0.054
cat	-0.216	0.604	-1.650	-1.061	1.876
chased	1.462	-0.410	-1.009	-0.990	0.822
the	1.506	-1.180	0.659	-1.810	0.521
dog	0.995	0.521	0.627	-0.511	0.315

Calculating Attention Weights



compare

scaled dot product

```
K_t = K.transpose(-2, 1)
scale = math.sqrt(Q.size(-1))

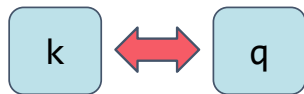
# KQ compare: scaled dot product
scores = torch.matmul(Q, K_t)/scale
```

```
attn = scores.softmax(dim=-1)
```

Attention Scores

	the	cat	chased	the	dog
the	1.21	-0.259	-0.762	-0.995	0.054
cat	-0.216	0.604	-1.650	-1.061	1.876
chased	1.462	-0.410	-1.009	-0.990	0.822
the	1.506	-1.180	0.659	-1.810	0.521
dog	0.995	0.521	0.627	-0.511	0.315

Calculating Attention Weights



compare

scaled dot product

```
K_t = K.transpose(-2, 1)
scale = math.sqrt(Q.size(-1))

# KQ compare: scaled dot product
scores = torch.matmul(Q, K_t)/scale
```

```
attn = scores.softmax(dim=-1)
```

Attention Weights

	the	cat	chased	the	dog	
the	.557	.129	.077	.061	.175	Sum=1
cat	.083	.189	.020	.036	.673	Sum=1
chased	.540	.083	.046	.047	.285	Sum=1
the	.524	.036	.225	.019	.196	Sum=1
dog	.329	.205	.227	.073	.167	Sum=1

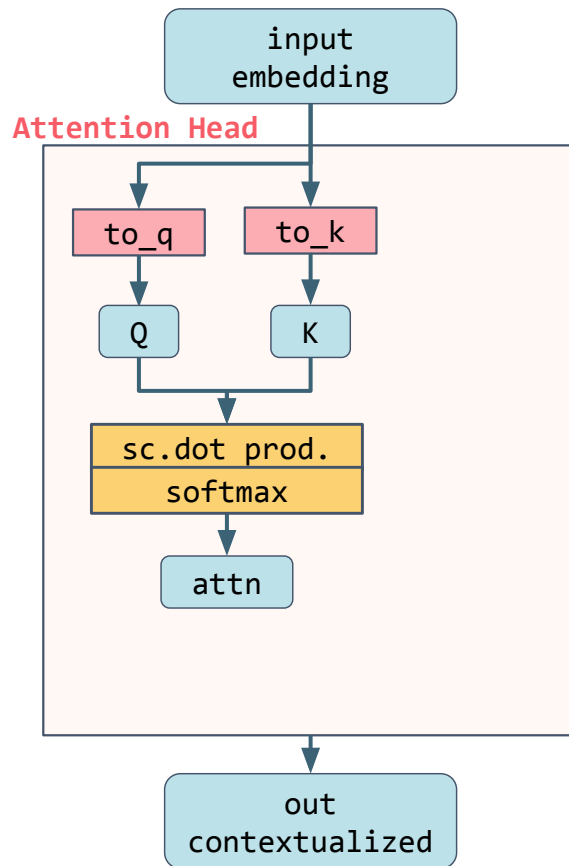
Attention Heads

```
to_q = nn.Linear(4, 4)
to_k = nn.Linear(4, 4)
```

```
Q = to_q(input_embedding)
K = to_k(input_embedding)
```

```
K_t = K.transpose(-2, 1)
scale = math.sqrt(Q.size(-1))
# KQ compare: scaled dot product
scores = torch.matmul(Q, K_t)/scale
```

```
# attn weights: softmax on scaled dp
attn = F.softmax(scores, dim=-1)
```



Query

What I'm looking for

Key

What I look like in this space

*But what do
tokens contribute?*

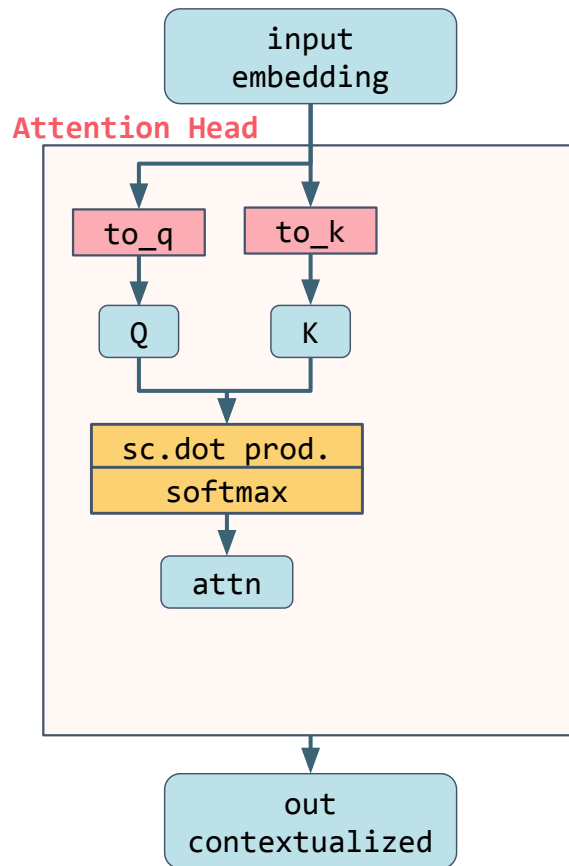
Attention Heads

```
to_q = nn.Linear(4, 4)
to_k = nn.Linear(4, 4)
```

```
Q = to_q(input_embedding)
K = to_k(input_embedding)
```

```
K_t = K.transpose(-2, 1)
scale = math.sqrt(Q.size(-1))
# KQ compare: scaled dot product
scores = torch.matmul(Q, K_t)/scale
```

```
# attn weights: softmax on scaled dp
attn = F.softmax(scores, dim=-1)
```



Query

What I'm looking for

Key

What I look like in this space

Value

If you attend to me,
here's what I'll give you

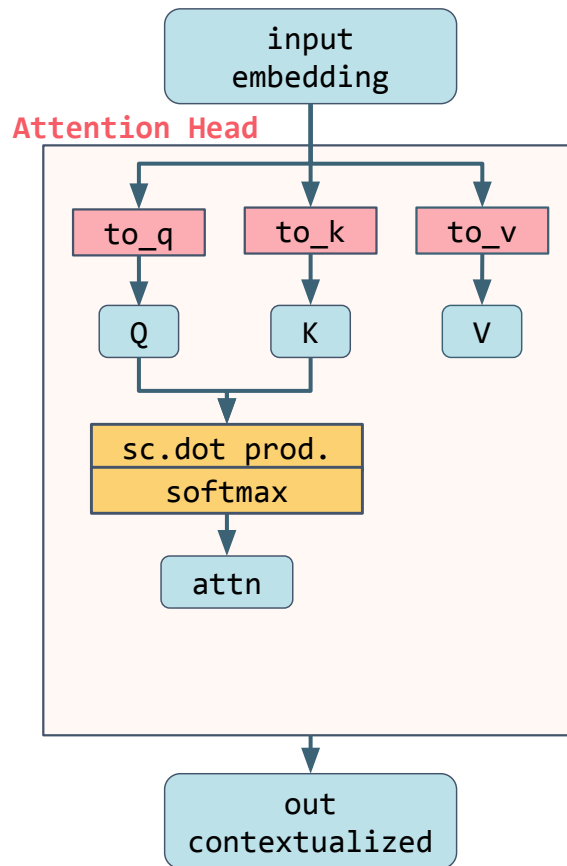
Attention Heads

```
to_q = nn.Linear(4, 4)
to_k = nn.Linear(4, 4)
to_v = nn.Linear(4, 4)
```

```
Q = to_q(input_embedding)
K = to_k(input_embedding)
V = to_v(input_embedding)
```

```
K_t = K.transpose(-2, 1)
scale = math.sqrt(Q.size(-1))
# KQ compare: scaled dot product
scores = torch.matmul(Q, K_t)/scale

# attn weights: softmax on scaled dp
attn = F.softmax(scores, dim=-1)
```



Query

What I'm looking for

Key

What I look like in this space

Value

If you attend to me,
here's what I'll give you

Attention Output

The **cat** chased the dog



"cat" **attn.** to others

the	.083
cat	.189
chased	.020
the	.036
dog	.673

Value (v) vector of other tokens

"the" **value**

"cat" **value**

"chased" **value**

"the" **value**

"dog" **value**

Attention Output

The **cat** chased the dog



"Cat" **attn.** * **Value** vector of other tokens

.083 * "the" **value**

.189 * "cat" **value**

.020 * "chased" **value**

.036 * "the" **value**

.673 * "dog" **value**

SUM

→ **"cat" contextualized output**
[0.27, -0.35, 0.80, -0.58]

```
# contextualized output: dot product  
out = torch.matmul(attn, V)
```

Attention Heads

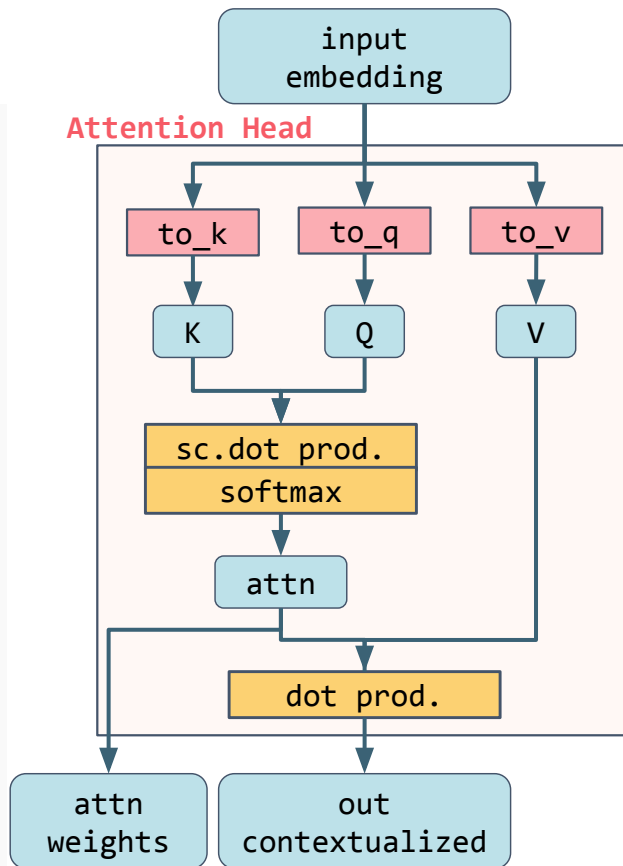
```
to_k = nn.Linear(4, 4)
to_q = nn.Linear(4, 4)
to_v = nn.Linear(4, 4)
```

```
K = to_k(input_embedding)
Q = to_q(input_embedding)
V = to_v(input_embedding)
```

```
K_t = K.transpose(-2, 1)
scale = math.sqrt(Q.size(-1))
# KQ compare: scaled dot product
scores = torch.matmul(Q, K_t)/scale
```

```
# attn weights: softmax on scaled dp
attn = F.softmax(scores, dim=-1)
```

```
# contextualized output: dot product
out = torch.matmul(attn, V)
return out, attn
```



Query

What I'm looking for

Key

What I look like in this space

Value

If you attend to me,
here's what I'll give you

Attention Heads

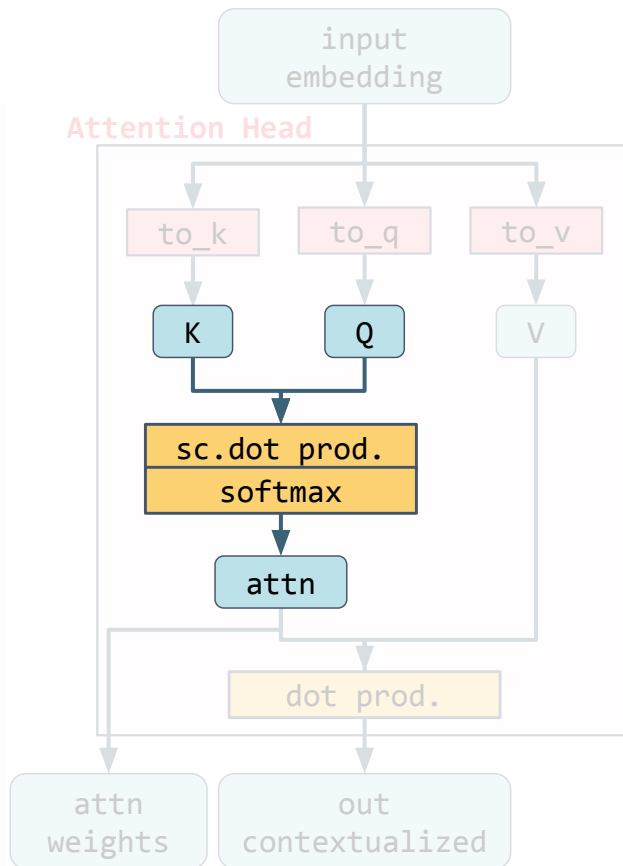
```
to_k = nn.Linear(4, 4)
to_q = nn.Linear(4, 4)
to_v = nn.Linear(4, 4)
```

```
K = to_k(input_embedding)
Q = to_q(input_embedding)
V = to_v(input_embedding)
```

```
K_t = K.transpose(-2, 1)
scale = math.sqrt(Q.size(-1))
# KQ compare: scaled dot product
scores = torch.matmul(Q, K_t)/scale
```

```
# attn weights: softmax on scaled dp
attn = F.softmax(scores, dim=-1)
```

```
# contextualized output: dot product
out = torch.matmul(attn, V)
return out, attn
```



Query

What I'm looking for

Key

What I look like in this space

Value

If you attend to me,
here's what I'll give you

Attention Heads

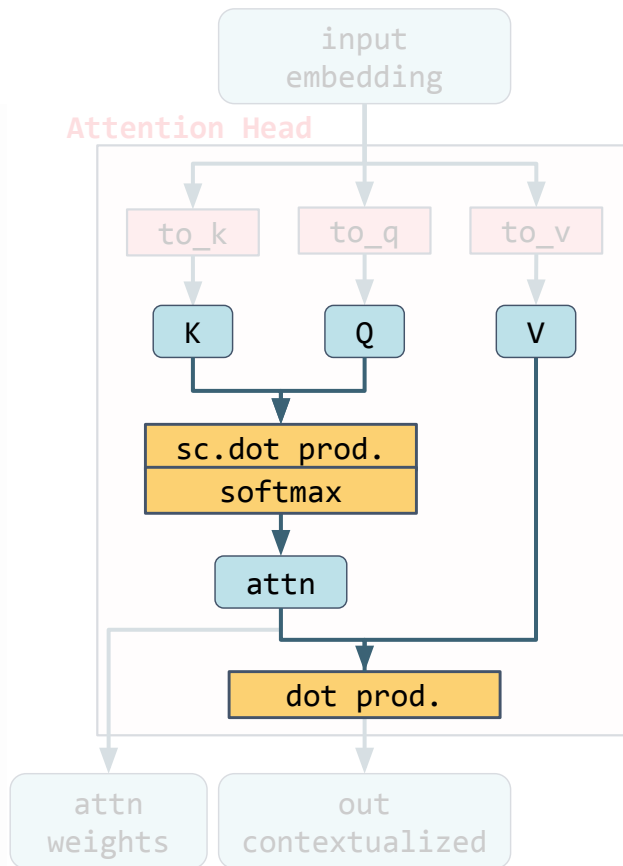
```
to_k = nn.Linear(4, 4)
to_q = nn.Linear(4, 4)
to_v = nn.Linear(4, 4)
```

```
K = to_k(input_embedding)
Q = to_q(input_embedding)
V = to_v(input_embedding)
```

```
K_t = K.transpose(-2, 1)
scale = math.sqrt(Q.size(-1))
# KQ compare: scaled dot product
scores = torch.matmul(Q, K_t)/scale
```

```
# attn weights: softmax on scaled dp
attn = F.softmax(scores, dim=-1)
```

```
# contextualized output: dot product
out = torch.matmul(attn, V)
return out, attn
```



Query

What I'm looking for

Key

What I look like in this space

Value

If you attend to me,
here's what I'll give you

Attention Heads

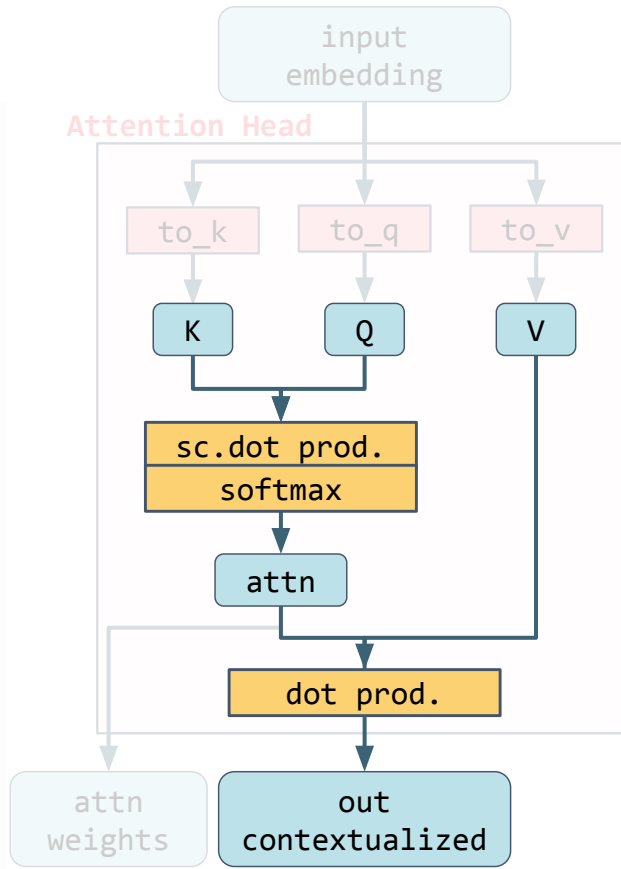
```
to_k = nn.Linear(4, 4)
to_q = nn.Linear(4, 4)
to_v = nn.Linear(4, 4)
```

```
K = to_k(input_embedding)
Q = to_q(input_embedding)
V = to_v(input_embedding)
```

```
K_t = K.transpose(-2, 1)
scale = math.sqrt(Q.size(-1))
# KQ compare: scaled dot product
scores = torch.matmul(Q, K_t)/scale
```

```
# attn weights: softmax on scaled dp
attn = F.softmax(scores, dim=-1)
```

```
# contextualized output: dot product
out = torch.matmul(attn, V)
return out, attn
```



Query

What I'm looking for

Key

What I look like in this space

Value

If you attend to me,
here's what I'll give you

nn.MultiheadAttention

```
import torch
import torch.nn as nn

mha = nn.MultiheadAttention(embed_dim=4,
                             num_heads=2,
                             dropout=0.1,
                             batch_first=True)
```

nn.MultiheadAttention

```
import torch
import torch.nn as nn

mha = nn.MultiheadAttention(embed_dim=4,
                             num_heads=2,
                             dropout=0.1,
                             batch_first=True)

torch.tensor([[ [ 0.12, -0.55, 0.33, 0.10], # the
                [-0.44, 0.91, -0.12, -0.77], # cat
                [ 0.48, 0.02, 0.05, 0.39], # chased
                [ 0.12, -0.55, 0.33, 0.10], # the
                [-0.30, 0.14, -0.70, 0.81]]]) # dog
```

nn.MultiheadAttention

```
import torch
import torch.nn as nn

mha = nn.MultiheadAttention(embed_dim=4,
                             num_heads=2,
                             dropout=0.1,
                             batch_first=True)
```

Attention Head 1

Attention Head 2

Attention Head 3

Attention Head 4

Attention Head 5

...

nn.MultiheadAttention

```
import torch
import torch.nn as nn

mha = nn.MultiheadAttention(embed_dim=4,
                             num_heads=2,
                             dropout=0.1,
                             batch_first=True)
```

nn.MultiheadAttention

```
import torch
import torch.nn as nn
```

```
mha = nn.MultiheadAttention(embed_dim=4,
                             num_heads=2,
                             dropout=0.1,
                             batch_first=True)
```

(batch_size, seq_len, embed_dim)

nn.MultiheadAttention

```
import torch
import torch.nn as nn

mha = nn.MultiheadAttention(embed_dim=4,
                             num_heads=2,
                             dropout=0.1,
                             batch_first=True)

      Q   K   V
attn_output, attn_weights = mha(x, x, x) self attention
```

nn.MultiheadAttention

```
import torch
import torch.nn as nn

mha = nn.MultiheadAttention(embed_dim=4,
                             num_heads=2,
                             dropout=0.1,
                             batch_first=True)

      Q   K   V
attn_output, attn_weights = mha(x, x, x) cross attention
```

nn.MultiheadAttention

```
import torch
import torch.nn as nn

mha = nn.MultiheadAttention(embed_dim=4,
                             num_heads=2,
                             dropout=0.1,
                             batch_first=True)

      Q   K   V
attn_output, attn_weights = mha(x, x, x) cross attention
```

nn.MultiheadAttention

```
import torch
import torch.nn as nn

mha = nn.MultiheadAttention(embed_dim=4,
                             num_heads=2,
                             dropout=0.1,
                             batch_first=True)

      Q   K   V
attn_output, attn_weights = mha(x, y, y) cross attention
```



DeepLearning.AI

Encoders

Specialized Approaches to
Natural Language Processing in Pytorch

Attention



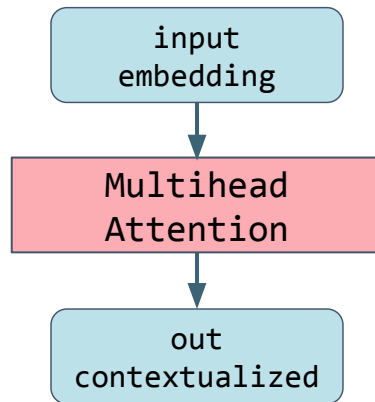
The cat chased the mouse

A diagram illustrating attention weights. It consists of a series of overlapping, downward-curving arcs positioned above the words of the sentence. The arcs are centered over the words 'cat', 'chased', and 'the', indicating that these words are the primary focus of attention.

