Data 188: Introduction to Deep Learning

Transformers (Part 3), Visual Transformers

Speaker: Eric Kim
Lecture 18 (Week 12)
2026-04-07, Spring 2026. UC Berkeley.

Announcements

- HW3 continues: "Intro to Pytorch"

Today's lecture

Attention masks: padding, causal ("shift right")

Encoder-decoder train loss

Attention: quadratic cost

Visual Transformer ("ViT")

Sequence padding

Input sequences can have varying lengths:

```
["a", "shorter", "phrase"] -> [33, 8, 42]
["This", "is", "a", "longer", "phrase"] -> [9, 22, 33, 93, 42]
```

To "tensorize" these two sequences into a single tensor with shape=[B, n], we typically apply padding to a predetermined maximum sequence length, using a special [PAD] token:

```
# assume max_seq_len = 8, and [PAD] has token id 0
["a", "shorter", "phrase"] -> [33, 8, 42, 0, 0, 0, 0, 0]
["This", "is", "a", "longer", "phrase"] -> [9, 22, 33, 93, 42, 0, 0, 0]
```

Padding masks

To keep track of which sequence elements are valid (ie not [PAD] tokens), we create a **padding mask**, where `True` means invalid (ie [PAD]) token, and `False` means valid token:

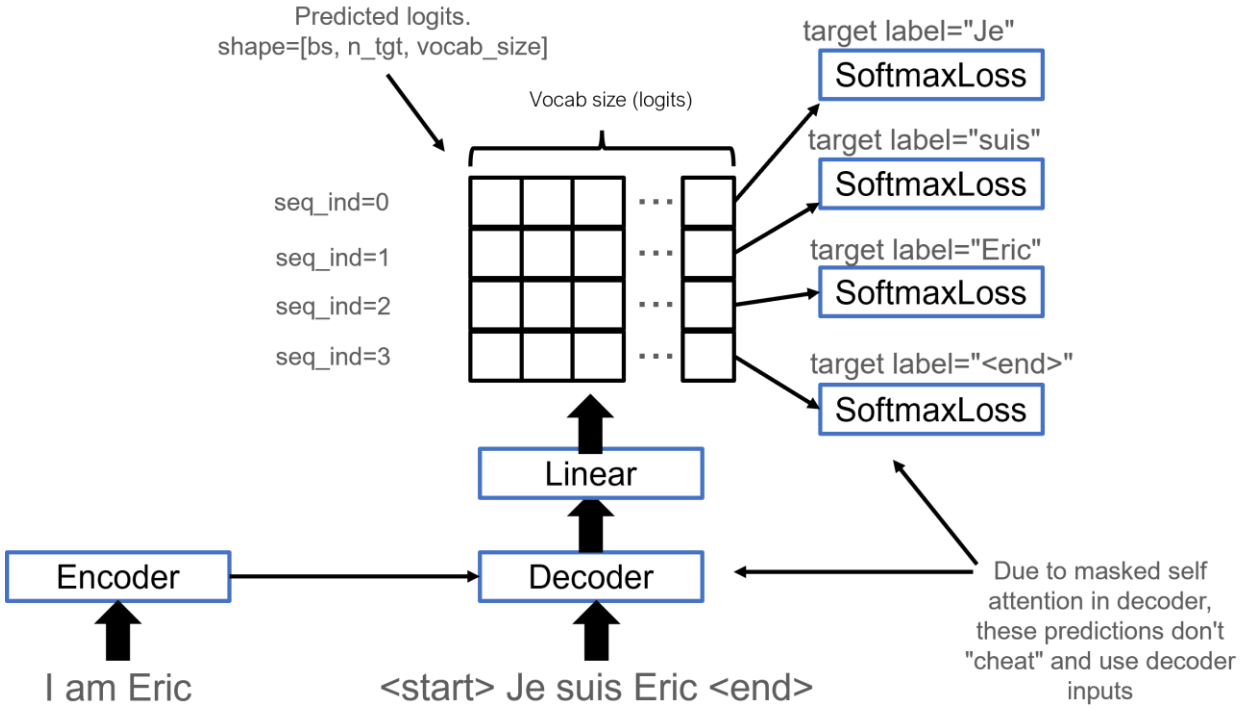
```
import torch
max_seq_len = 8
pad_id = 0
seq_tokens = torch.zeros(size=(2, max_seq_len), dtype=torch.int64)
seq_tokens[0, :] = torch.tensor([33, 8, 42, pad_id, pad_id, pad_id, pad_id, pad_id])
seq_tokens[1, :] = torch.tensor([9, 22, 33, 93, 42, pad_id, pad_id, pad_id])

pad_mask = torch.zeros(size=(2, max_seq_len), dtype=torch.bool)
pad_mask[0, :] = torch.tensor([False, False, False, True, True, True, True, True], dtype=torch.bool)
pad_mask[1, :] = torch.tensor([False, False, False, False, False, True, True, True], dtype=torch.bool)
# equivalently:
pad_mask_2 = (seq_tokens == pad_id)
```