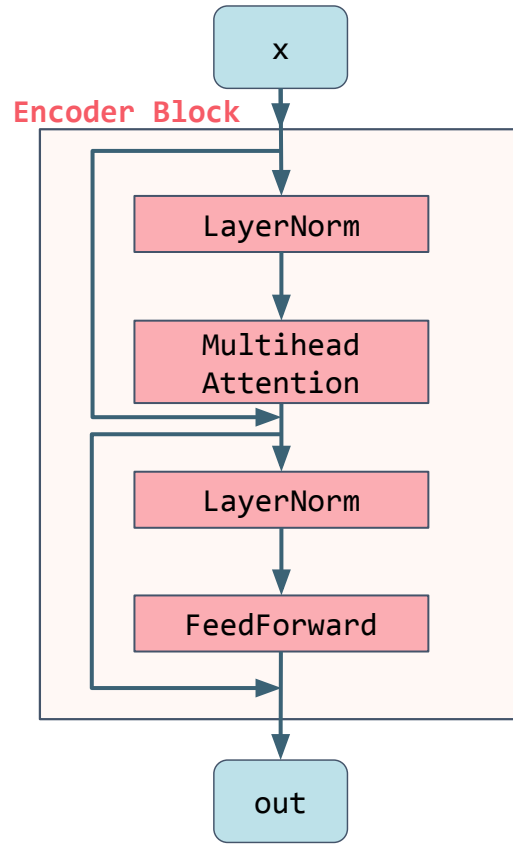
Attention



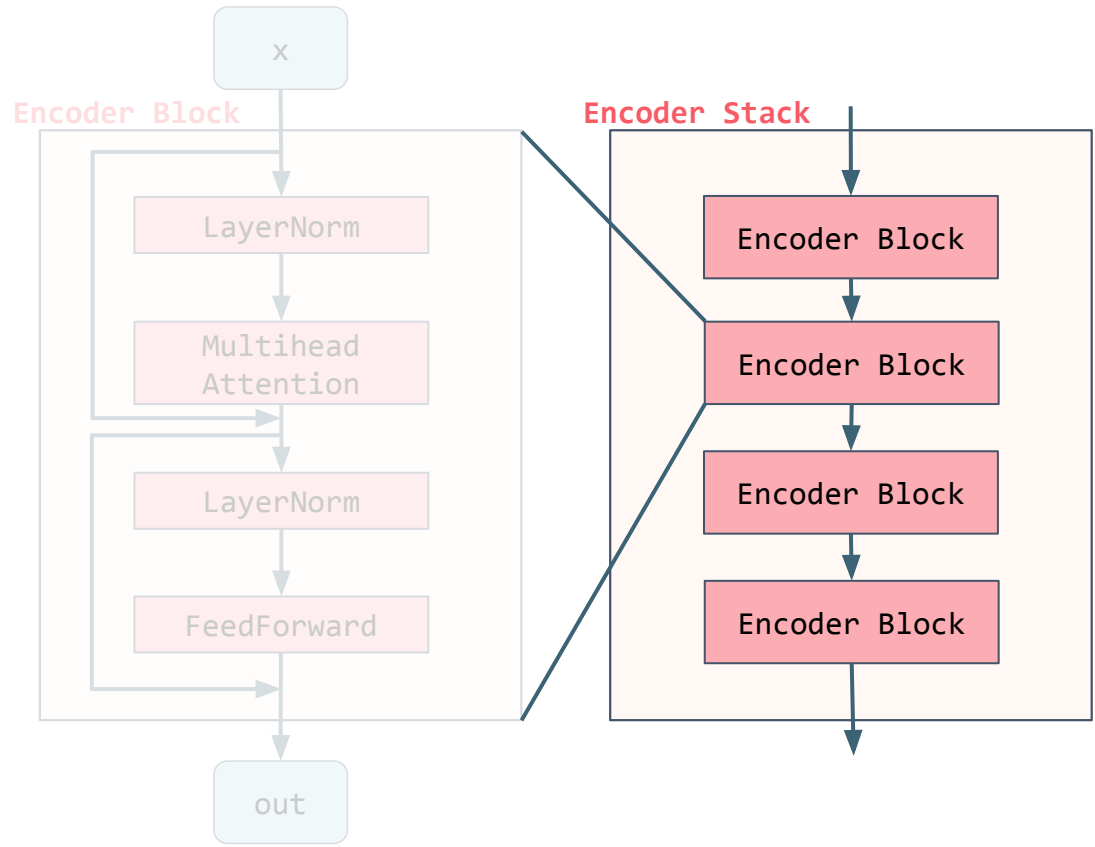
The cat chased the mouse



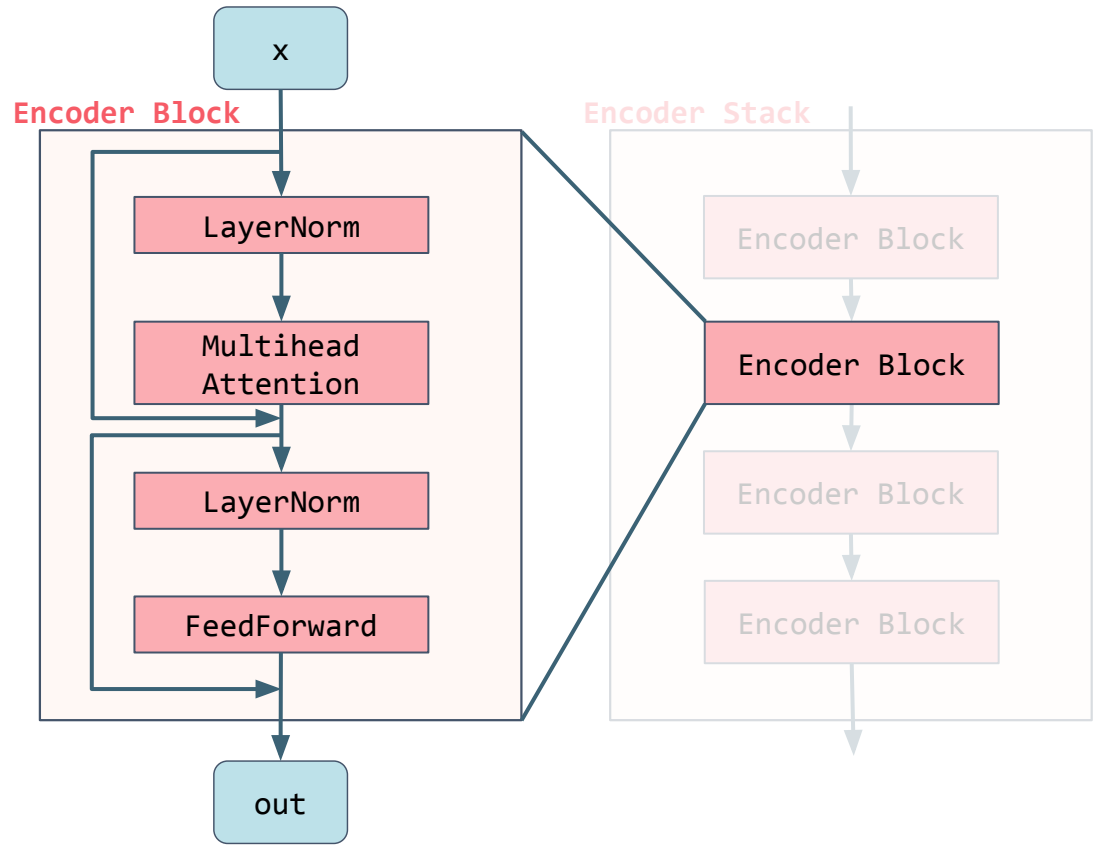
Encoder Block



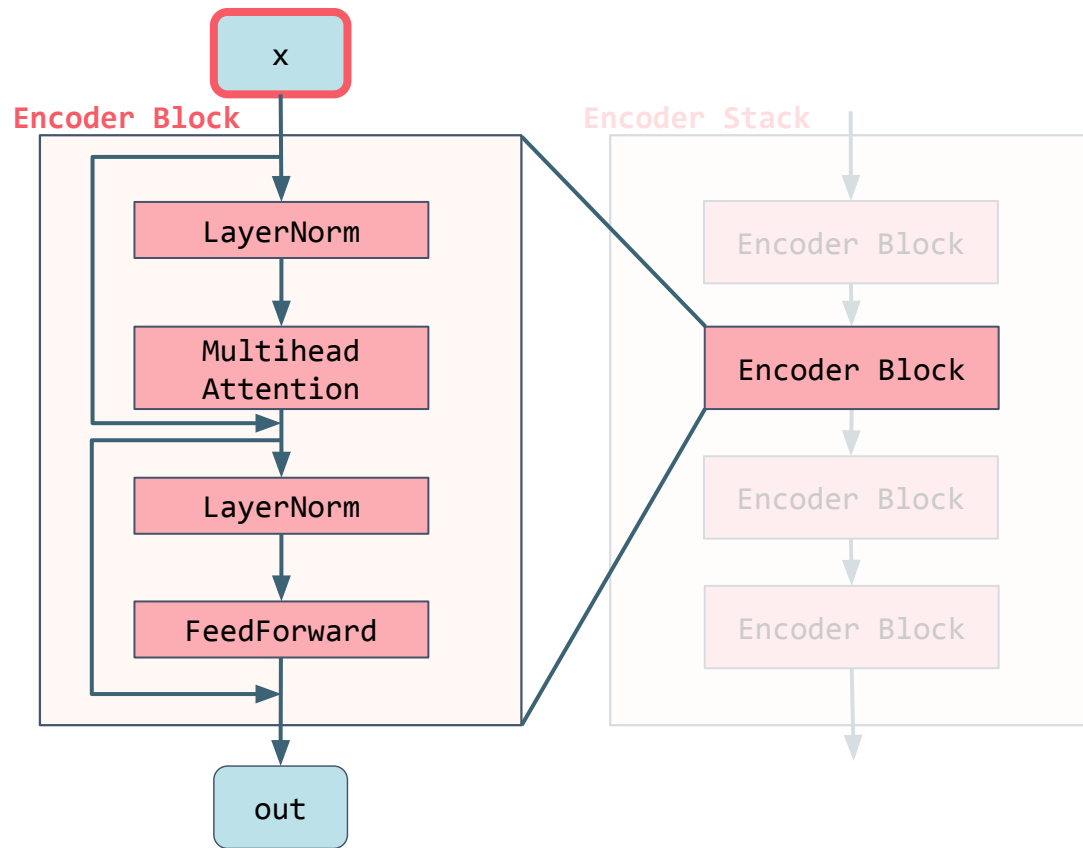
Encoder Block



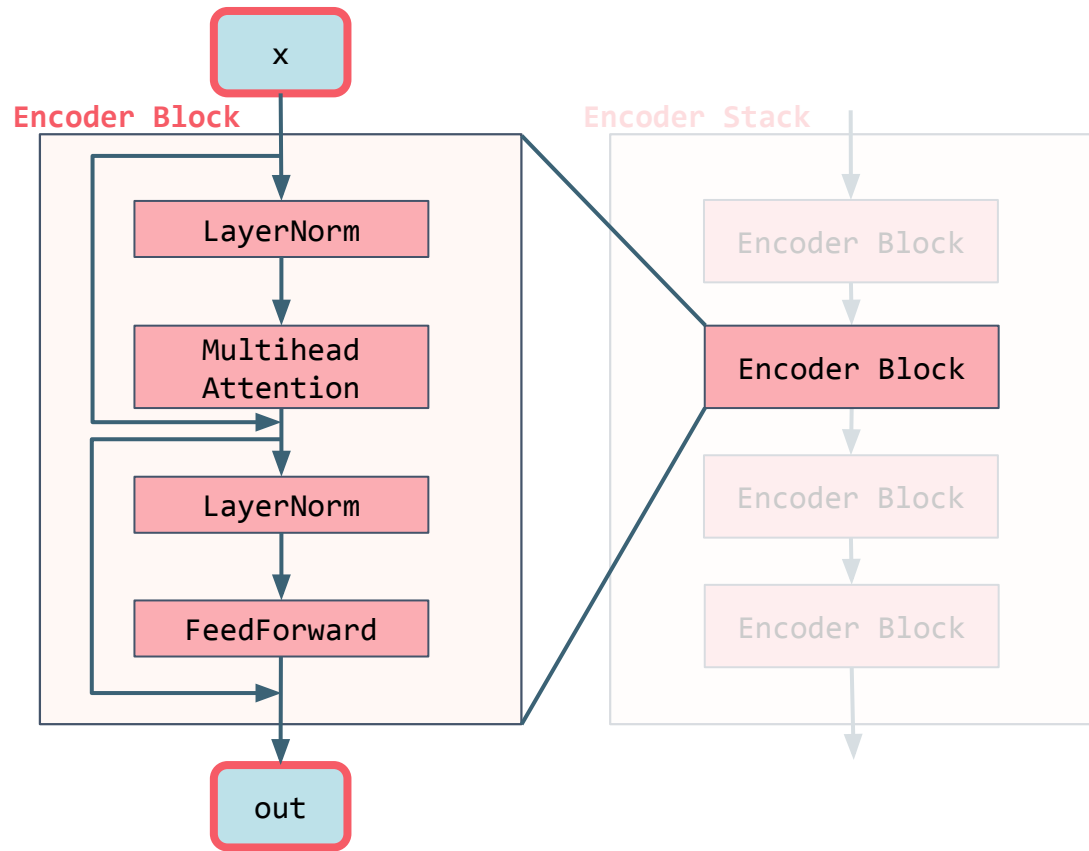
Encoder Block



Encoder Block



Encoder Block



nn.TransformerEncoder

```
import torch.nn as nn  
  
mha = nn.MultiheadAttention(. . .)
```

nn.TransformerEncoder

```
import torch.nn as nn

mha = nn.MultiheadAttention(. . .)

layer = nn.TransformerEncoderLayer(d_model=768,
                                    nhead=12,
                                    dim_feedforward=3072,
                                    dropout=0.1,
                                    batch_first=True)

stack = nn.TransformerEncoder(layer, num_layers=4)
```

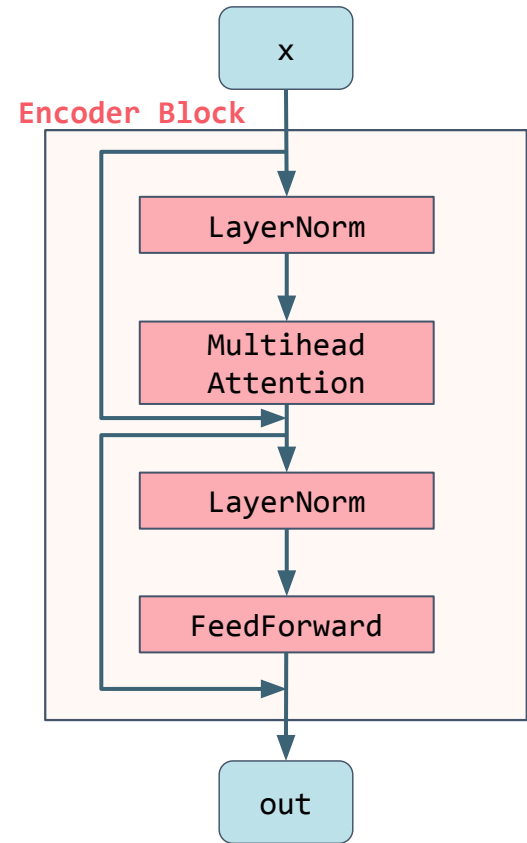

nn.TransformerEncoder

```
import torch.nn as nn

mha = nn.MultiheadAttention(. . .)

layer = nn.TransformerEncoderLayer(d_model=768,
                                    nhead=12,
                                    dim_feedforward=3072,
                                    dropout=0.1,
                                    batch_first=True)

stack = nn.TransformerEncoder(layer, num_layers=4)
```



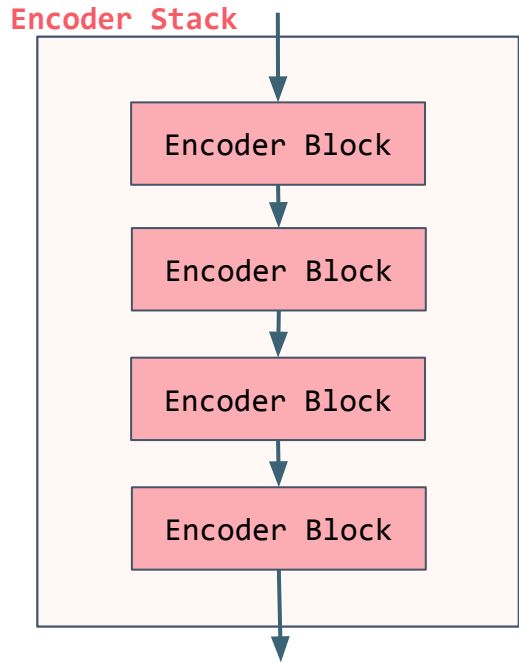
nn.TransformerEncoder

```
import torch.nn as nn

mha = nn.MultiheadAttention(. . .)

layer = nn.TransformerEncoderLayer(d_model=768,
                                   nhead=12,
                                   dim_feedforward=3072,
                                   dropout=0.1,
                                   batch_first=True)

stack = nn.TransformerEncoder(layer, num_layers=4)
```



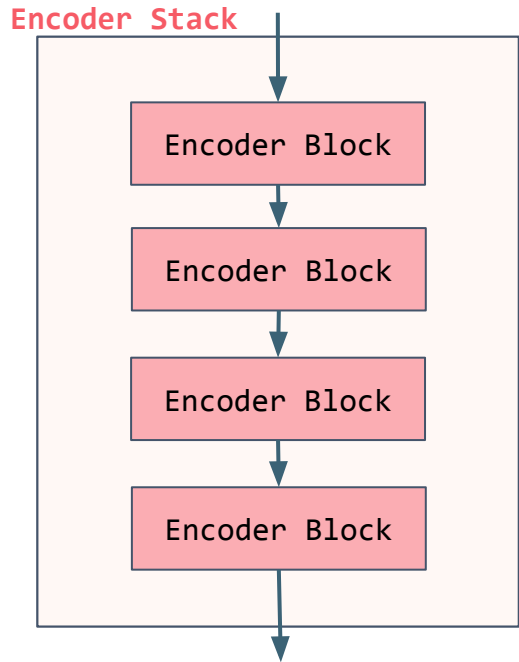
nn.TransformerEncoder

```
import torch.nn as nn

mha = nn.MultiheadAttention(. . .)

layer = nn.TransformerEncoderLayer(d_model=768,
                                   nhead=12,
                                   dim_feedforward=3072,
                                   dropout=0.1,
                                   batch_first=True)

stack = nn.TransformerEncoder(layer, num_layers=4)
```



nn.TransformerEncoder

```
import torch.nn as nn
```

```
mha = nn.MultiheadAttention(. . .
```

```
layer = nn.TransformerEncoderLayer
```

```
stack = nn.TransformerEncoder(layer
```

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com	Noam Shazeer* Google Brain noam@google.com	Niki Parmar* Google Research nikip@google.com	Jakob Uszkoreit* Google Research usz@google.com
Llion Jones* Google Research llion@google.com	Aidan N. Gomez*[†] University of Toronto aidan@cs.toronto.edu	Lukasz Kaiser* Google Brain lukaszkaizer@google.com	
Illia Polosukhin*[‡] illia.polosukhin@gmail.com			

Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature. We show that the Transformer generalizes well to other tasks by applying it successfully to English constituency parsing both with large and limited training data.

nn.TransformerEncoder

```
import torch.nn as nn

mha = nn.MultiheadAttention(. . .)

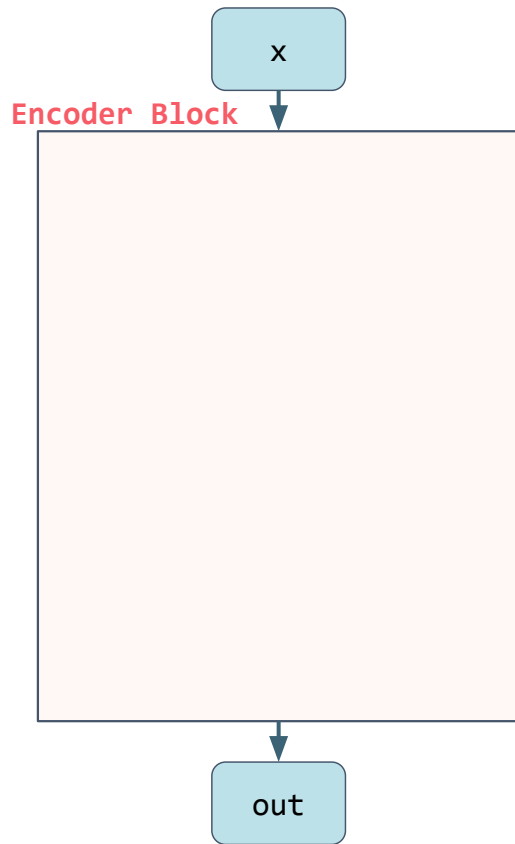
layer = nn.TransformerEncoderLayer(d_model=768,
                                    nhead=12,
                                    dim_feedforward=3072,
                                    dropout=0.1,
                                    batch_first=True)

stack = nn.TransformerEncoder(layer, num_layers=4)
```

Encoder Block

```
class EncoderBlock(nn.Module):  
    def __init__(self, d_model=4, nhead=1, ffn_mult=4):  
        super().__init__()
```

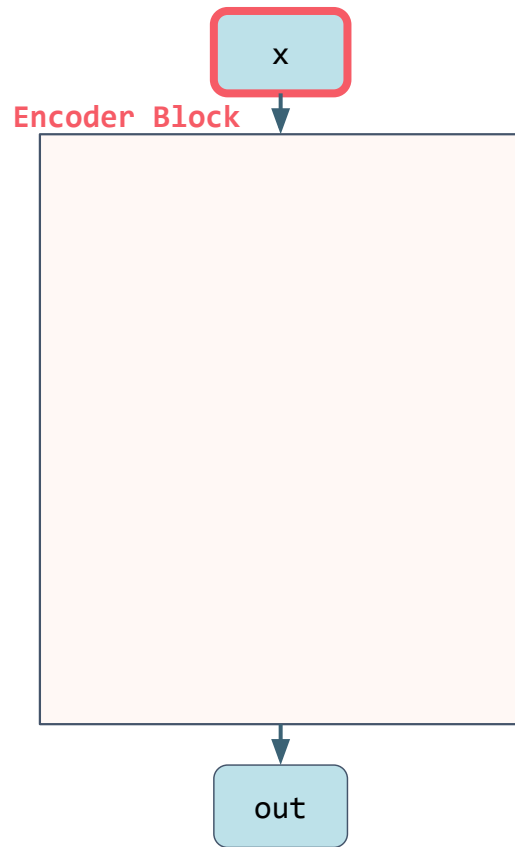
```
    def forward(self, x):
```



Encoder Block

```
class EncoderBlock(nn.Module):  
    def __init__(self, d_model=4, nhead=1, ffn_mult=4):  
        super().__init__()
```

```
    def forward(self, x):
```



Input

x

```
torch.tensor([[[ 0.12, -0.55,  0.33,  0.10], # the
                 [-0.44,  0.91, -0.12, -0.77], # cat
                 [ 0.48,  0.02,  0.05,  0.39], # chased
                 [ 0.12, -0.55,  0.33,  0.10], # the
                 [-0.30,  0.14, -0.70,  0.81]]]) # dog
```


Input

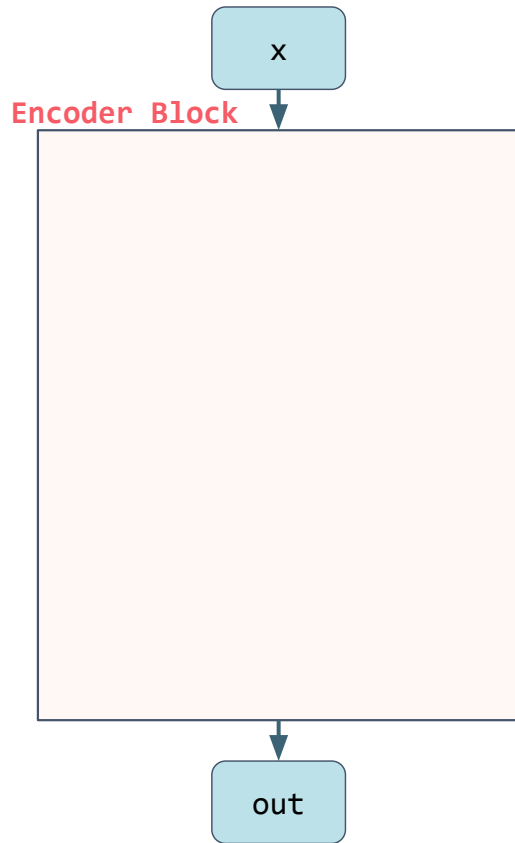
$$x = \text{positional embedding} + \text{token embedding}$$

```
torch.tensor([[[ 0.12, -0.55,  0.33,  0.10], # the
                [-0.44,  0.91, -0.12, -0.77], # cat
                [ 0.48,  0.02,  0.05,  0.39], # chased
                [ 0.12, -0.55,  0.33,  0.10], # the
                [-0.30,  0.14, -0.70,  0.81]]]) # dog
```

Encoder Block

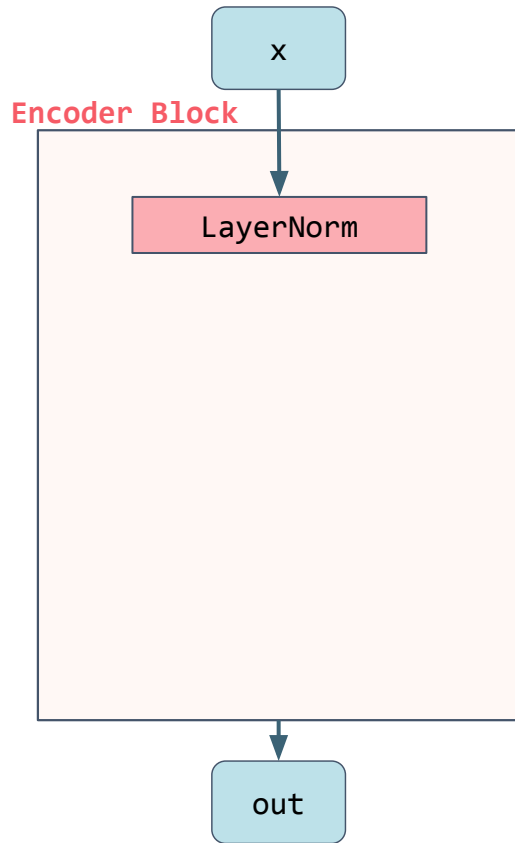
```
class EncoderBlock(nn.Module):  
    def __init__(self, d_model=4, nhead=1, ffn_mult=4):  
        super().__init__()
```

```
    def forward(self, x):
```



Encoder Block

```
class EncoderBlock(nn.Module):  
    def __init__(self, d_model=4, nhead=1, ffn_mult=4):  
        super().__init__()  
        self.ln1 = nn.LayerNorm(d_model)  
  
    def forward(self, x):  
        x_norm = self.ln1(x)
```



LayerNorm

LayerNorm

Tokens normalized independently

```
torch.tensor([[[ 0.12, -0.55, 0.33, 0.10], # the  
               [-0.44, 0.91, -0.12, -0.77], # cat  
               [ 0.48, 0.02, 0.05, 0.39], # chased  
               [ 0.12, -0.55, 0.33, 0.10], # the  
               [-0.30, 0.14, -0.70, 0.81]]]) # dog
```

LayerNorm

LayerNorm

Tokens normalized independently

```
torch.tensor([[[ 0.12, -0.55, 0.33, 0.10], # the  
               [-0.53, 1.61, -0.02, -1.05], # cat  
               [ 0.48, 0.02, 0.05, 0.39], # chased  
               [ 0.12, -0.55, 0.33, 0.10], # the  
               [-0.30, 0.14, -0.70, 0.81]]]) # dog
```

LayerNorm

LayerNorm

Tokens normalized independently

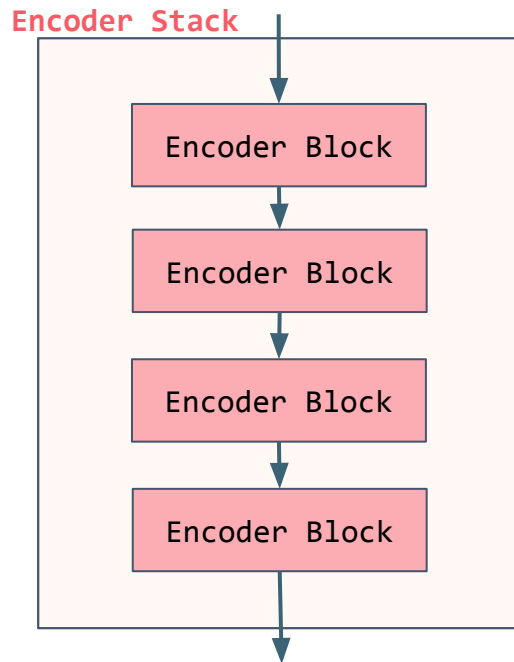
```
torch.tensor([[[ 0.12, -0.55, 0.33, 0.10], # the  
               [-0.53, 1.61, -0.02, -1.05], # cat  
               [ 0.48, 0.02, 0.05, 0.39], # chased  
               [ 0.12, -0.55, 0.33, 0.10], # the  
               [-0.51, 0.27, -1.23, 1.47]]]) # dog
```

LayerNorm

LayerNorm

Tokens normalized independently

```
torch.tensor([[[ 0.12, -0.55, 0.33, 0.10], # the  
                [-0.53, 1.61, -0.02, -1.05], # cat  
                [ 0.48, 0.02, 0.05, 0.39], # chased  
                [ 0.12, -0.55, 0.33, 0.10], # the  
                [-0.51, 0.27, -1.23, 1.47]]]) # dog
```

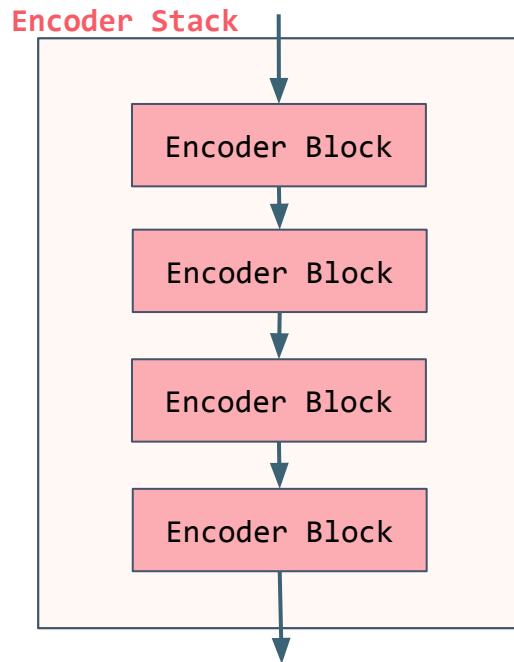


LayerNorm

LayerNorm

Tokens normalized independently

```
torch.tensor([[[ 0.12, -0.55, 0.33, 0.10], # the  
               [-0.53, 1.61, -0.02, -1.05], # cat  
               [ 0.48, 0.02, 0.05, 0.39], # chased  
               [ 0.12, -0.55, 0.33, 0.10], # the  
               [-0.51, 0.27, -1.23, 1.47]]]) # dog
```

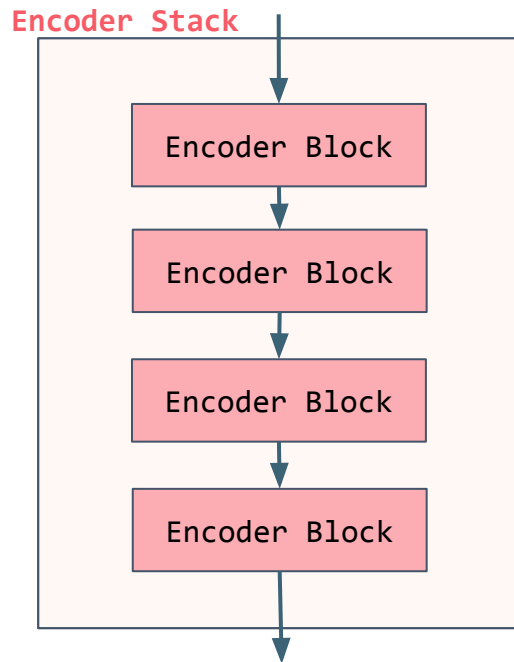


LayerNorm

LayerNorm

Tokens normalized independently

```
torch.tensor([[[ 0.12, -0.55, 0.33, 0.10], # the  
[ 12.4, -57.8, 30.2, 8.9], # cat  
[ 0.48, 0.02, 0.05, 0.39], # chased  
[ 0.12, -0.55, 0.33, 0.10], # the  
[-0.51, 0.27, -1.23, 1.47]]]) # dog
```

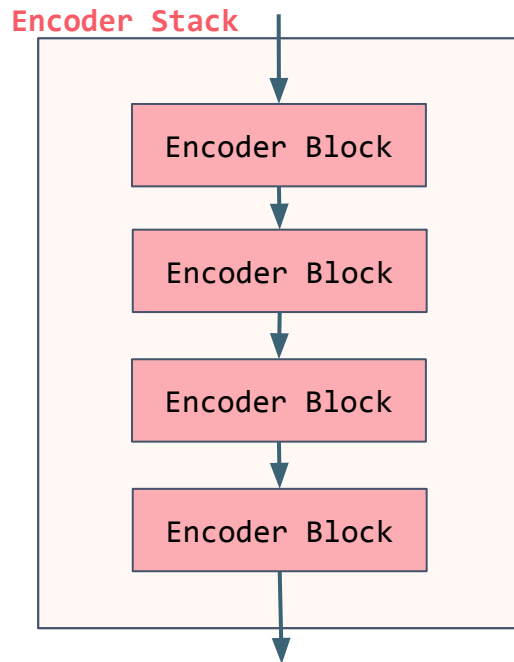


LayerNorm

LayerNorm

Tokens normalized independently

```
torch.tensor([[[ 0.12, -0.55, 0.33, 0.10], # the  
[ .001, -.004, .002, 0.009], # cat  
[ 0.48, 0.02, 0.05, 0.39], # chased  
[ 0.12, -0.55, 0.33, 0.10], # the  
[-0.51, 0.27, -1.23, 1.47]]]) # dog
```

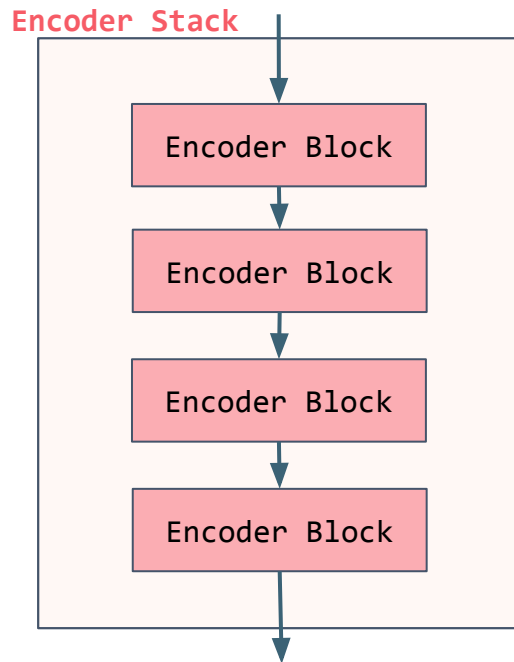


LayerNorm

LayerNorm

Tokens normalized independently

```
torch.tensor([[[ 0.12, -0.55, 0.33, 0.10], # the  
                [-0.44, 0.91, -0.12, -0.77], # cat  
                [ 0.48, 0.02, 0.05, 0.39], # chased  
                [ 0.12, -0.55, 0.33, 0.10], # the  
                [-0.51, 0.27, -1.23, 1.47]]]) # dog
```

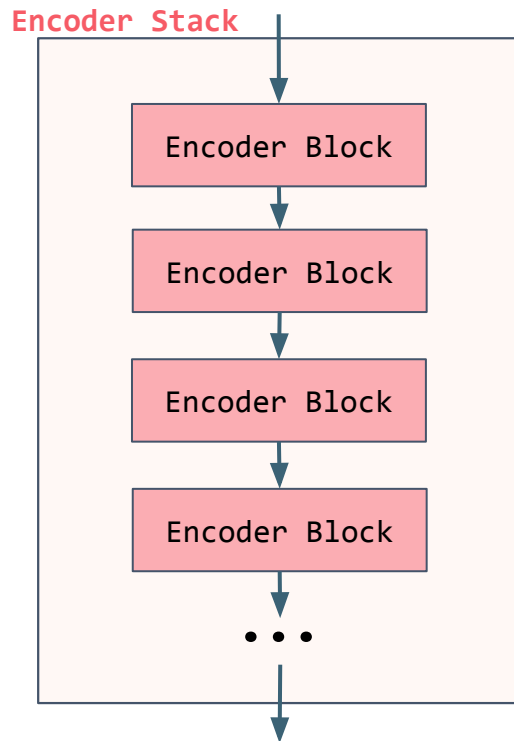


LayerNorm

LayerNorm

Tokens normalized independently

```
torch.tensor([[[ 0.12, -0.55, 0.33, 0.10], # the  
                [-0.44, 0.91, -0.12, -0.77], # cat  
                [ 0.48, 0.02, 0.05, 0.39], # chased  
                [ 0.12, -0.55, 0.33, 0.10], # the  
                [-0.51, 0.27, -1.23, 1.47]]]) # dog
```



LayerNorm

LayerNorm

mean: 0 **variance:** 1

```
torch.tensor([[[ 0.12, -0.55,  0.33,  0.10], # the
                [-0.44,  0.91, -0.12, -0.77], # cat
                [ 0.48,  0.02,  0.05,  0.39], # chased
                [ 0.12, -0.55,  0.33,  0.10], # the
                [-0.51,  0.27, -1.23,  1.47]]]) # dog
```

LayerNorm

LayerNorm

mean: 0 variance: 1

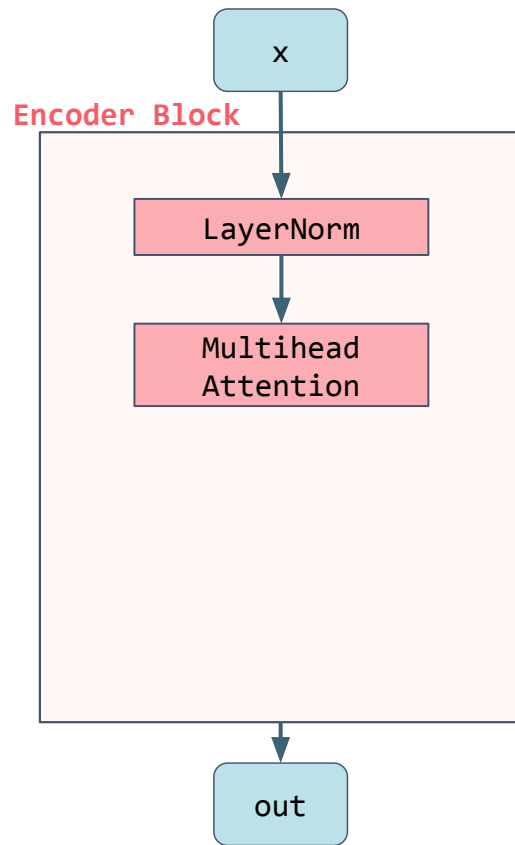
```
torch.tensor([[ 0.12, -0.55,  0.33,  0.10], # the  
               [-0.44,  0.91, -0.12, -0.77], # cat  
               [ 0.48,  0.02,  0.05,  0.39], # chased  
               [ 0.12, -0.55,  0.33,  0.10], # the  
               [-0.51,  0.27, -1.23,  1.47]]) # dog
```

scale	γ_1	γ_2	γ_3	γ_4
shift	β_1	β_2	β_3	β_4

$$\gamma_1 \mathbf{x}_{t,1} + \beta_1$$

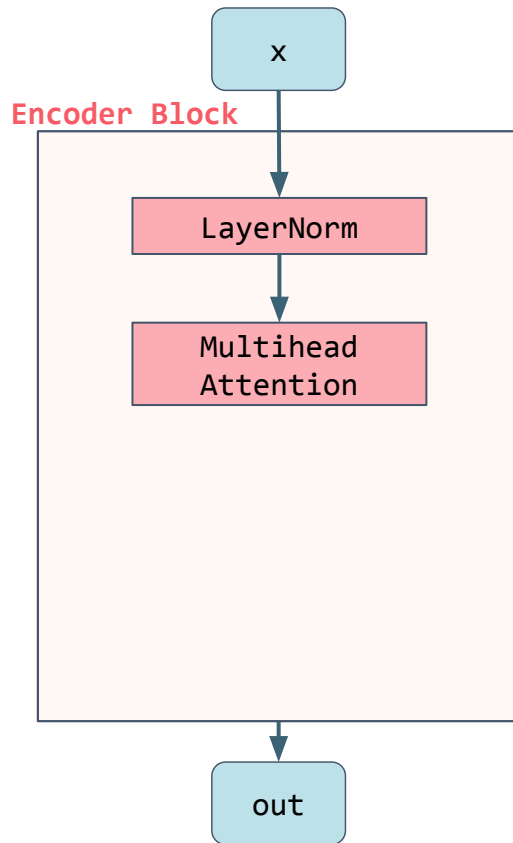
Encoder Block

```
class EncoderBlock(nn.Module):  
    def __init__(self, d_model=4, nhead=1, ffn_mult=4):  
        super().__init__()  
        self.ln1 = nn.LayerNorm(d_model)  
  
    def forward(self, x):  
        x_norm = self.ln1(x)
```



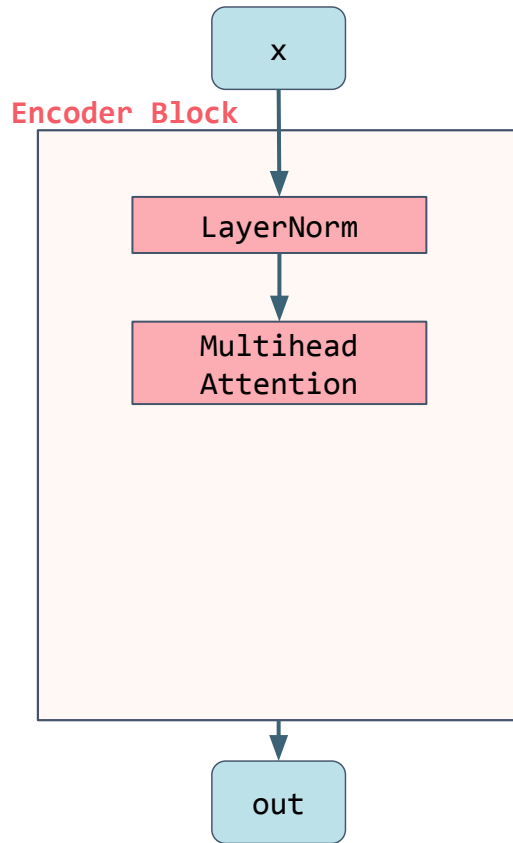
Encoder Block

```
class EncoderBlock(nn.Module):  
    def __init__(self, d_model=4, nhead=1, ffn_mult=4):  
        super().__init__()  
        self.ln1 = nn.LayerNorm(d_model)  
        self.mha = nn.MultiheadAttention(. . .)  
  
    def forward(self, x):  
        x_norm = self.ln1(x)
```



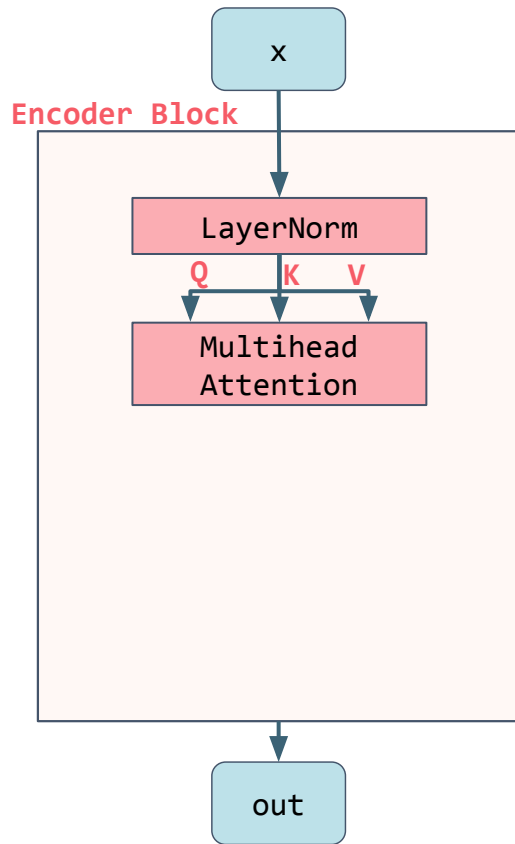
Encoder Block

```
class EncoderBlock(nn.Module):  
    def __init__(self, d_model=4, nhead=1, ffn_mult=4):  
        super().__init__()  
        self.ln1 = nn.LayerNorm(d_model)  
        self.mha = nn.MultiheadAttention(. . .)  
  
    def forward(self, x):  
        x_norm = self.ln1(x)  
        attn_out, _ = self.mha(x_norm, x_norm, x_norm)
```



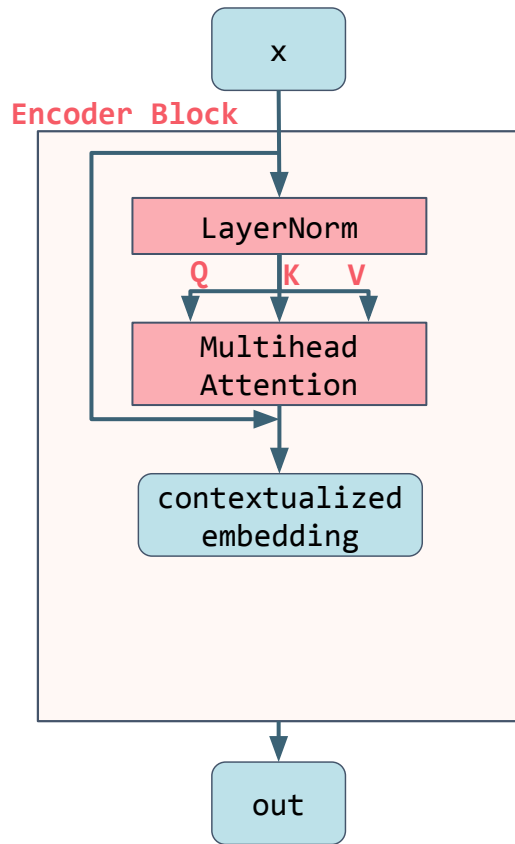
Encoder Block

```
class EncoderBlock(nn.Module):  
    def __init__(self, d_model=4, nhead=1, ffn_mult=4):  
        super().__init__()  
        self.ln1 = nn.LayerNorm(d_model)  
        self.mha = nn.MultiheadAttention(. . .)  
  
    def forward(self, x):  
        x_norm = self.ln1(x)  
        attn_out, _ = self.mha(x_norm, x_norm, x_norm)
```



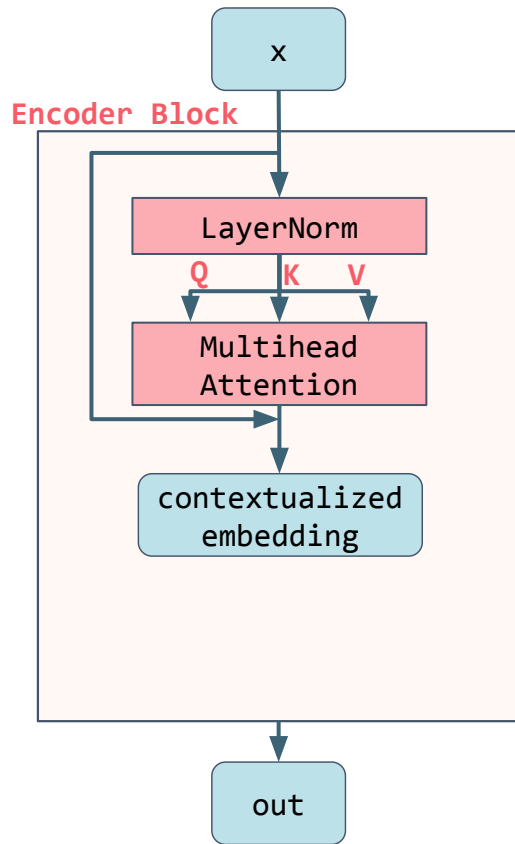
Encoder Block

```
class EncoderBlock(nn.Module):  
    def __init__(self, d_model=4, nhead=1, ffn_mult=4):  
        super().__init__()  
        self.ln1 = nn.LayerNorm(d_model)  
        self.mha = nn.MultiheadAttention(. . .)  
  
    def forward(self, x):  
        x_norm = self.ln1(x)  
        attn_out, _ = self.mha(x_norm, x_norm, x_norm)
```



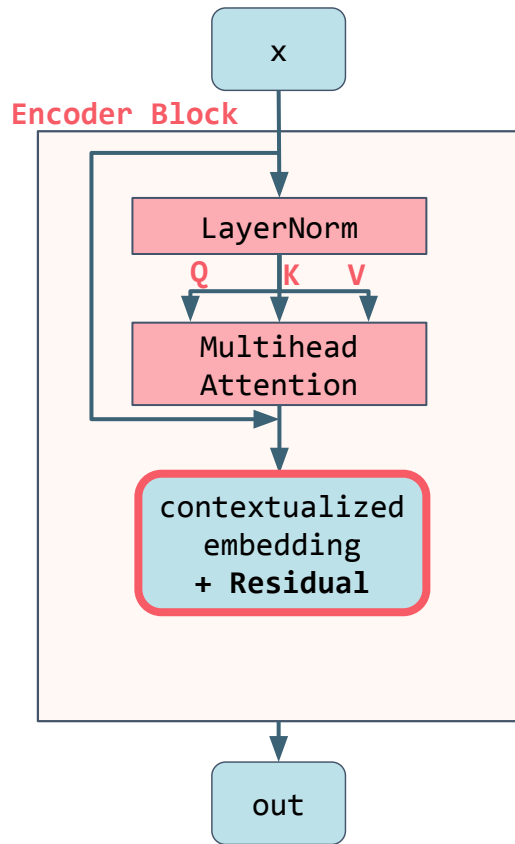
Encoder Block

```
class EncoderBlock(nn.Module):  
    def __init__(self, d_model=4, nhead=1, ffn_mult=4):  
        super().__init__()  
        self.ln1 = nn.LayerNorm(d_model)  
        self.mha = nn.MultiheadAttention(. . .)  
  
    def forward(self, x):  
        x_norm = self.ln1(x)  
        attn_out, _ = self.mha(x_norm, x_norm, x_norm)  
        x = x + attn_out
```



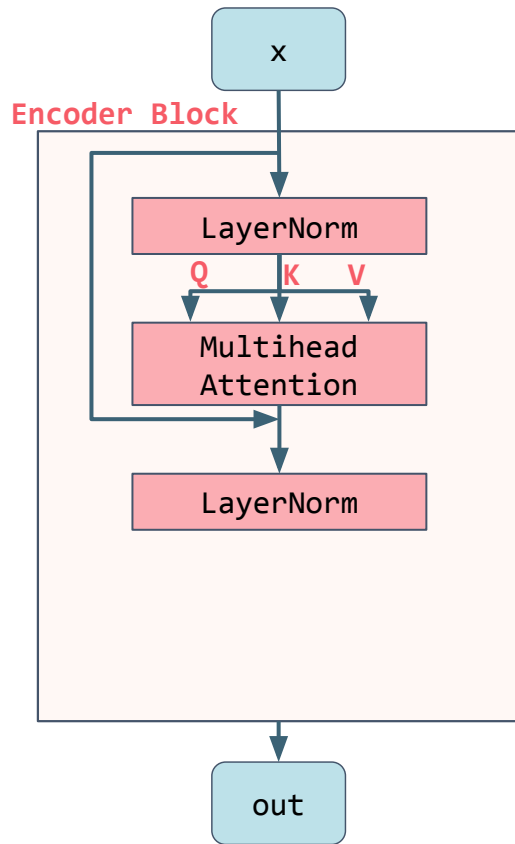
Encoder Block

```
class EncoderBlock(nn.Module):  
    def __init__(self, d_model=4, nhead=1, ffn_mult=4):  
        super().__init__()  
        self.ln1 = nn.LayerNorm(d_model)  
        self.mha = nn.MultiheadAttention(. . .)  
  
    def forward(self, x):  
        x_norm = self.ln1(x)  
        attn_out, _ = self.mha(x_norm, x_norm, x_norm)  
        x = x + attn_out
```



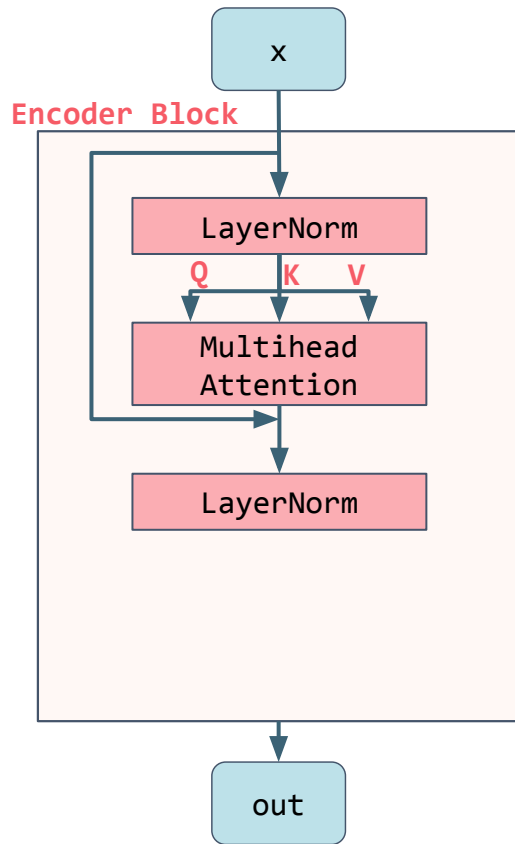
Encoder Block

```
class EncoderBlock(nn.Module):  
    def __init__(self, d_model=4, nhead=1, ffn_mult=4):  
        super().__init__()  
        self.ln1 = nn.LayerNorm(d_model)  
        self.mha = nn.MultiheadAttention(. . .)  
  
    def forward(self, x):  
        x_norm = self.ln1(x)  
        attn_out, _ = self.mha(x_norm, x_norm, x_norm)  
        x = x + attn_out
```



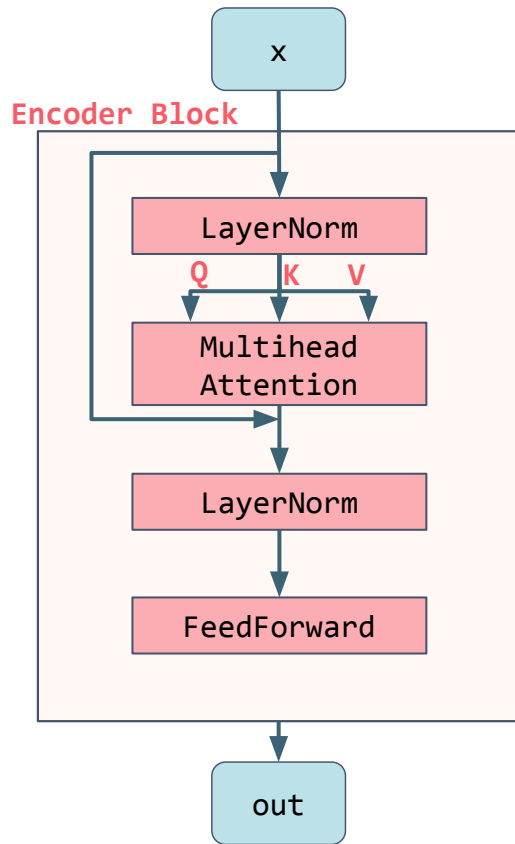
Encoder Block

```
class EncoderBlock(nn.Module):  
    def __init__(self, d_model=4, nhead=1, ffn_mult=4):  
        super().__init__()  
        self.ln1 = nn.LayerNorm(d_model)  
        self.mha = nn.MultiheadAttention(. . .)  
        self.ln2 = nn.LayerNorm(d_model)  
  
    def forward(self, x):  
        x_norm = self.ln1(x)  
        attn_out, _ = self.mha(x_norm, x_norm, x_norm)  
        x = x + attn_out  
        ffn_in = self.ln2(x)
```



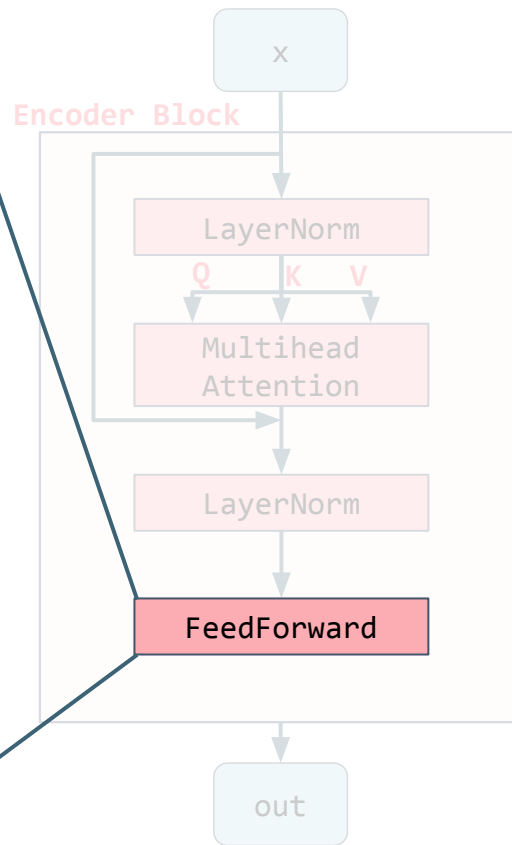
Encoder Block

```
class EncoderBlock(nn.Module):  
    def __init__(self, d_model=4, nhead=1, ffn_mult=4):  
        super().__init__()  
        self.ln1 = nn.LayerNorm(d_model)  
        self.mha = nn.MultiheadAttention(. . .)  
        self.ln2 = nn.LayerNorm(d_model)  
  
    def forward(self, x):  
        x_norm = self.ln1(x)  
        attn_out, _ = self.mha(x_norm, x_norm, x_norm)  
        x = x + attn_out  
        ffn_in = self.ln2(x)
```



Encoder Block

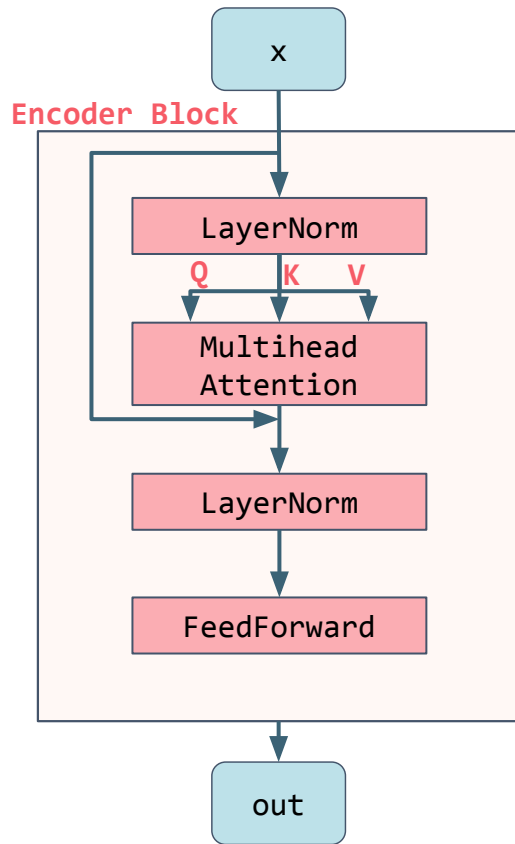
```
nn.Sequential(  
    nn.Linear(d_model, hidden),  
    nn.ReLU(),  
    nn.Linear(hidden, d_model))
```



Encoder Block

```
class EncoderBlock(nn.Module):
    def __init__(self, d_model=4, nhead=1, ffn_mult=4):
        super().__init__()
        self.ln1 = nn.LayerNorm(d_model)
        self.mha = nn.MultiheadAttention(. . .)
        self.ln2 = nn.LayerNorm(d_model)
        hidden = ffn_mult * d_model
        self.ffn = nn.Sequential(. . .)

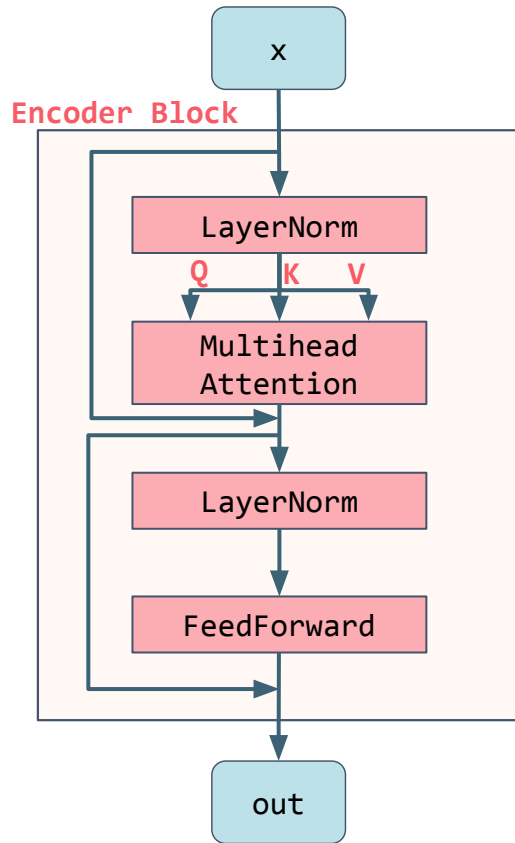
    def forward(self, x):
        x_norm = self.ln1(x)
        attn_out, _ = self.mha(x_norm, x_norm, x_norm)
        x = x + attn_out
        ffn_in = self.ln2(x)
        ffn_out = self.ffn(ffn_in)
```



Encoder Block

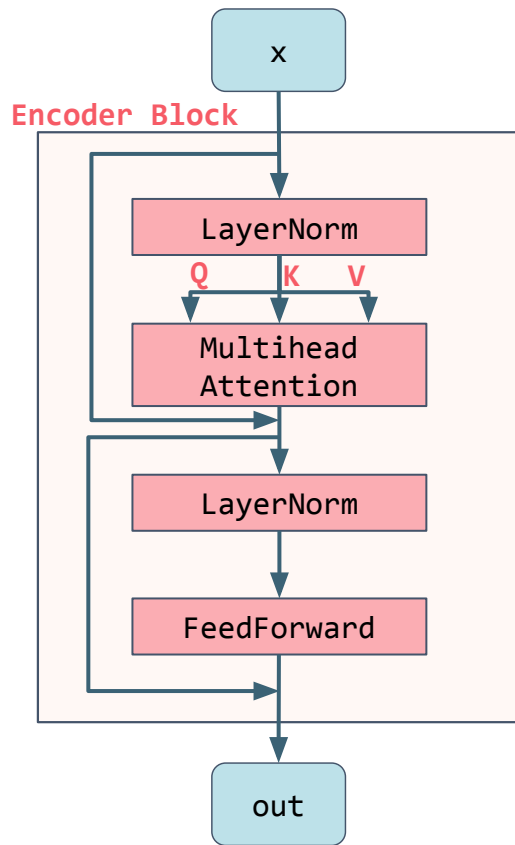
```
class EncoderBlock(nn.Module):
    def __init__(self, d_model=4, nhead=1, ffn_mult=4):
        super().__init__()
        self.ln1 = nn.LayerNorm(d_model)
        self.mha = nn.MultiheadAttention(. . .)
        self.ln2 = nn.LayerNorm(d_model)
        hidden = ffn_mult * d_model
        self.ffn = nn.Sequential(. . .)

    def forward(self, x):
        x_norm = self.ln1(x)
        attn_out, _ = self.mha(x_norm, x_norm, x_norm)
        x = x + attn_out
        ffn_in = self.ln2(x)
        ffn_out = self.ffn(ffn_in)
        x = x + ffn_out
        return x
```



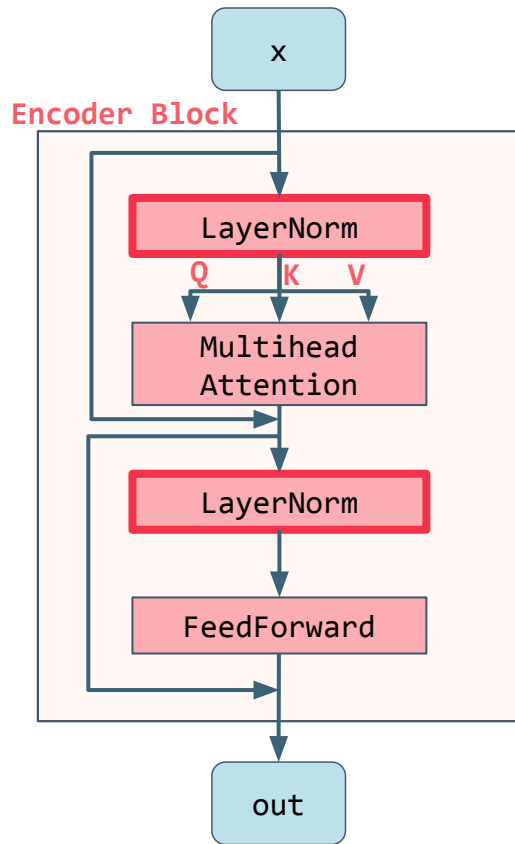
Encoder Block

```
class EncoderBlock(nn.Module):  
    def __init__(self, d_model=4, nhead=1, ffn_mult=4):  
        super().__init__()  
        self.ln1 = nn.LayerNorm(d_model)  
        self.mha = nn.MultiheadAttention(. . .)  
        self.ln2 = nn.LayerNorm(d_model)  
        hidden = ffn_mult * d_model  
        self.ffn = nn.Sequential(. . .)  
  
    def forward(self, x):  
        x_norm = self.ln1(x)  
        attn_out, _ = self.mha(x_norm, x_norm, x_norm)  
        x = x + attn_out  
        ffn_in = self.ln2(x)  
        ffn_out = self.ffn(ffn_in)  
        x = x + ffn_out  
        return x
```



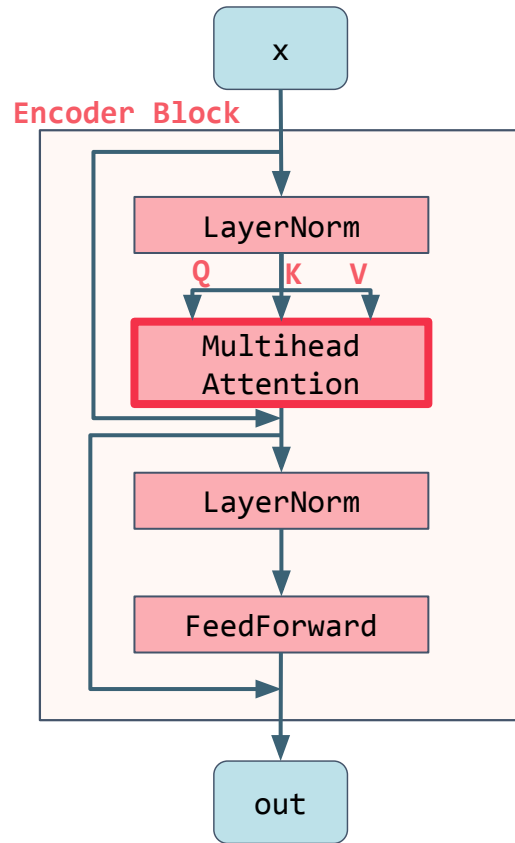
Encoder Block

```
class EncoderBlock(nn.Module):  
    def __init__(self, d_model=4, nhead=1, ffn_mult=4):  
        super().__init__()  
        self.ln1 = nn.LayerNorm(d_model)  
        self.mha = nn.MultiheadAttention(. . .)  
        self.ln2 = nn.LayerNorm(d_model)  
        hidden = ffn_mult * d_model  
        self.ffn = nn.Sequential(. . .)  
  
    def forward(self, x):  
        x_norm = self.ln1(x)  
        attn_out, _ = self.mha(x_norm, x_norm, x_norm)  
        x = x + attn_out  
        ffn_in = self.ln2(x)  
        ffn_out = self.ffn(ffn_in)  
        x = x + ffn_out  
        return x
```



Encoder Block

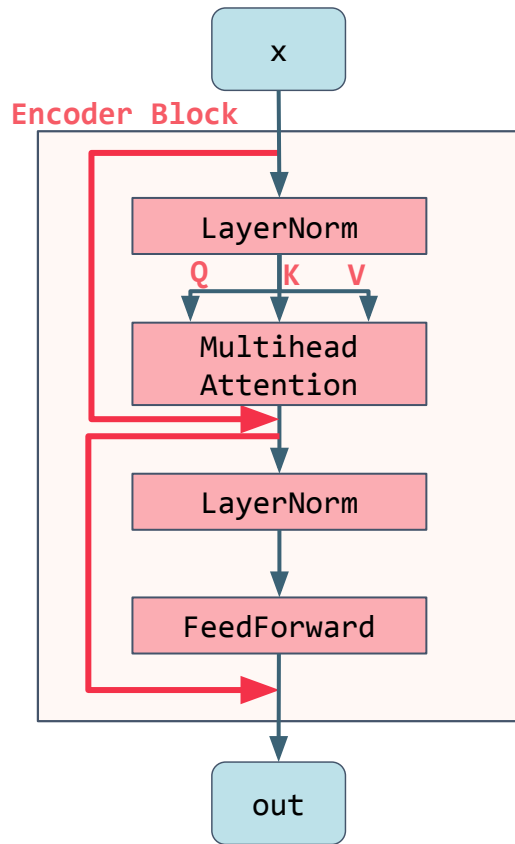
```
class EncoderBlock(nn.Module):  
    def __init__(self, d_model=4, nhead=1, ffn_mult=4):  
        super().__init__()  
        self.ln1 = nn.LayerNorm(d_model)  
        self.mha = nn.MultiheadAttention(. . .)  
        self.ln2 = nn.LayerNorm(d_model)  
        hidden = ffn_mult * d_model  
        self.ffn = nn.Sequential(. . .)  
  
    def forward(self, x):  
        x_norm = self.ln1(x)  
        attn_out, _ = self.mha(x_norm, x_norm, x_norm)  
        x = x + attn_out  
        ffn_in = self.ln2(x)  
        ffn_out = self.ffn(ffn_in)  
        x = x + ffn_out  
        return x
```



Encoder Block

```
class EncoderBlock(nn.Module):
    def __init__(self, d_model=4, nhead=1, ffn_mult=4):
        super().__init__()
        self.ln1 = nn.LayerNorm(d_model)
        self.mha = nn.MultiheadAttention(. . .)
        self.ln2 = nn.LayerNorm(d_model)
        hidden = ffn_mult * d_model
        self.ffn = nn.Sequential(. . .)

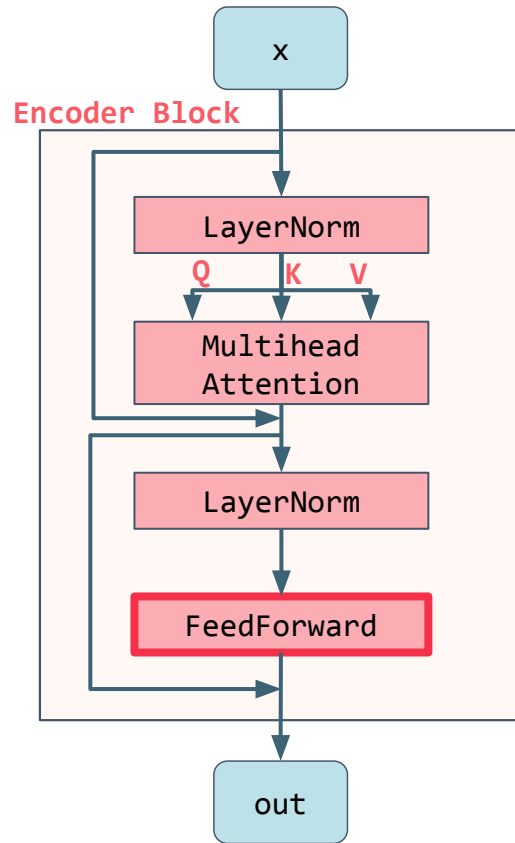
    def forward(self, x):
        x_norm = self.ln1(x)
        attn_out, _ = self.mha(x_norm, x_norm, x_norm)
        x = x + attn_out
        ffn_in = self.ln2(x)
        ffn_out = self.ffn(ffn_in)
        x = x + ffn_out
        return x
```



Encoder Block

```
class EncoderBlock(nn.Module):
    def __init__(self, d_model=4, nhead=1, ffn_mult=4):
        super().__init__()
        self.ln1 = nn.LayerNorm(d_model)
        self.mha = nn.MultiheadAttention(. . .)
        self.ln2 = nn.LayerNorm(d_model)
        hidden = ffn_mult * d_model
        self.ffn = nn.Sequential(. . .)

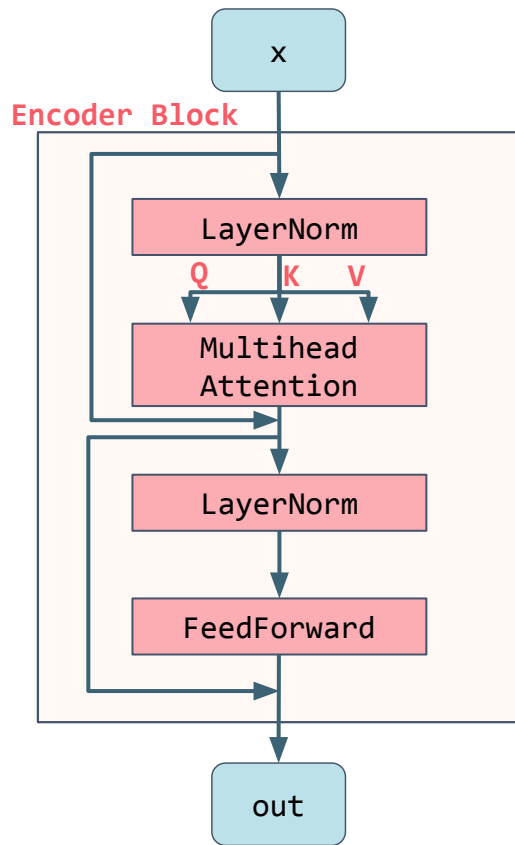
    def forward(self, x):
        x_norm = self.ln1(x)
        attn_out, _ = self.mha(x_norm, x_norm, x_norm)
        x = x + attn_out
        ffn_in = self.ln2(x)
        ffn_out = self.ffn(ffn_in)
        x = x + ffn_out
        return x
```



Encoder Block

```
class EncoderBlock(nn.Module):
    def __init__(self, d_model=4, nhead=1, ffn_mult=4):
        super().__init__()
        self.ln1 = nn.LayerNorm(d_model)
        self.mha = nn.MultiheadAttention(. . .)
        self.ln2 = nn.LayerNorm(d_model)
        hidden = ffn_mult * d_model
        self.ffn = nn.Sequential(. . .)

    def forward(self, x):
        x_norm = self.ln1(x)
        attn_out, _ = self.mha(x_norm, x_norm, x_norm)
        x = x + attn_out
        ffn_in = self.ln2(x)
        ffn_out = self.ffn(ffn_in)
        x = x + ffn_out
        return x
```



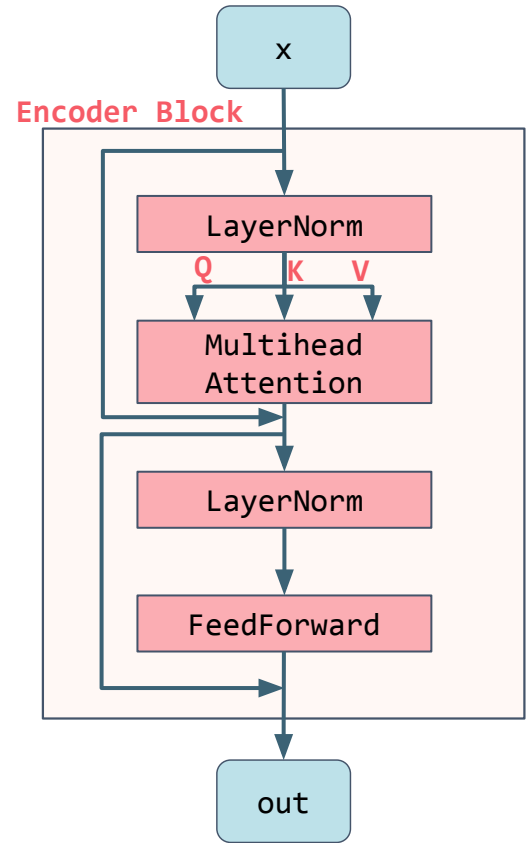
nn.TransformerEncoder

```
layer = nn.TransformerEncoderLayer(d_model=768,  
                                   nhead=12,  
                                   dim_feedforward=3072,  
                                   dropout=0.1,  
                                   batch_first=True)  
  
stack = nn.TransformerEncoder(layer, num_layers=4)
```

nn.TransformerEncoder

```
layer = nn.TransformerEncoderLayer(d_model=768,  
                                   nhead=12,  
                                   dim_feedforward=3072,  
                                   dropout=0.1,  
                                   batch_first=True)
```

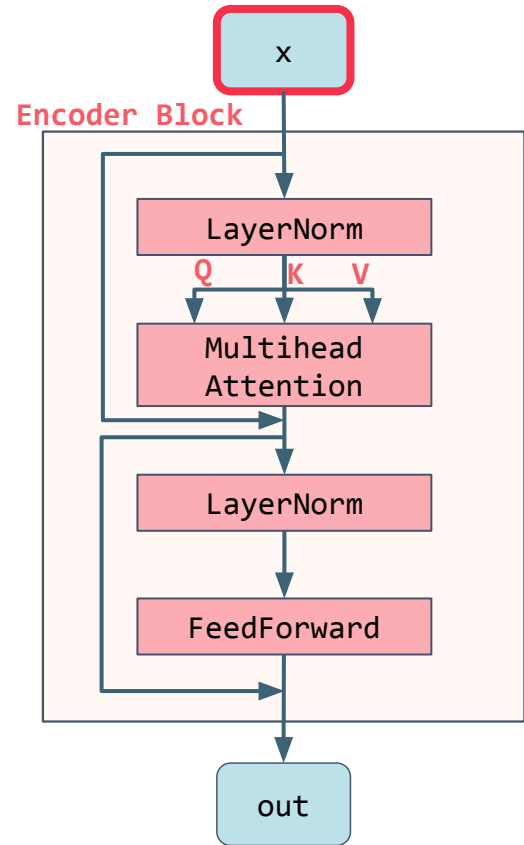
```
stack = nn.TransformerEncoder(layer, num_layers=4)
```



nn.TransformerEncoder

```
layer = nn.TransformerEncoderLayer(d_model=768,  
                                   nhead=12,  
                                   dim_feedforward=3072,  
                                   dropout=0.1,  
                                   batch_first=True)
```

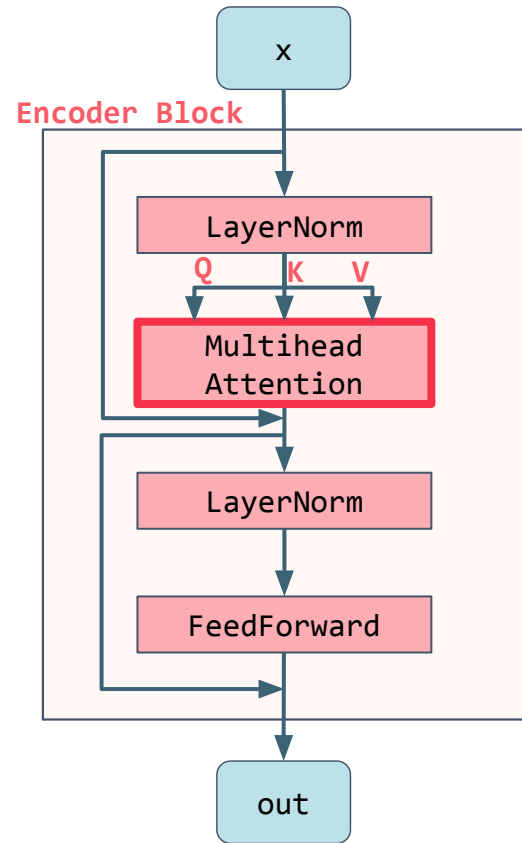
```
stack = nn.TransformerEncoder(layer, num_layers=4)
```



nn.TransformerEncoder

```
layer = nn.TransformerEncoderLayer(d_model=768,  
                                   nhead=12,  
                                   dim_feedforward=3072,  
                                   dropout=0.1,  
                                   batch_first=True)
```

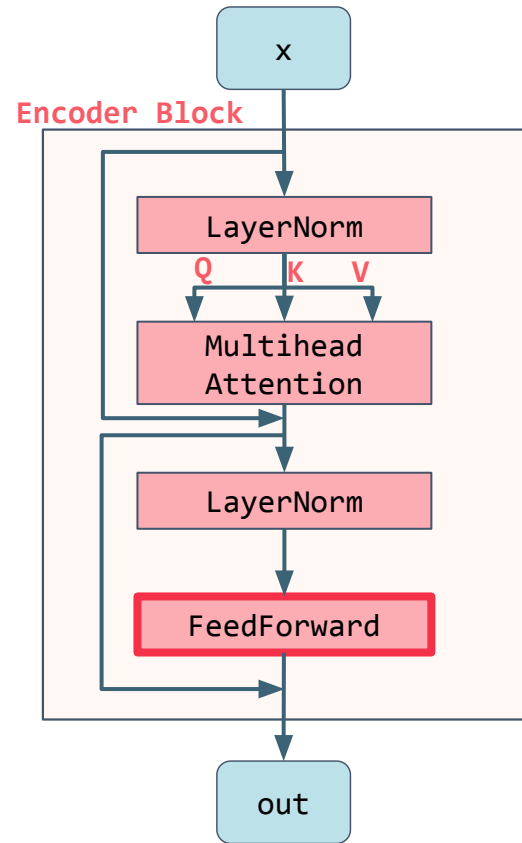
```
stack = nn.TransformerEncoder(layer, num_layers=4)
```



nn.TransformerEncoder

```
layer = nn.TransformerEncoderLayer(d_model=768,  
                                   nhead=12,  
                                   dim_feedforward=3072,  
                                   dropout=0.1,  
                                   batch_first=True)
```

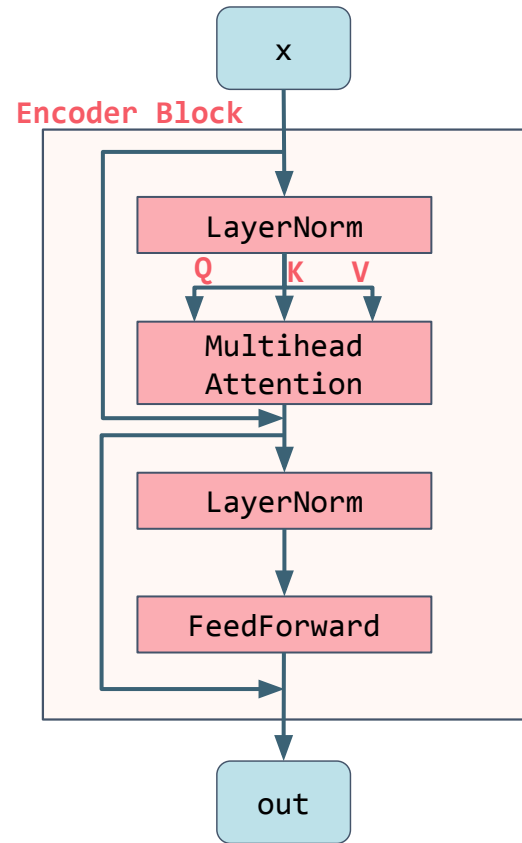
```
stack = nn.TransformerEncoder(layer, num_layers=4)
```



nn.TransformerEncoder

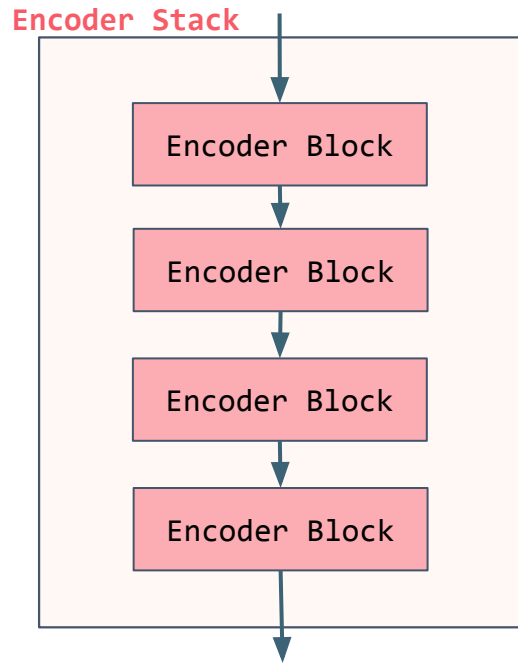
```
layer = nn.TransformerEncoderLayer(d_model=768,  
                                   nhead=12,  
                                   dim_feedforward=3072,  
                                   dropout=0.1,  
                                   batch_first=True)
```

```
stack = nn.TransformerEncoder(layer, num_layers=4)
```



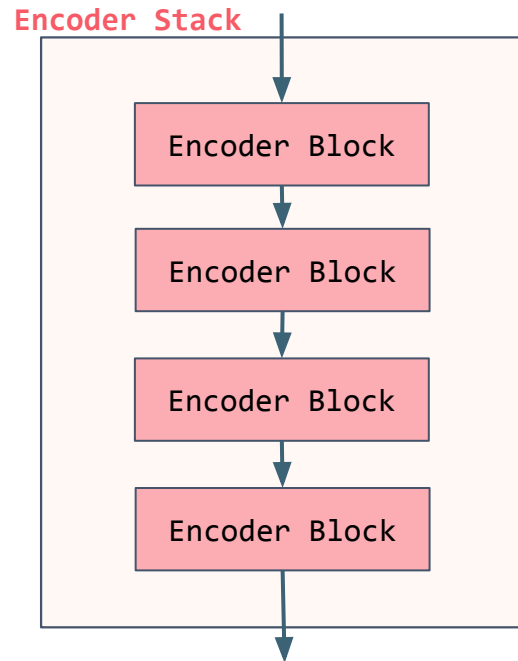
nn.TransformerEncoder

```
layer = nn.TransformerEncoderLayer(d_model=768,  
                                   nhead=12,  
                                   dim_feedforward=3072,  
                                   dropout=0.1,  
                                   batch_first=True)  
  
stack = nn.TransformerEncoder(layer, num_layers=4)
```



nn.TransformerEncoder

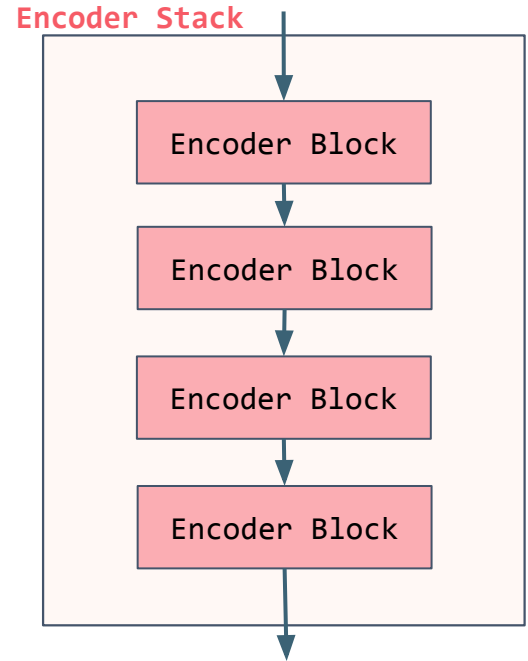
```
layer = nn.TransformerEncoderLayer(d_model=768,  
                                   nhead=12,  
                                   dim_feedforward=3072,  
                                   dropout=0.1,  
                                   batch_first=True)  
  
stack = nn.TransformerEncoder(layer, num_layers=4)
```



nn.TransformerEncoder

```
layer = nn.TransformerEncoderLayer(d_model=768,  
                                   nhead=12,  
                                   dim_feedforward=3072,  
                                   dropout=0.1,  
                                   batch_first=True)
```

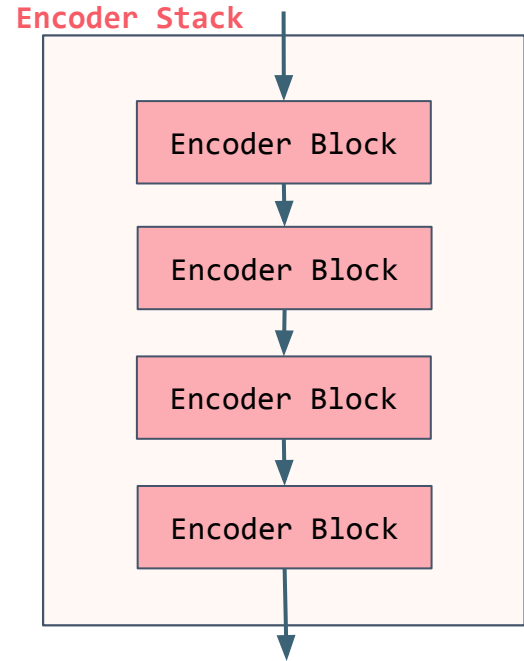
```
stack = nn.TransformerEncoder(layer, num_layers=4)
```



nn.TransformerEncoder

```
layer = nn.TransformerEncoderLayer(d_model=768,  
                                   nhead=12,  
                                   dim_feedforward=3072,  
                                   dropout=0.1,  
                                   batch_first=True)
```

```
stack = nn.TransformerEncoder(layer, num_layers=4)
```



Sentiment Analysis

```
class Sentiment(nn.Module):  
    . . .  
    def forward(self, x):  
        # Input embeddings  
        x = self.embedding(x)  
        pos_encoding = self.positional_encoding(x)  
        x = x + pos_encoding  
        x = self.dropout(x)  
  
        for encoder_layer in self.encoder_layers:  
            x = encoder_layer(x)  
  
        # Simple pooling  
        x = x.mean(dim=1)  
        # 2-class linear layer nn.Linear(dim, 2)  
        output = self.classifier(x)  
        return output
```

Sentiment Analysis

```
class Sentiment(nn.Module):  
    . . .  
    def forward(self, x):  
        # Input embeddings  
        x = self.embedding(x)  
        pos_encoding = self.positional_encoding(x)  
        x = x + pos_encoding  
        x = self.dropout(x)  
  
        for encoder_layer in self.encoder_layers:  
            x = encoder_layer(x)  
  
        # Simple pooling  
        x = x.mean(dim=1)  
        # 2-class linear layer nn.Linear(dim, 2)  
        output = self.classifier(x)  
        return output
```

Sentiment Analysis

```
class Sentiment(nn.Module):  
    . . .  
    def forward(self, x):  
        # Input embeddings  
        x = self.embedding(x)  
        pos_encoding = self.positional_encoding(x)  
        x = x + pos_encoding  
        x = self.dropout(x)  
  
        for encoder_layer in self.encoder_layers:  
            x = encoder_layer(x)  
  
        # Simple pooling  
        x = x.mean(dim=1)  
        # 2-class linear layer nn.Linear(dim, 2)  
        output = self.classifier(x)  
        return output
```

"I really loved this movie –
the acting was fantastic!"

POSITIVE

Transformer Architectures

- **Encoders:**
 - Understanding text
 - Classification
 - Named entity recognition
 - Question answering

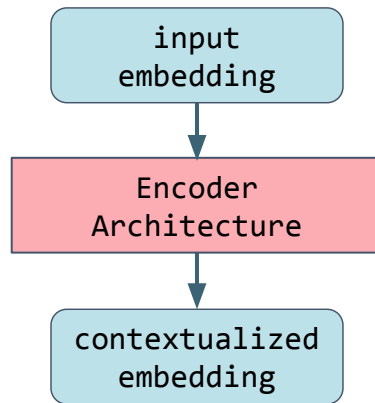


DeepLearning.AI

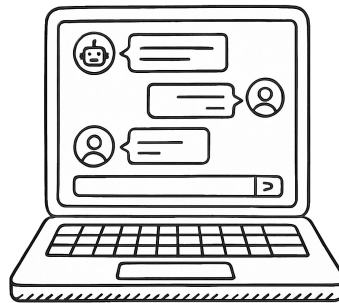
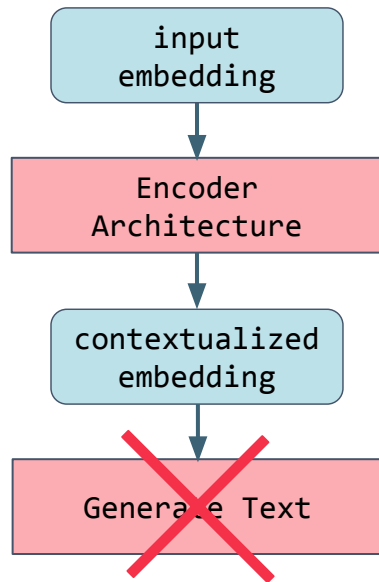
Decoders

Specialized Approaches to
Natural Language Processing in Pytorch

Encoders



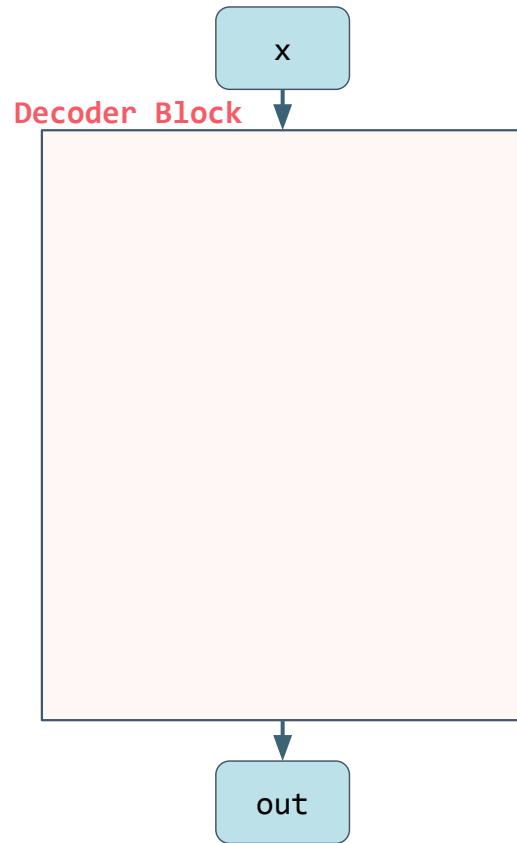
Encoders



Decoder Block

```
class DecoderBlock(nn.Module):  
    def __init__(self, d_model=4, nhead=1, ffn_mult=4):  
        super().__init__()
```

```
    def forward(self, x):
```



Predicting Tokens

The cat chased the _____

Predicting Tokens

The cat chased **the** **?** _____

Predicting Tokens

The cat chased **the** _____

Predicting Tokens

The cat chased **the** _____

A diagram illustrating the task of predicting a missing token in a sequence. The text "The cat chased the" is shown, with "the" highlighted in red. A blue curved arrow points from the end of the sequence to the blank space, indicating the prediction task. A horizontal line follows the blank space.

Predicting Tokens

The cat chased **the** _____

A diagram illustrating token prediction. The sentence "The cat chased the" is shown, with "the" in red. A horizontal line follows the word "the". Two curved arrows originate from the word "the": one points to the word "The" and the other points to the word "cat", indicating that the model is predicting these tokens based on the current context.

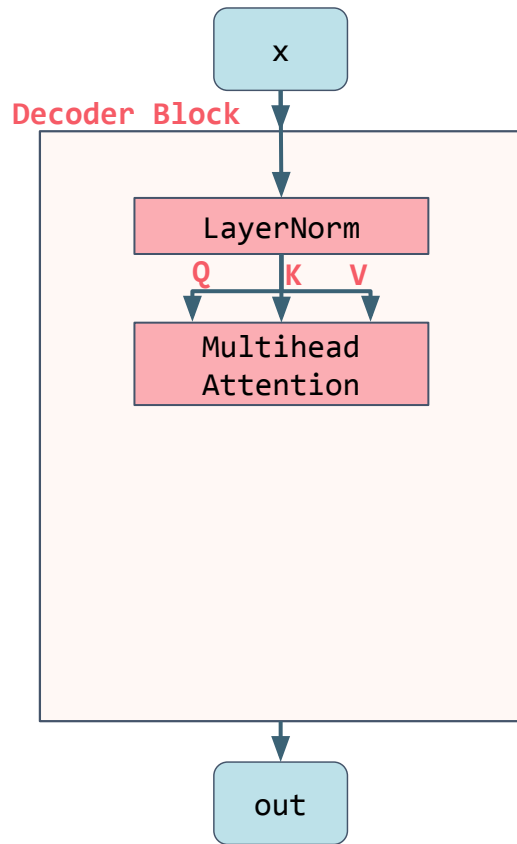
Predicting Tokens



Decoder Block

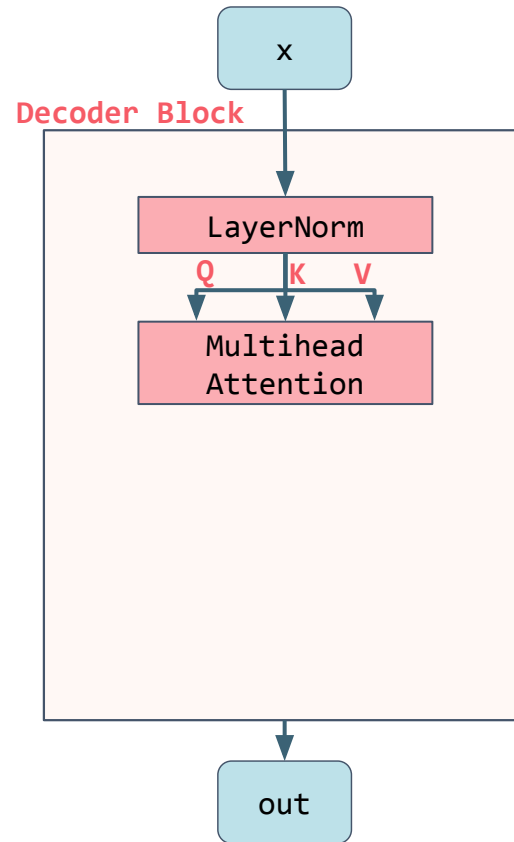
```
class DecoderBlock(nn.Module):  
    def __init__(self, d_model=4, nhead=1, ffn_mult=4):  
        super().__init__()
```

```
    def forward(self, x):
```



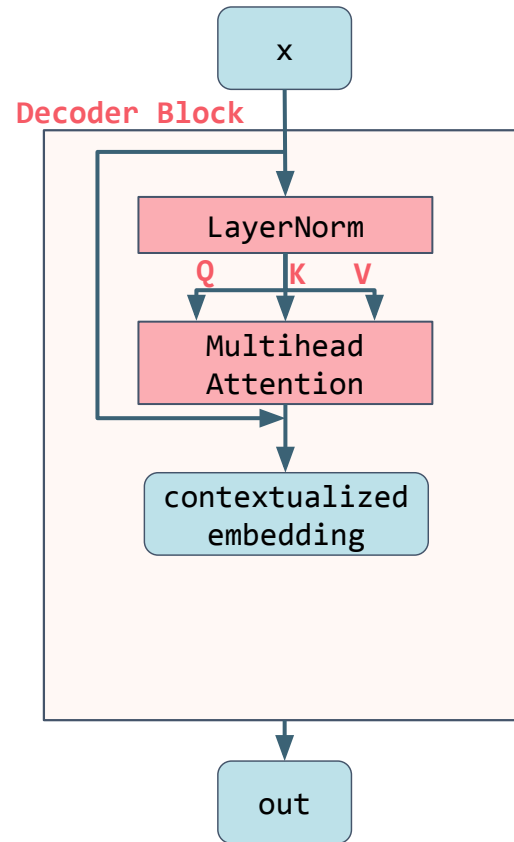
Decoder Block

```
class DecoderBlock(nn.Module):  
    def __init__(self, d_model=4, nhead=1, ffn_mult=4):  
        super().__init__()  
        self.ln1 = nn.LayerNorm(d_model)  
        self.mha = nn.MultiheadAttention(. . .)  
  
    def forward(self, x):  
        x_norm = self.ln1(x)  
        attn_out, _ = self.mha(x_norm, x_norm, x_norm)
```



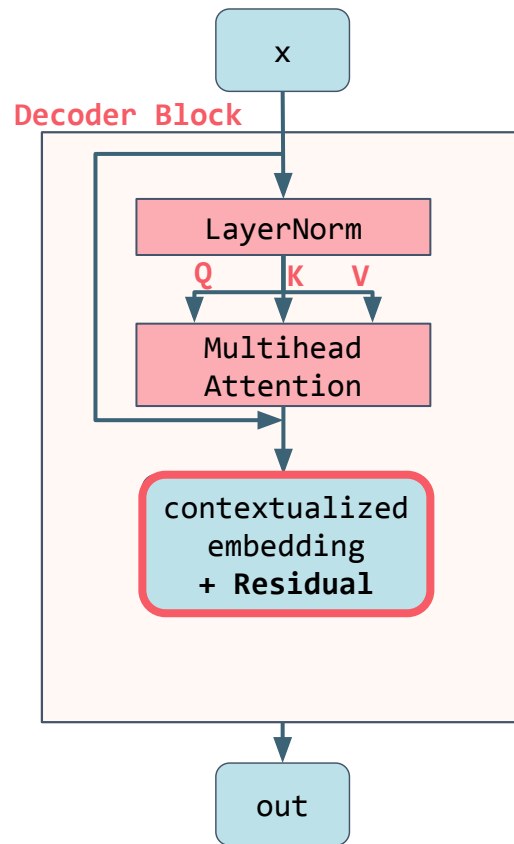
Decoder Block

```
class DecoderBlock(nn.Module):  
    def __init__(self, d_model=4, nhead=1, ffn_mult=4):  
        super().__init__()  
        self.ln1 = nn.LayerNorm(d_model)  
        self.mha = nn.MultiheadAttention(. . .)  
  
    def forward(self, x):  
        x_norm = self.ln1(x)  
        attn_out, _ = self.mha(x_norm, x_norm, x_norm)
```



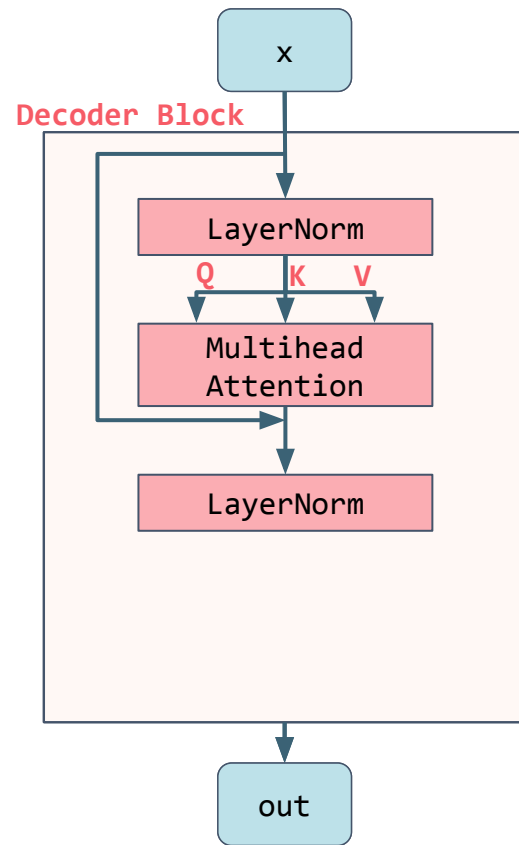
Decoder Block

```
class DecoderBlock(nn.Module):  
    def __init__(self, d_model=4, nhead=1, ffn_mult=4):  
        super().__init__()  
        self.ln1 = nn.LayerNorm(d_model)  
        self.mha = nn.MultiheadAttention(. . .)  
  
    def forward(self, x):  
        x_norm = self.ln1(x)  
        attn_out, _ = self.mha(x_norm, x_norm, x_norm)  
        x = x + attn_out
```



Decoder Block

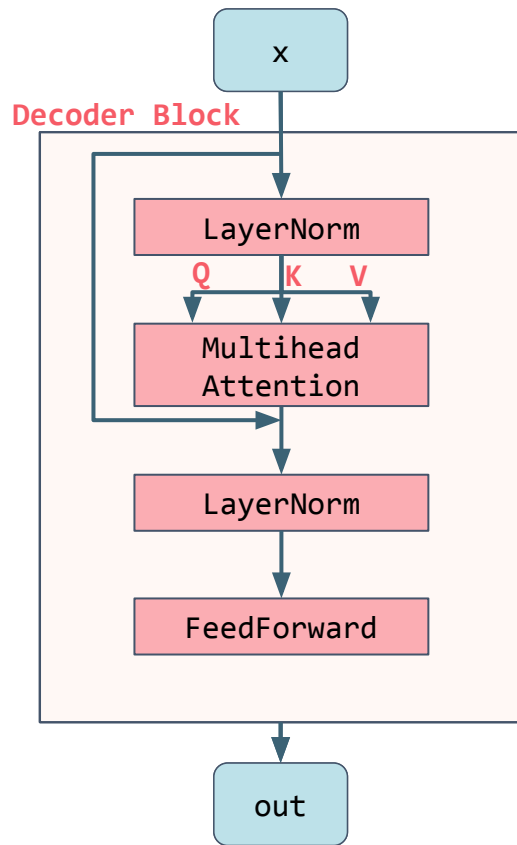
```
class DecoderBlock(nn.Module):  
    def __init__(self, d_model=4, nhead=1, ffn_mult=4):  
        super().__init__()  
        self.ln1 = nn.LayerNorm(d_model)  
        self.mha = nn.MultiheadAttention(. . .)  
        self.ln2 = nn.LayerNorm(d_model)  
  
    def forward(self, x):  
        x_norm = self.ln1(x)  
        attn_out, _ = self.mha(x_norm, x_norm, x_norm)  
        x = x + attn_out  
        ffn_in = self.ln2(x)
```



Decoder Block

```
class DecoderBlock(nn.Module):
    def __init__(self, d_model=4, nhead=1, ffn_mult=4):
        super().__init__()
        self.ln1 = nn.LayerNorm(d_model)
        self.mha = nn.MultiheadAttention(. . .)
        self.ln2 = nn.LayerNorm(d_model)
        hidden = ffn_mult * d_model
        self.ffn = nn.Sequential(. . .)

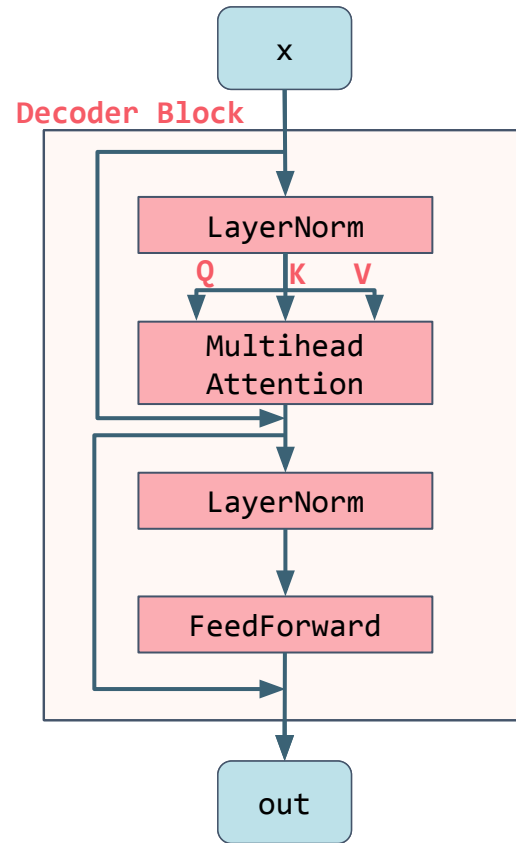
    def forward(self, x):
        x_norm = self.ln1(x)
        attn_out, _ = self.mha(x_norm, x_norm, x_norm)
        x = x + attn_out
        ffn_in = self.ln2(x)
        ffn_out = self.ffn(ffn_in)
```



Decoder Block

```
class DecoderBlock(nn.Module):
    def __init__(self, d_model=4, nhead=1, ffn_mult=4):
        super().__init__()
        self.ln1 = nn.LayerNorm(d_model)
        self.mha = nn.MultiheadAttention(. . .)
        self.ln2 = nn.LayerNorm(d_model)
        hidden = ffn_mult * d_model
        self.ffn = nn.Sequential(. . .)

    def forward(self, x):
        x_norm = self.ln1(x)
        attn_out, _ = self.mha(x_norm, x_norm, x_norm)
        x = x + attn_out
        ffn_in = self.ln2(x)
        ffn_out = self.ffn(ffn_in)
        x = x + ffn_out
        return x
```



Predicting Tokens



The cat chased the dog

Predicting Tokens

The cat chased **the** ?

Predicting Tokens

The cat chased **the** ?

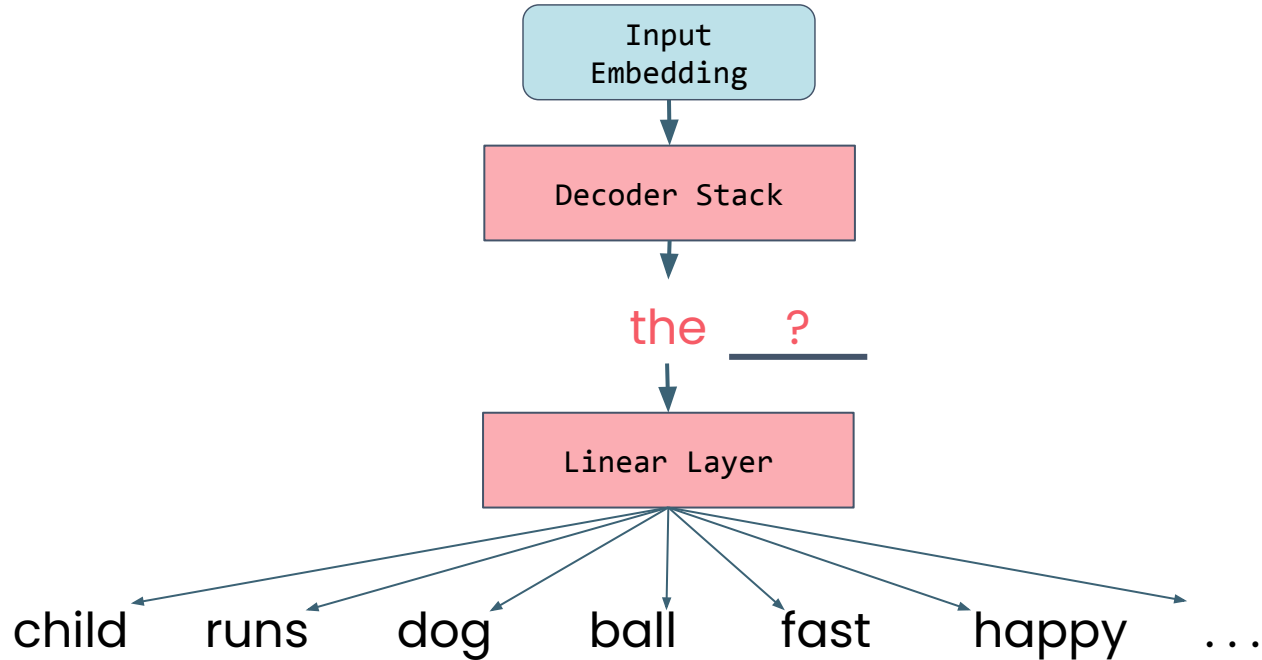
A diagram illustrating the task of predicting a missing token in a sequence. The text "The cat chased the ?" is shown, with "the" in red and a red question mark on a line. Three blue curved arrows point from the words "The", "cat", and "chased" to the question mark, indicating that the model uses the preceding context to predict the missing token.

Predicting Tokens

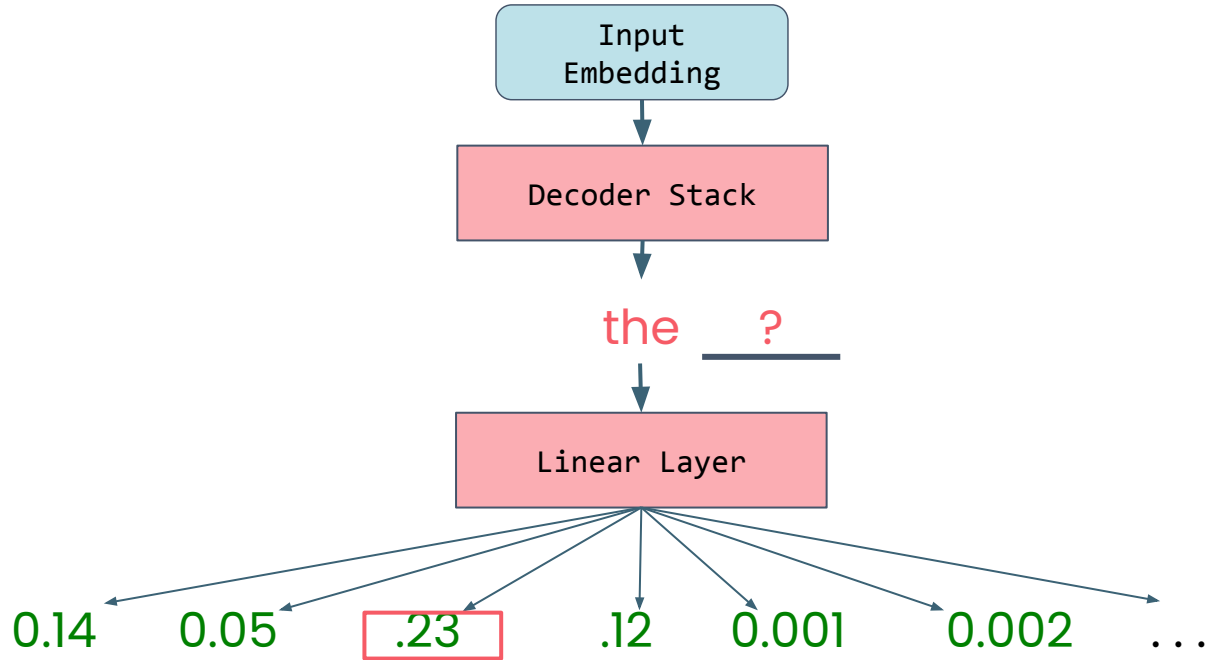
The cat chased the ?

A diagram illustrating a sequence prediction task. The text "The cat chased the" is followed by a red question mark on a red underline. Above the text, there are three overlapping blue arcs, each spanning two words: the first arc covers "The" and "cat", the second covers "cat" and "chased", and the third covers "chased" and "the".

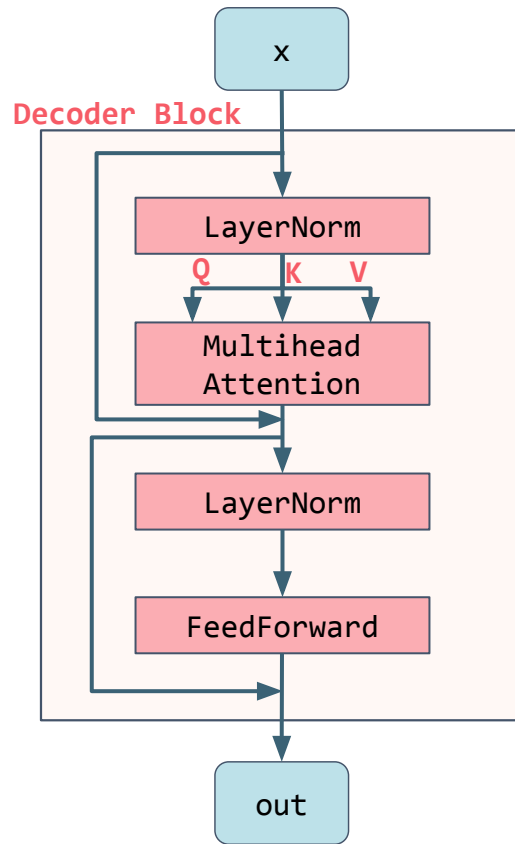
Predicting Tokens



Predicting Tokens



Decoder Block



How to Train a Decoder

Input

The cat chased the _____

When the lights went _____

I would have _____

She saw a _____

Targets

dog

out

left

stranger

How to Train a Decoder

The cat chased the dog

Input

The _____

The cat _____

The cat chased _____

The cat chased the _____

Targets

cat

chased

the

dog

Causal Masks

```
mha = nn.MultiheadAttention(. . .)
```

Causal Masks

```
mha = nn.MultiheadAttention(. . .)

def forward(self, x):

    # Custom masks
    out, _ = mha(attn_mask=custom_mask, . . .)
```

Causal Masks

```
mha = nn.MultiheadAttention(. . .)

def forward(self, x):

    # Custom masks
    out, _ = mha(attn_mask=custom_mask, . . .)

    # is_causal
    out, _ = mha(is_causal=True, . . .)
```

Causal Masks

Attention Scores

	the	cat	chased	the	dog
the	1.21	-0.259	-0.762	-0.995	0.054
cat	-0.216	0.604	-1.650	-1.061	1.876
chased	1.462	-0.410	-1.009	-0.990	0.822
the	1.506	-1.180	0.659	-1.810	0.521
dog	0.995	0.521	0.627	-0.511	0.315

Causal Masks

Attention Scores

	the	cat	chased	the	dog
the	1.21	-0.259	-0.762	-0.995	0.054
cat	-0.216	0.604	-1.650	-1.061	1.876
chased	1.462	-0.410	-1.009	-0.990	0.822
the	1.506	-1.180	0.659	-1.810	0.521
dog	0.995	0.521	0.627	-0.511	0.315

Causal Masks

Attention Scores

	the	cat	chased	the	dog
the	1.21	-0.259	-0.762	-0.995	0.054
cat	-0.216	0.604	-1.650	-1.061	1.876
chased	1.462	-0.410	-1.009	-0.990	0.822
the	1.506	-1.180	0.659	-1.810	0.521
dog	0.995	0.521	0.627	-0.511	0.315

Causal Masks

Attention Scores

	the	cat	chased	the	dog
the	1.21	-0.259	-0.762	-0.995	0.054
cat	-0.216	0.604	-1.650	-1.061	1.876
chased	1.462	-0.410	-1.009	-0.990	0.822
the	1.506	-1.180	0.659	-1.810	0.521
dog	0.995	0.521	0.627	-0.511	0.315

Causal Masks

Attention Scores

	the	cat	chased	the	dog
the	1.21	-0.259	-0.762	-0.995	0.054
cat	-0.216	0.604	-1.650	-1.061	1.876
chased	1.462	-0.410	-1.009	-0.990	0.822
the	1.506	-1.180	0.659	-1.810	0.521
dog	0.995	0.521	0.627	-0.511	0.315

Causal Masks

Attention Scores

		the	cat	chased	the	dog
“the	the	1.21	-0.259	-0.762	-0.995	0.054
	cat	-0.216	0.604	-1.650	-1.061	1.876
	chased	1.462	-0.410	-1.009	-0.990	0.822
	the	1.506	-1.180	0.659	-1.810	0.521
	dog	0.995	0.521	0.627	-0.511	0.315

Causal Masks

Attention Scores

		the	cat	chased	the	dog
“the cat	the	1.21	-0.259	-0.762	-0.995	0.054
	cat	-0.216	0.604	-1.650	-1.061	1.876
	chased	1.462	-0.410	-1.009	-0.990	0.822
	the	1.506	-1.180	0.659	-1.810	0.521
	dog	0.995	0.521	0.627	-0.511	0.315

Causal Masks

Attention Scores

	the	cat	chased	the	dog
the	1.21				
“the	cat	-0.216	0.604		
“the cat	chased	1.462	-0.410	-1.009	
“the cat chased	the	1.506	-1.180	0.659	-1.810
“the cat chased the	dog	0.995	0.521	0.627	-0.511
					0.315

Causal Masks

```
def make_causal_mask(sz: int):  
    mask = torch.full((sz, sz), float("-inf"))  
    mask = torch.triu(mask, diagonal=1)  
    return mask
```

0.	-inf	-inf	-inf	-inf
0.	0.	-inf	-inf	-inf
0.	0.	0.	-inf	-inf
0.	0.	0.	0.	-inf
0.	0.	0.	0.	0.

Causal Masks

Attention Scores

	the	cat	chased	the	dog
the	1.21	-0.259	-0.762	-0.995	0.054
cat	-0.216	0.604	-1.650	-1.061	1.876
chased	1.462	-0.410	-1.009	-0.990	0.822
the	1.506	-1.180	0.659	-1.810	0.521
dog	0.995	0.521	0.627	-0.511	0.315

+

0.	-inf	-inf	-inf	-inf
0.	0.	-inf	-inf	-inf
0.	0.	0.	-inf	-inf
0.	0.	0.	0.	-inf
0.	0.	0.	0.	0.

Causal Masks

Attention Scores

	the	cat	chased	the	dog
the	1.21	-inf	-inf	-inf	-inf
cat	-0.216	0.604	-inf	-inf	-inf
chased	1.462	-0.410	-1.009	-inf	-inf
the	1.506	-1.180	0.659	-1.810	-inf
dog	0.995	0.521	0.627	-0.511	0.315

Causal Masks

Apply softmax

	the	cat	chased	the	dog
the	1.21	-inf	-inf	-inf	-inf
cat	-0.216	0.604	-inf	-inf	-inf
chased	1.462	-0.410	-1.009	-inf	-inf
the	1.506	-1.180	0.659	-1.810	-inf
dog	0.995	0.521	0.627	-0.511	0.315

Causal Masks

Apply softmax

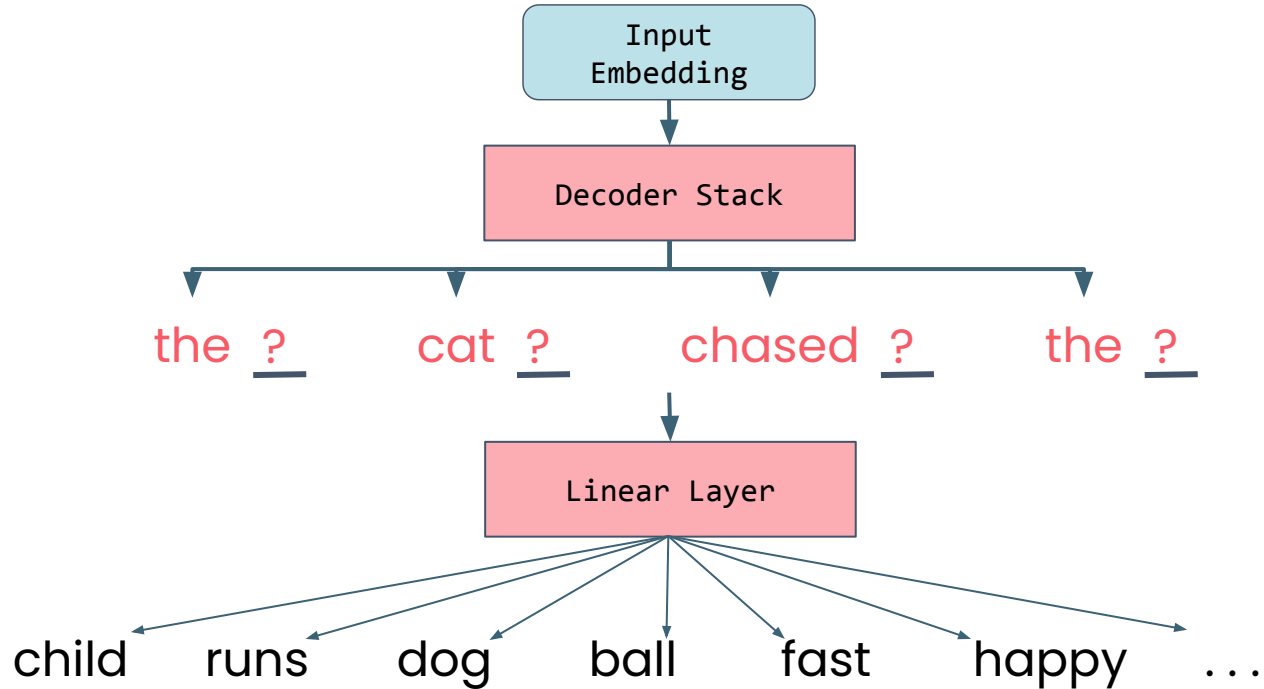
	the	cat	chased	the	dog
the	1.	0.	0.	0.	0.
cat	0.3058	0.6942	0.	0.	0.
chased	0.8075	0.1242	0.0682	0.	0.
the	0.6523	0.0445	0.2796	0.0237	0.
dog	0.3286	0.2046	0.2274	0.0729	0.1665

Causal Masks

Attention Weights

	the	cat	chased	the	dog
the	1.	0.	0.	0.	0.
cat	0.3058	0.6942	0.	0.	0.
chased	0.8075	0.1242	0.0682	0.	0.
the	0.6523	0.0445	0.2796	0.0237	0.
dog	0.3286	0.2046	0.2274	0.0729	0.1665

Predicting Tokens



Training Data

```
token_sequences = [['the', 'cat', 'chased', 'the', 'dog']] # Convert to token IDs
```

Training Data

```
token_sequences = [['the', 'cat', 'chased', 'the', 'dog']] # Convert to token IDs

# Inputs
inputs = token_sequences[:, :-1] # Inputs as strings ['the', 'cat', 'chased', 'the']
```

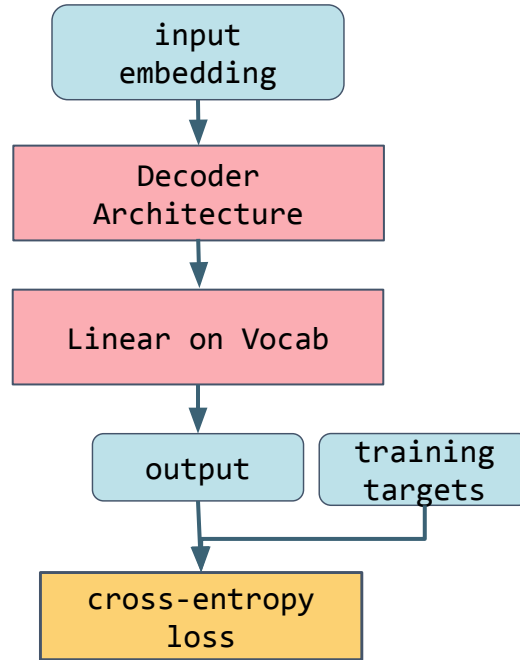
Training Data

```
token_sequences = [['the', 'cat', 'chased', 'the', 'dog']] # Convert to token IDs

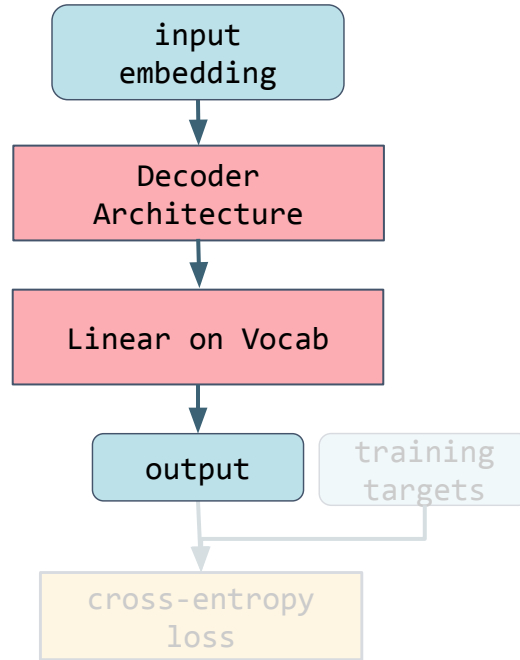
# Inputs
inputs = token_sequences[:, :-1] # Inputs as strings ['the', 'cat', 'chased', 'the']

# Targets
targets = token_sequences[:, 1:] # Targets as strings ['cat', 'chased', 'the', 'dog']
```

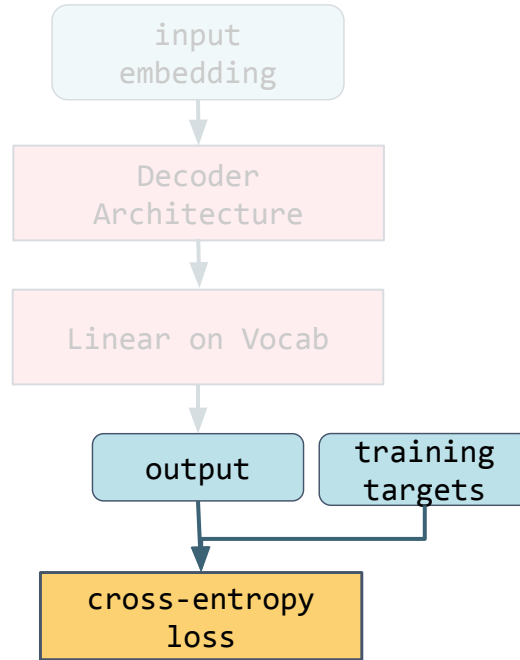
Training Data



Training Data



Training Data



Decoder Only Architectures

```
import torch.nn as nn  
  
nn.TransformerEncoderLayer  
nn.TransformerEncoder
```

Decoder Only Architectures

```
import torch.nn as nn
```

```
nn.TransformerEncoderLayer  
nn.TransformerEncoder
```

```
nn.TransformerDecoderLayer  
nn.TransformerDecoder
```

For encoder decoder models

Decoder Only Architectures

```
import torch.nn as nn

nn.TransformerEncoderLayer
nn.TransformerEncoder

nn.TransformerDecoderLayer
nn.TransformerDecoder
```

Decoder Only Architectures

```
import torch.nn as nn
```

```
nn.TransformerEncoderLayer + causal mask  
nn.TransformerEncoder
```

Decoder Only Architectures

```
import torch.nn as nn
```

```
nn.TransformerEncoderLayer  
nn.TransformerEncoder
```

```
out = encoder_layer (is_causal=True, . . .)  
out = encoder_layer (src_mask=custom_mask, . . .)
```

```
out = encoder_stack (is_causal=True, . . .)  
out = encoder_stack (mask=custom_mask, . . .)
```

Decoder Only Architectures

```
import torch.nn as nn

nn.TransformerEncoderLayer
nn.TransformerEncoder

out = encoder_layer (is_causal=True, . . .)
out = encoder_layer (src_mask=custom_mask, . . .)

out = encoder_stack (is_causal=True, . . .)
out = encoder_stack (mask=custom_mask, . . .)
```

Decoder Only Architectures

```
import torch.nn as nn

nn.TransformerEncoderLayer
nn.TransformerEncoder

out = encoder_layer (is_causal=True, . . .)
out = encoder_layer (src_mask=custom_mask, . . .)

out = encoder_stack (is_causal=True, . . .)
out = encoder_stack (mask=custom_mask, . . .)
```




DeepLearning.AI

Encoder - Decoder

Specialized Approaches to
Natural Language Processing in Pytorch

Transformer Architectures

- **Encoders:**
 - Analyze using rich contextual representations
 - Does **not** generate new tokens
- **Decoders:**
 - Can generate new text from old
 - But only by continuing from whatever it was given

Decoders as Chatbots



`<user> </user>`

`<assistant> </assistant>`

`<s> </s>`

`<user>` *did the cat chase the dog?* `</user><assistant>` . . .

Decoders as Chatbots



`<user> </user>`


`<assistant> </assistant>`

`<s> </s>`

`<user>` *did the cat chase the dog?* `</user>``<assistant>` *it depends on who you ask* `</assistant>`

Translation



The cat chased the dog ~~...~~ 


Il gatto ha inseguito il cane

<user> Translate "the cat chased the dog" into Italian</user><assistant> ...

Input

The cat chased the dog

Tokenization — 001 002 003 001 004



The diagram illustrates the process of tokenization. The input sentence "The cat chased the dog" is shown in italics. Below each word, a red arrow points down to a corresponding token. The tokens are "001", "002", "003", "001", and "004", which are aligned under "The", "cat", "chased", "the", and "dog" respectively. The word "the" is repeated, and its token "001" is also repeated. The word "dog" has a unique token "004". The label "Tokenization" is in bold blue text, followed by a horizontal line and the sequence of tokens.

Encoder

```
class Encoder(nn.Module):  
    def __init__():  
        super().__init__()  
  
    def forward(self, src):
```

Encoder

```
class Encoder(nn.Module):  
    def __init__():  
        super().__init__()  
  
    def forward(self, src):
```


Encoder

src

```
class Encoder(nn.Module):  
    def __init__():  
        super().__init__()
```

```
    def forward(self, src):  
        padding_mask = create_padding_mask(src)
```

Encoder

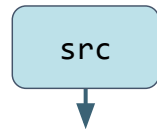
src

```
class Encoder(nn.Module):
    def __init__():
        super().__init__()
        self.token_emb = nn.Embedding(. . .)
        self.pos_enc = PositionalEncoding(. . .)

    def forward(self, src):
        padding_mask = create_padding_mask(src)
```

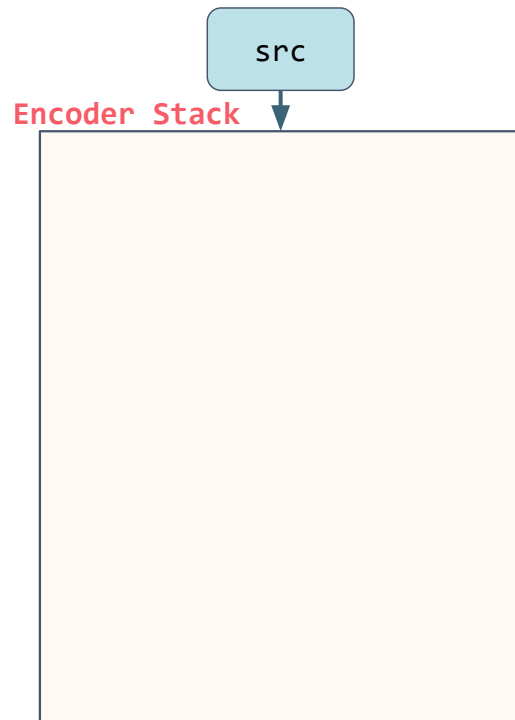
Encoder

```
class Encoder(nn.Module):  
    def __init__():  
        super().__init__()  
        self.token_emb = nn.Embedding(. . .)  
        self.pos_enc = PositionalEncoding(. . .)  
  
    def forward(self, src):  
        padding_mask = create_padding_mask(src)  
        src = self.token_emb(src) + self.pos_enc(src)
```



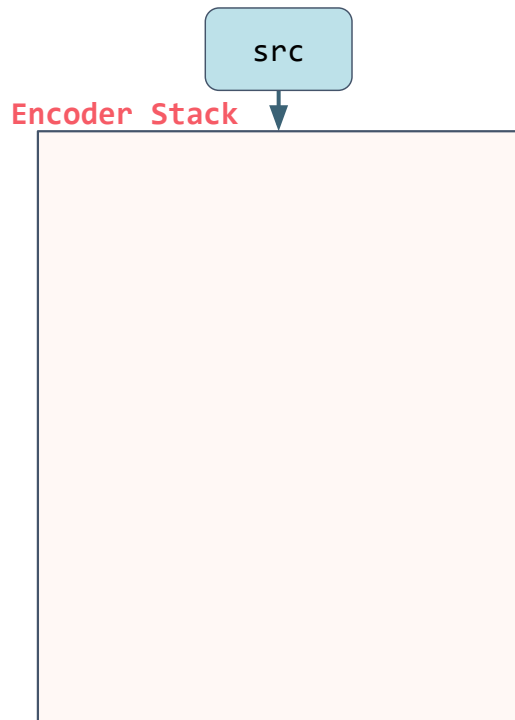
Encoder

```
class Encoder(nn.Module):  
    def __init__():  
        super().__init__()  
        . . .  
  
    def forward(self, src):  
        . . .
```



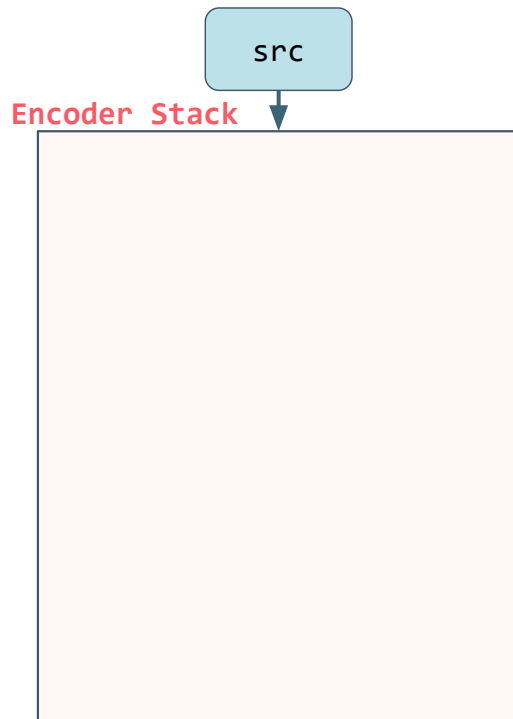
Encoder

```
class Encoder(nn.Module):  
    def __init__():  
        super().__init__()  
        . . .  
        enc_layer = nn.TransformerEncoderLayer(  
            d_model=d_model, nhead=nhead,  
            dim_feedforward=dim_feedforward,  
            dropout=dropout, batch_first=True)  
  
    def forward(self, src):  
        . . .
```



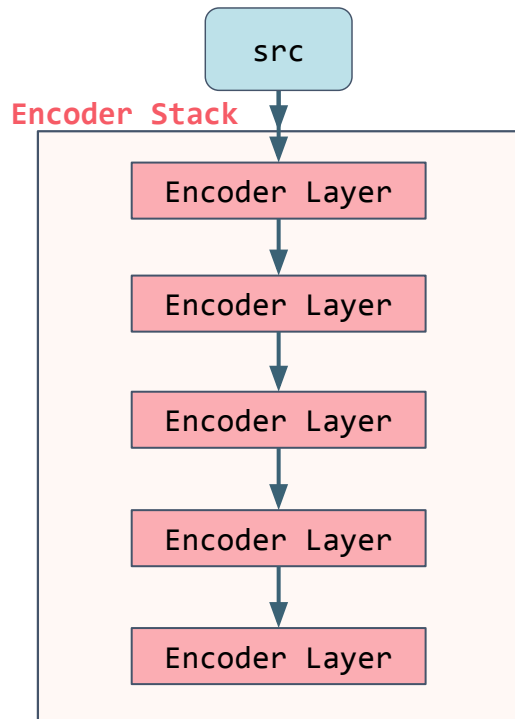
Encoder

```
class Encoder(nn.Module):  
    def __init__():  
        super().__init__()  
        . . .  
        enc_layer = nn.TransformerEncoderLayer(  
            d_model=d_model, nhead=nhead,  
            dim_feedforward=dim_feedforward,  
            dropout=dropout, batch_first=True)  
  
        self.transformer_encoder = nn.TransformerEncoder(  
            enc_layer,  
            num_layers=num_layers)  
    def forward(self, src):  
        . . .
```



Encoder

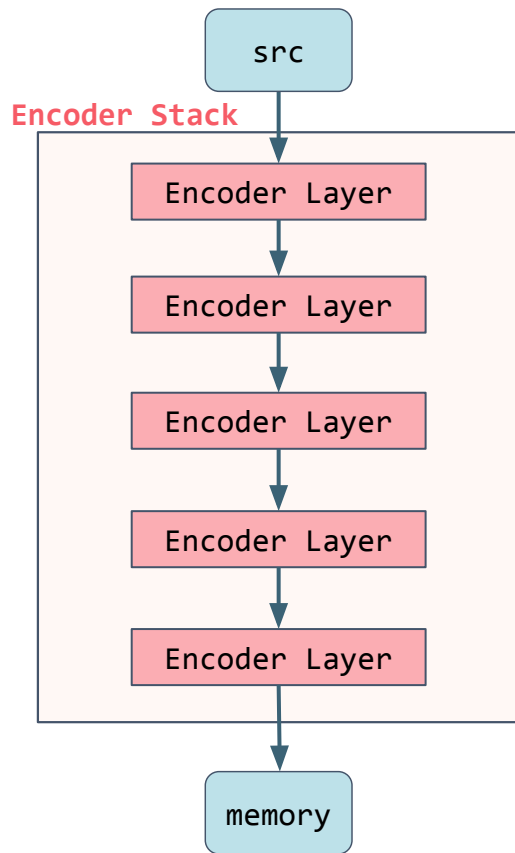
```
class Encoder(nn.Module):  
    def __init__():  
        super().__init__()  
        . . .  
        enc_layer = nn.TransformerEncoderLayer(  
            d_model=d_model, nhead=nhead,  
            dim_feedforward=dim_feedforward,  
            dropout=dropout, batch_first=True)  
  
        self.transformer_encoder = nn.TransformerEncoder(  
            enc_layer,  
            num_layers=num_layers)  
    def forward(self, src):  
        . . .
```



Encoder

```
class Encoder(nn.Module):
    def __init__():
        super().__init__()
        . . .
        enc_layer = nn.TransformerEncoderLayer(
            d_model=d_model, nhead=nhead,
            dim_feedforward=dim_feedforward,
            dropout=dropout, batch_first=True)

        self.transformer_encoder = nn.TransformerEncoder(
            enc_layer,
            num_layers=num_layers)
    def forward(self, src):
        . . .
        memory = self.transformer_encoder(src,
            src_key_padding_mask=padding_mask)
        return memory
```



Translation



The cat chased the dog



Il gatto ha inseguito il cane

Translation



The cat chased the dog

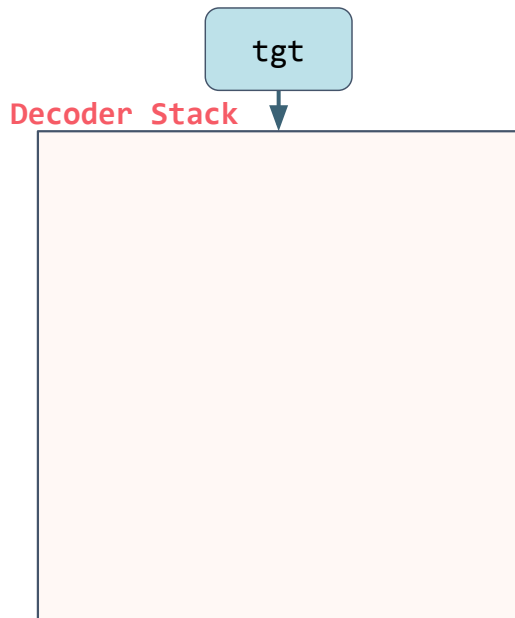


«S»



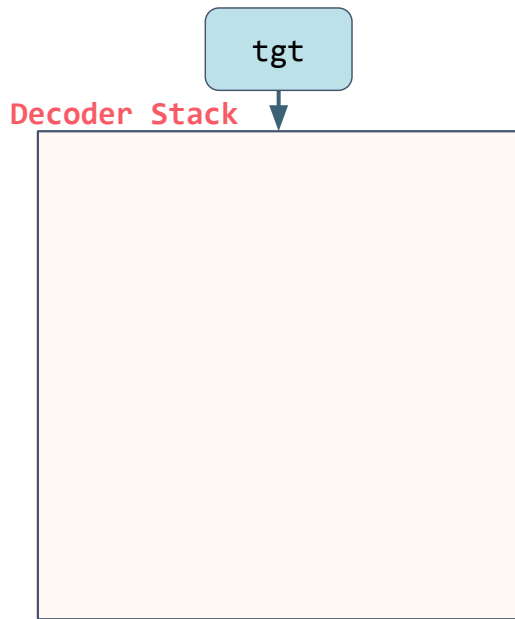
Decoder

```
class Decoder(nn.Module):  
    def __init__():  
        super().__init__()  
        self.token_emb = nn.Embedding(. . .)  
        self.pos_enc = PositionalEncoding(. . .)  
  
    def forward(self, tgt, memory, memory_padding_mask):  
        src_key_padding_mask = create_pad_mask(tgt)  
        src = self.src_tok(tgt) + self.src_pos(tgt)
```



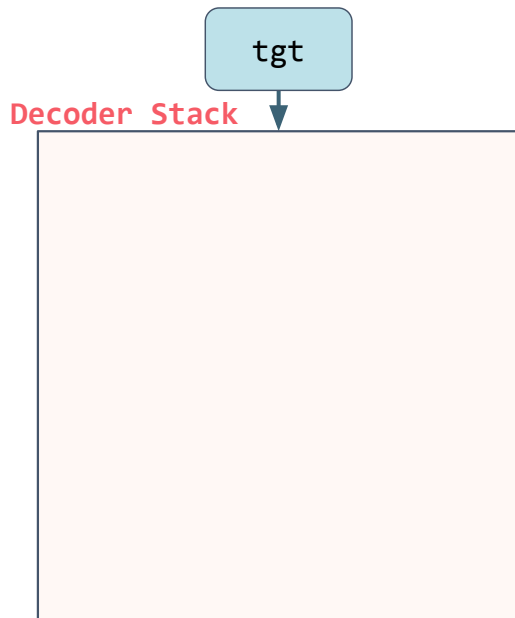
Decoder

```
class Decoder(nn.Module):  
    def __init__():  
        super().__init__()  
        self.token_emb = nn.Embedding(. . .)  
        self.pos_enc = PositionalEncoding(. . .)  
  
    def forward(self, tgt, memory, memory_padding_mask):  
        src_key_padding_mask = create_pad_mask(tgt)  
        src = self.src_tok(tgt) + self.src_pos(tgt)
```



Decoder

```
class Decoder(nn.Module):  
    def __init__():  
        super().__init__()  
        self.token_emb = nn.Embedding(. . .)  
        self.pos_enc = PositionalEncoding(. . .)  
  
    def forward(self, tgt, memory, memory_padding_mask):  
        src_key_padding_mask = create_pad_mask(tgt)  
        src = self.src_tok(tgt) + self.src_pos(tgt)
```



nn.DecoderLayer

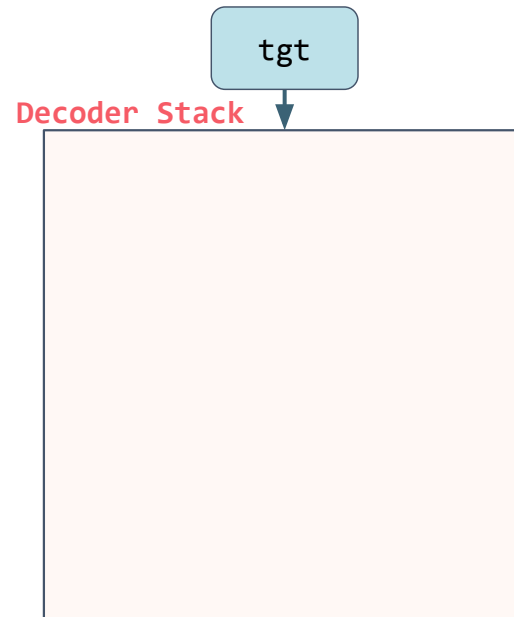
```
import torch.nn as nn
```

```
nn.TransformerDecoderLayer  
nn.TransformerDecoder
```

For encoder decoder models

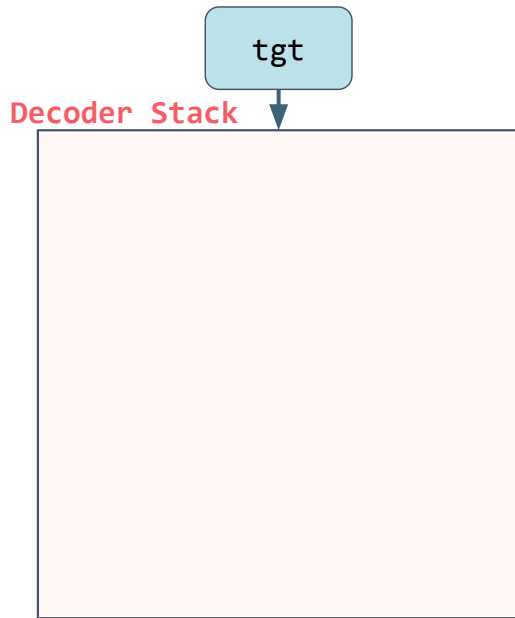
Decoder

```
class Decoder(nn.Module):  
    def __init__():  
        . . .  
  
    def forward(self, tgt, memory, memory_padding_mask):  
        . . .
```



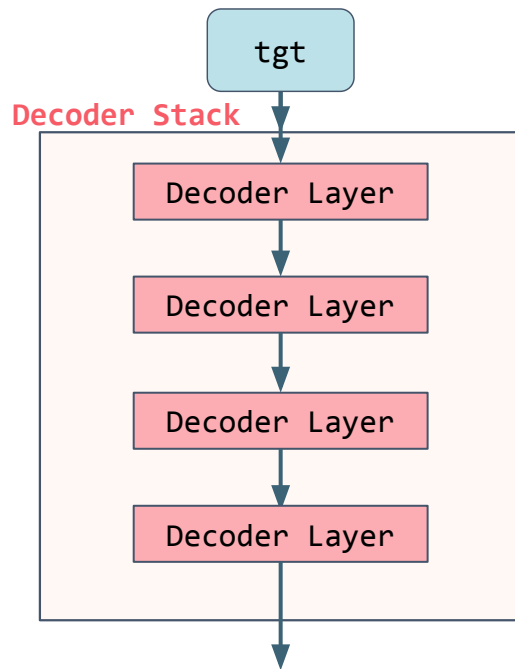
Decoder

```
class Decoder(nn.Module):  
    def __init__():  
        . . .  
        decoder_layer = nn.TransformerDecoderLayer(  
            d_model=d_model, nhead=nhead,  
            dim_feedforward=dim_feedforward,  
            dropout=dropout, batch_first=True)  
  
    def forward(self, tgt, memory, memory_padding_mask):  
        . . .
```



Decoder

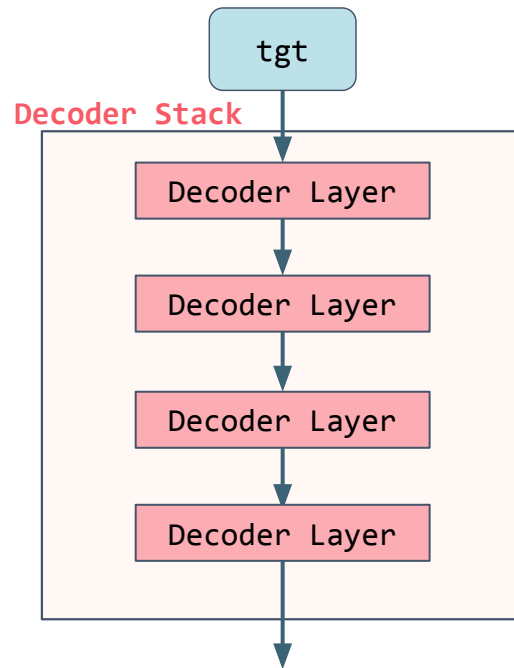
```
class Decoder(nn.Module):  
    def __init__():  
        . . .  
        decoder_layer = nn.TransformerDecoderLayer(  
            d_model=d_model, nhead=nhead,  
            dim_feedforward=dim_feedforward,  
            dropout=dropout, batch_first=True)  
  
        self.transformer_decoder = nn.TransformerDecoder(  
            decoder_layer,  
            num_layers=num_layers)  
  
    def forward(self, tgt, memory, memory_padding_mask):  
        . . .
```



Decoder

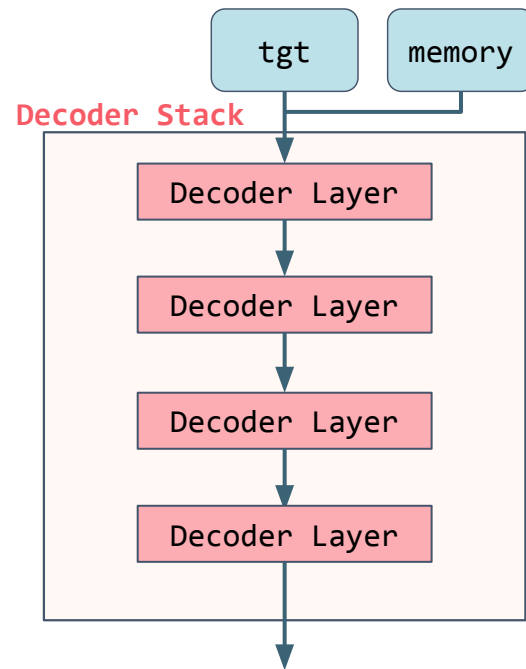
```
class Decoder(nn.Module):  
    def __init__():  
        . . .
```

```
def forward(self, tgt, memory, memory_padding_mask):  
    . . .  
    decoded= self.transformer_decoder(  
        tgt,  
        memory,  
        tgt_mask=tgt_causal_mask  
        memory_mask=None  
        tgt_key_padding_mask=tgt_padding_mask  
        memory_key_padding_mask=memory_padding_mask)
```



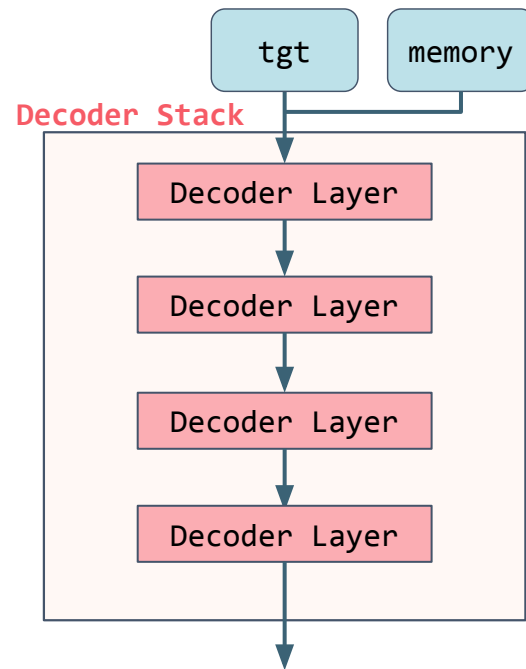
Decoder

```
class Decoder(nn.Module):  
    def __init__():  
        . . .  
  
    def forward(self, tgt, memory, memory_padding_mask):  
        . . .  
        decoded= self.transformer_decoder(  
            tgt,  
            memory,  
            tgt_mask=tgt_causal_mask  
            memory_mask=None  
            tgt_key_padding_mask=tgt_padding_mask  
            memory_key_padding_mask=memory_padding_mask)
```



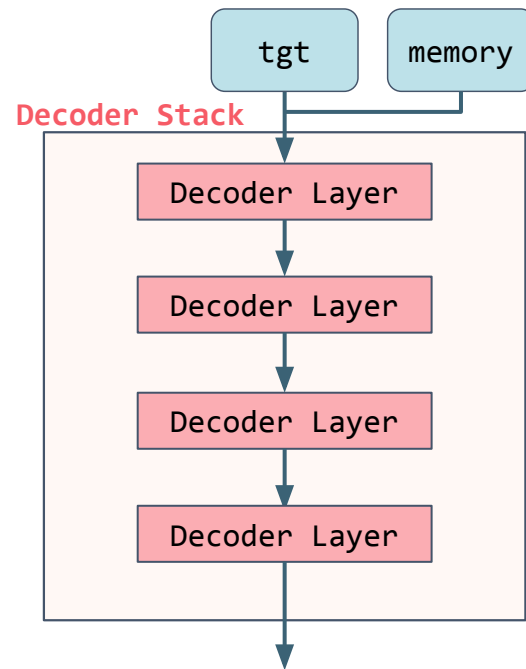
Decoder

```
class Decoder(nn.Module):  
    def __init__():  
        . . .  
  
    def forward(self, tgt, memory, memory_padding_mask):  
        . . .  
        decoded= self.transformer_decoder(  
            tgt,  
            memory,  
            tgt_mask=tgt_causal_mask  
            memory_mask=None  
            tgt_key_padding_mask=tgt_padding_mask  
            memory_key_padding_mask=memory_padding_mask)
```



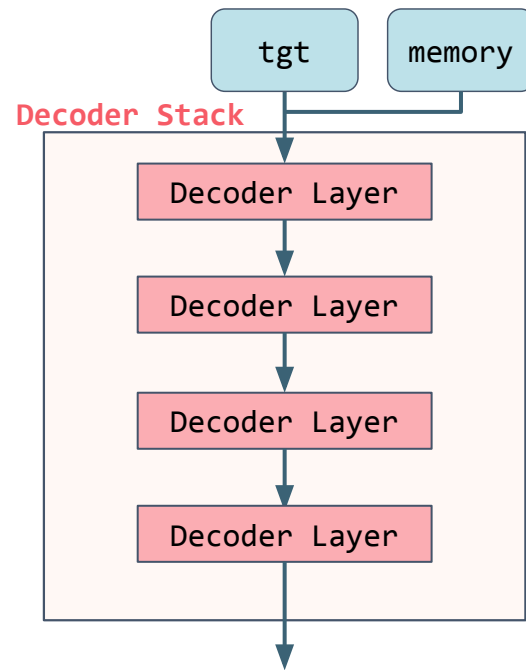
Decoder

```
class Decoder(nn.Module):  
    def __init__():  
        . . .  
  
    def forward(self, tgt, memory, memory_padding_mask):  
        . . .  
        decoded= self.transformer_decoder(  
            tgt,  
            memory,  
            tgt_mask=tgt_causal_mask  
            memory_mask=None  
            tgt_key_padding_mask=tgt_padding_mask  
            memory_key_padding_mask=memory_padding_mask)
```



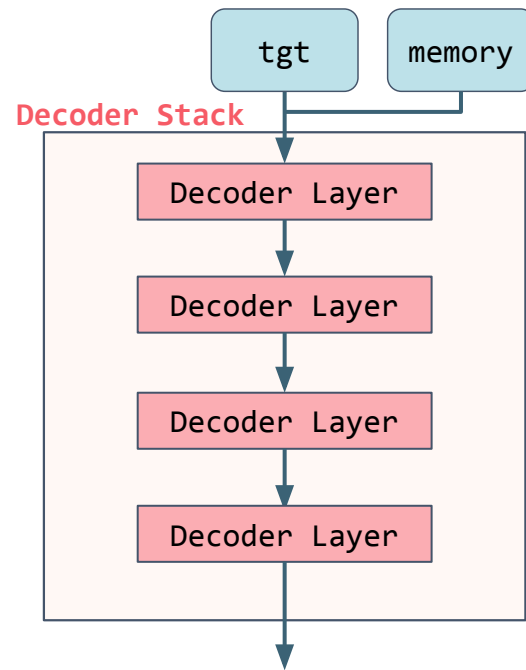
Decoder

```
class Decoder(nn.Module):  
    def __init__():  
        . . .  
  
    def forward(self, tgt, memory, memory_padding_mask):  
        . . .  
        decoded= self.transformer_decoder(  
            tgt,  
            memory,  
            tgt_mask=tgt causal mask  
            memory_mask=None  
            tgt_key_padding_mask=tgt_padding_mask  
            memory_key_padding_mask=memory_padding_mask)
```



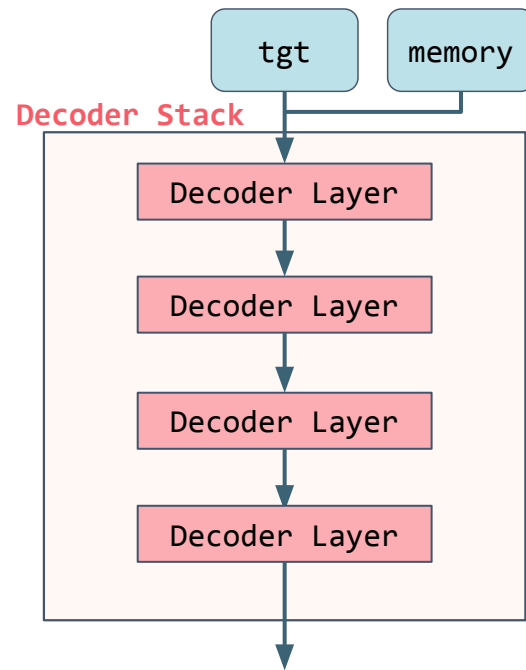
Decoder

```
class Decoder(nn.Module):  
    def __init__():  
        . . .  
  
    def forward(self, tgt, memory, memory_padding_mask):  
        . . .  
        decoded= self.transformer_decoder(  
            tgt,  
            memory,  
            tgt_mask=tgt_causal_mask  
            memory mask=None  
            tgt_key_padding_mask=tgt_padding_mask  
            memory_key_padding_mask=memory_padding_mask)
```



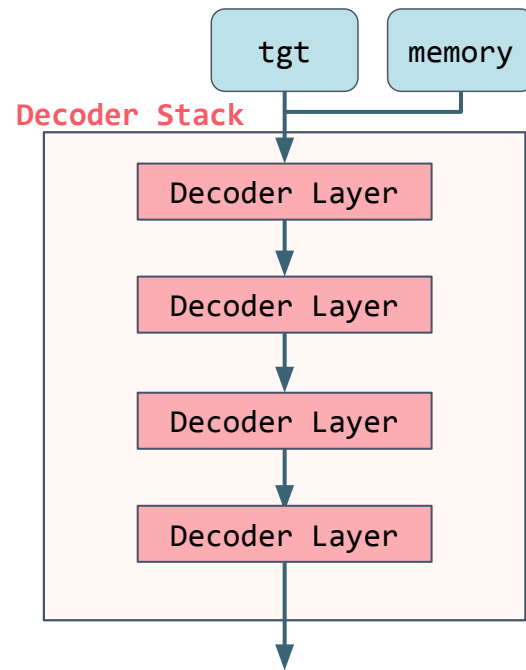
Decoder

```
class Decoder(nn.Module):  
    def __init__():  
        . . .  
  
    def forward(self, tgt, memory, memory_padding_mask):  
        . . .  
        decoded= self.transformer_decoder(. . .)
```



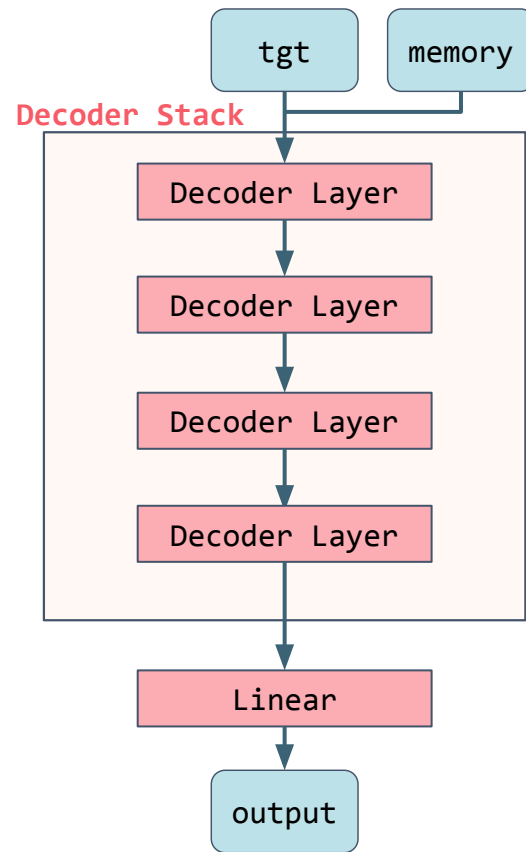
Decoder

```
class Decoder(nn.Module):  
    def __init__():  
        . . .  
        self.out_proj = nn.Linear(d_model, vocab_size)  
  
    def forward(self, tgt, memory, memory_padding_mask):  
        . . .  
        decoded= self.transformer_decoder(. . .)
```



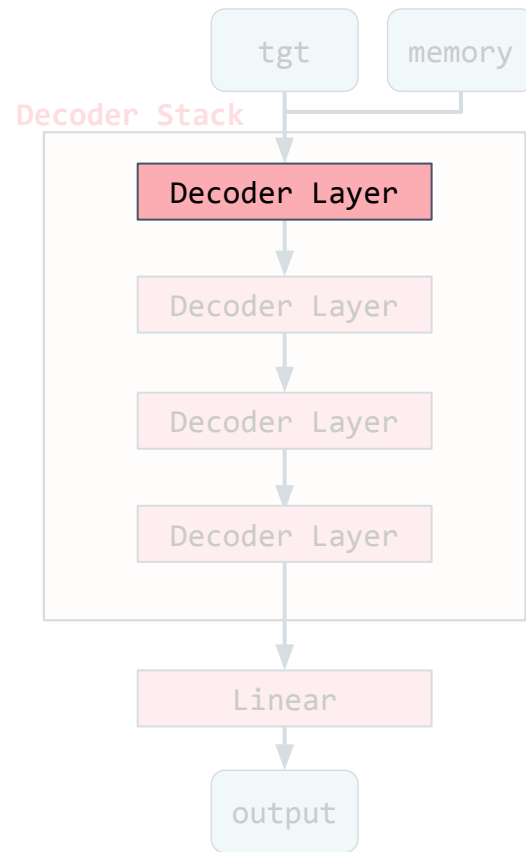
Decoder

```
class Decoder(nn.Module):  
    def __init__():  
        . . .  
        self.out_proj = nn.Linear(d_model, vocab_size)  
  
    def forward(self, tgt, memory, memory_padding_mask):  
        . . .  
        decoded= self.transformer_decoder(. . .)  
  
        output = self.output_proj(decoded)  
        return output
```

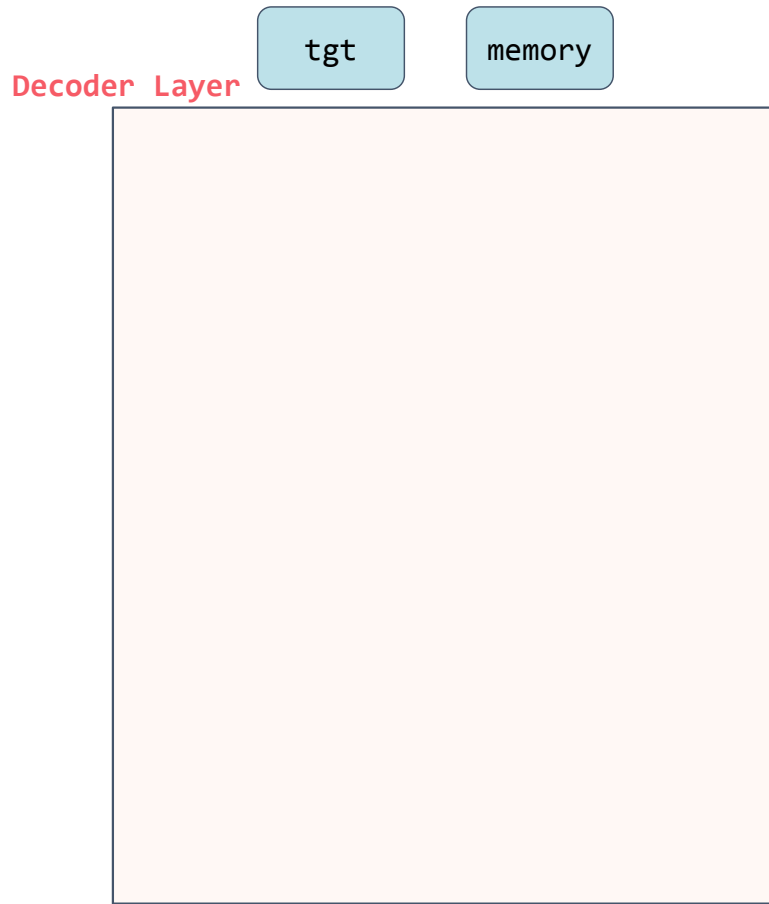


Decoder

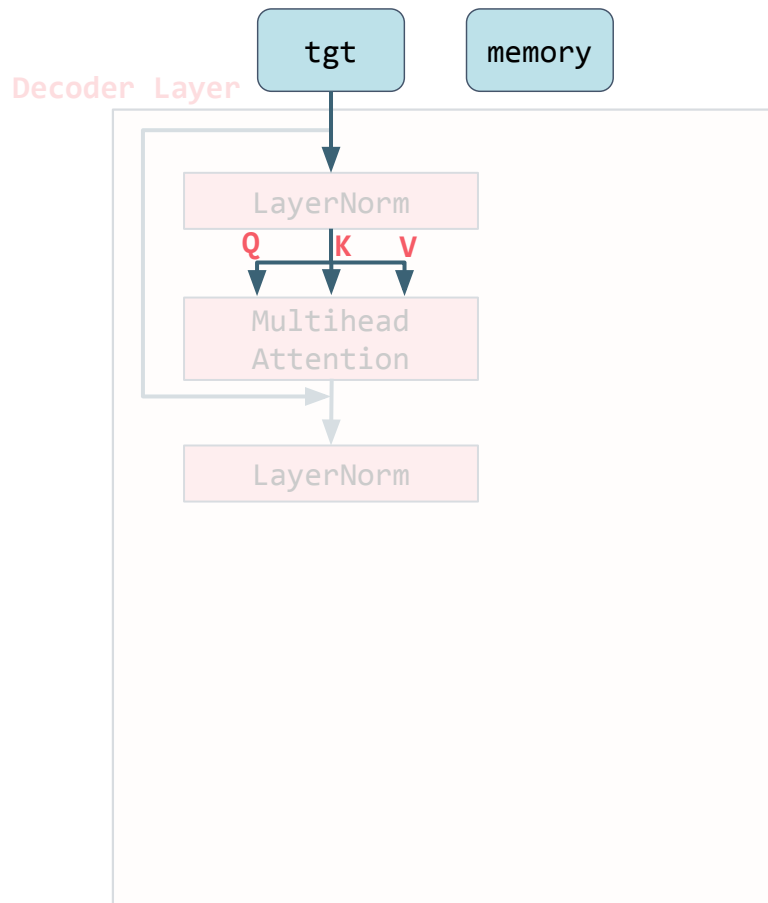
```
class Decoder(nn.Module):  
    def __init__():  
        . . .  
        self.out_proj = nn.Linear(d_model, vocab_size)  
  
    def forward(self, tgt, memory, memory_padding_mask):  
        . . .  
        decoded= self.transformer_decoder(. . .)  
  
        output = self.output_proj(decoded)  
  
        return output
```



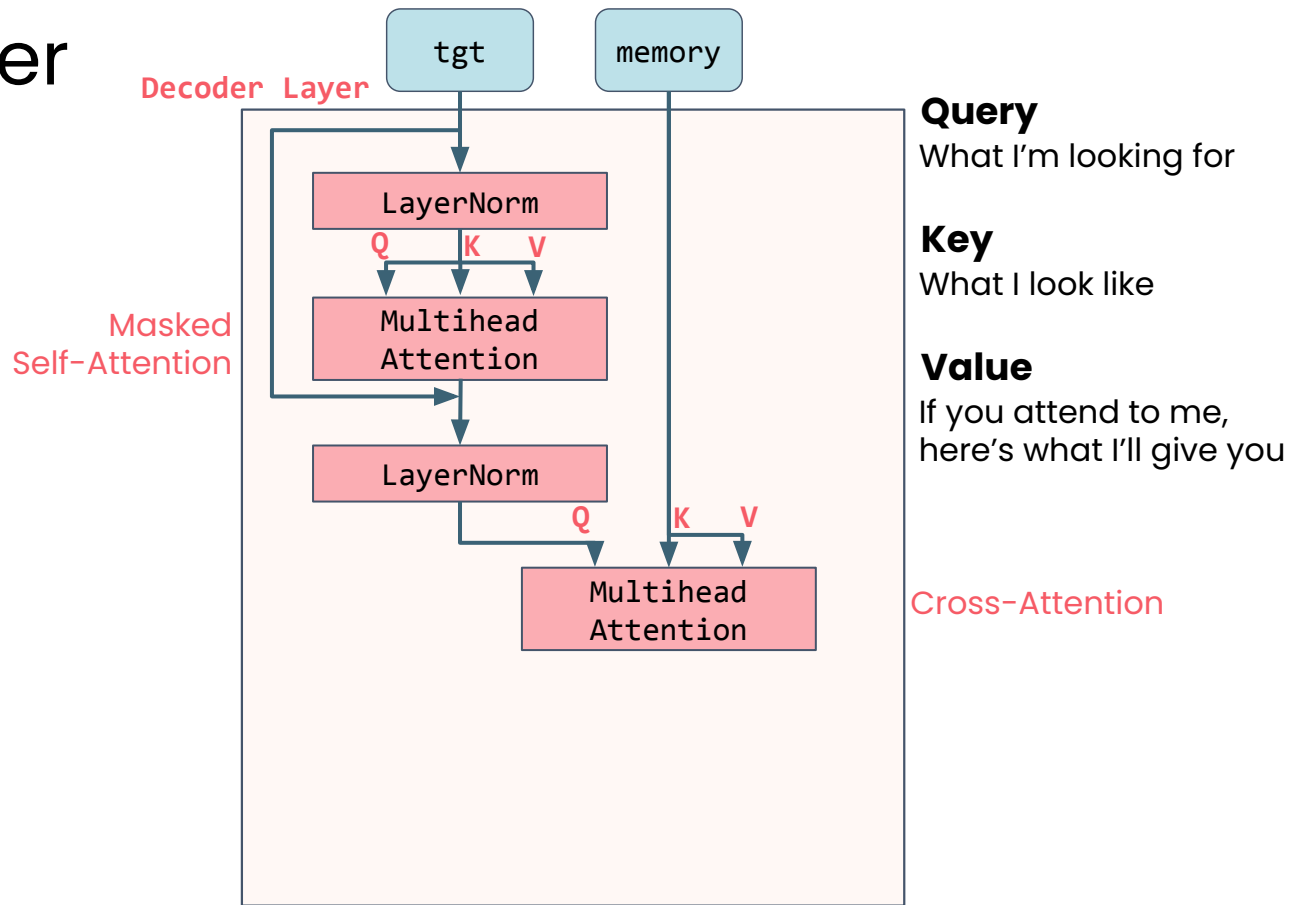
Decoder Layer



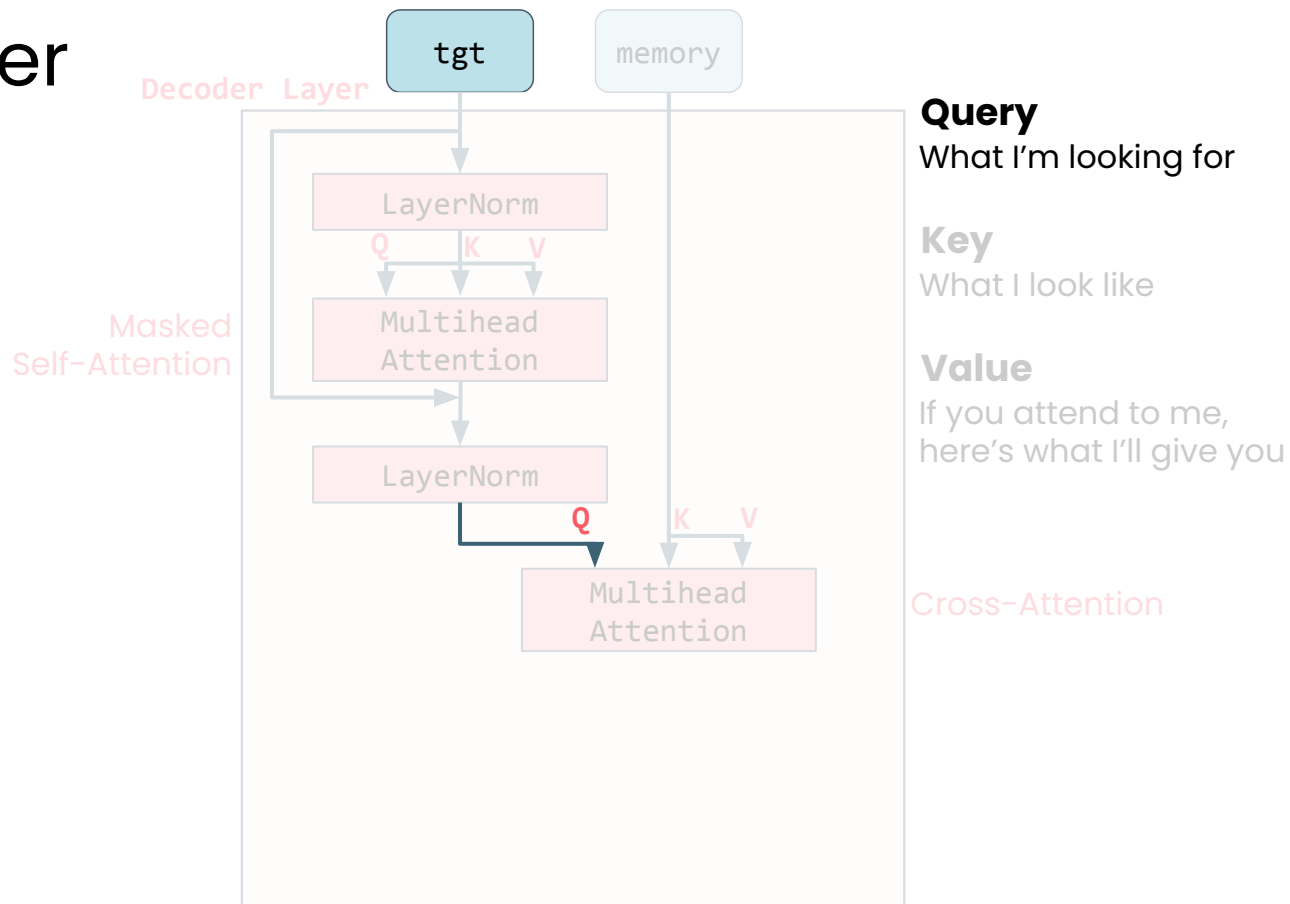
Decoder Layer



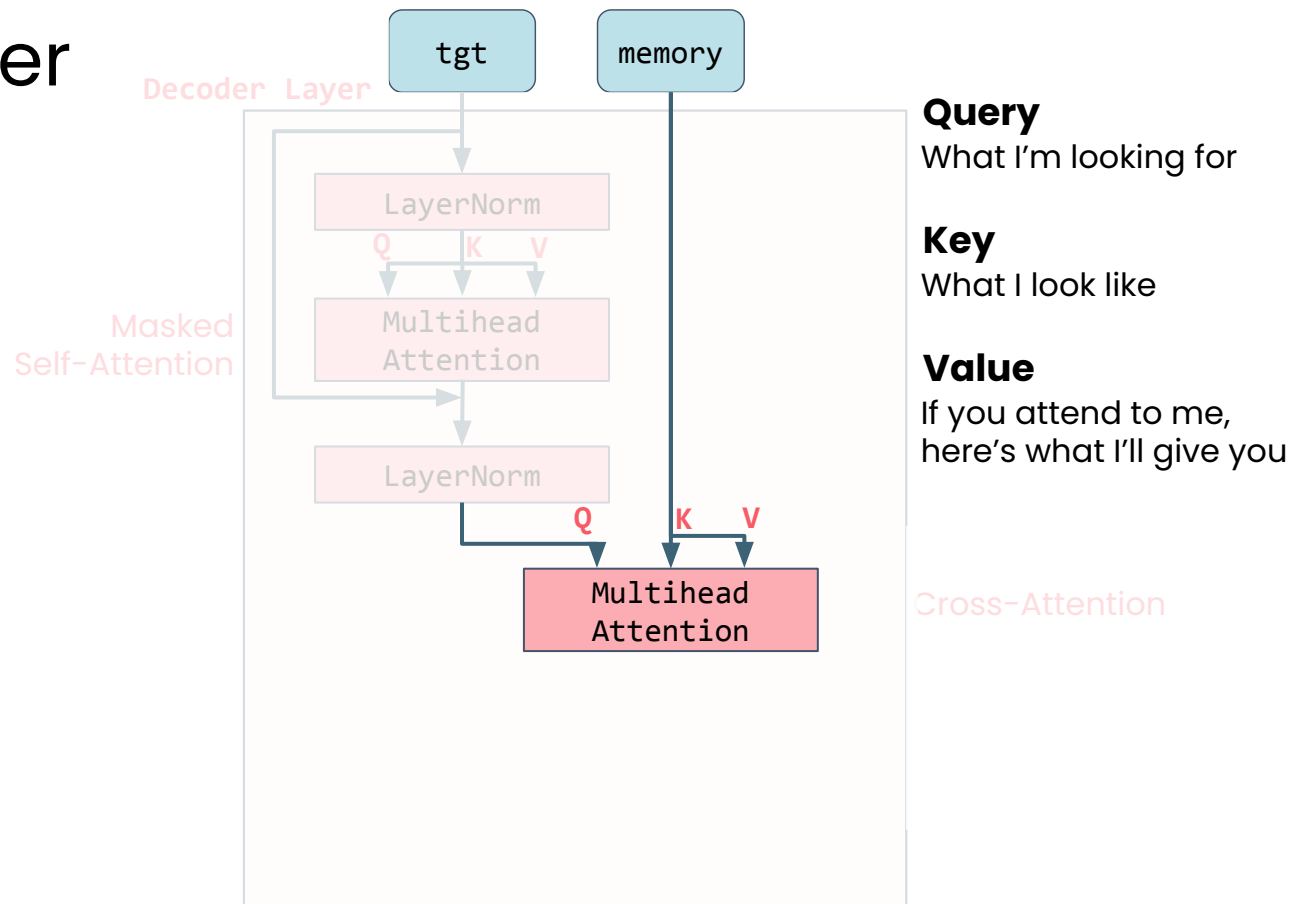
Decoder Layer



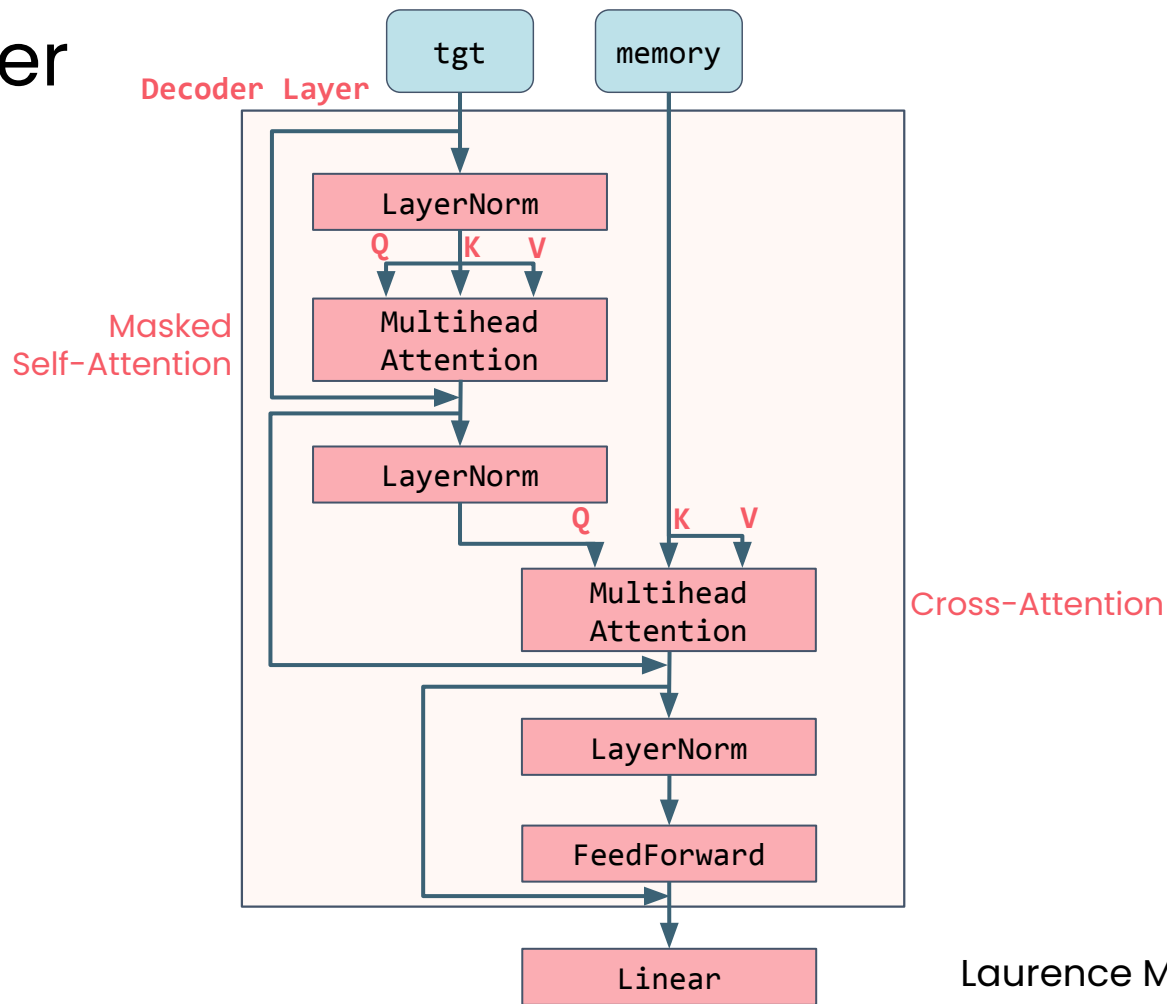
Decoder Layer



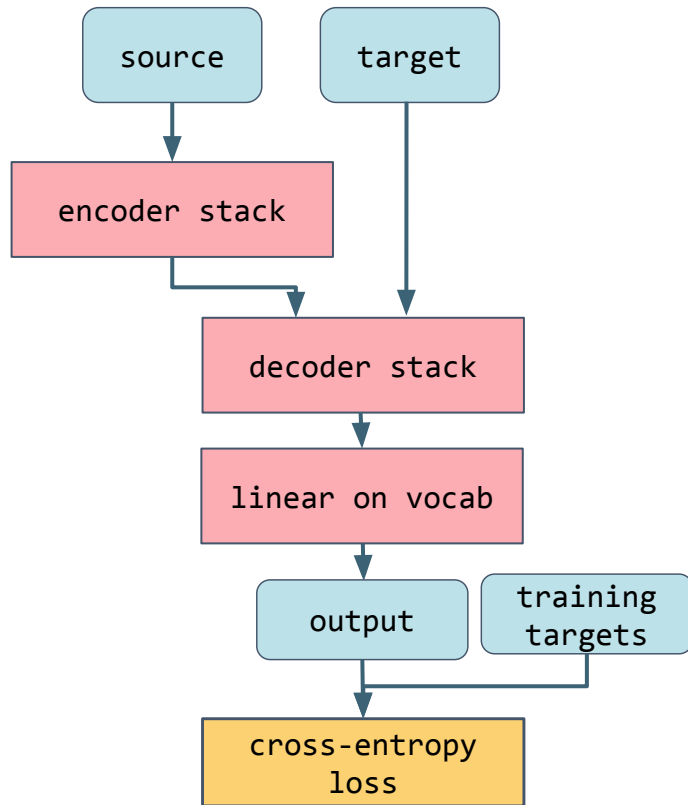
Decoder Layer



Decoder Layer



Encoder Decoder



Transformers

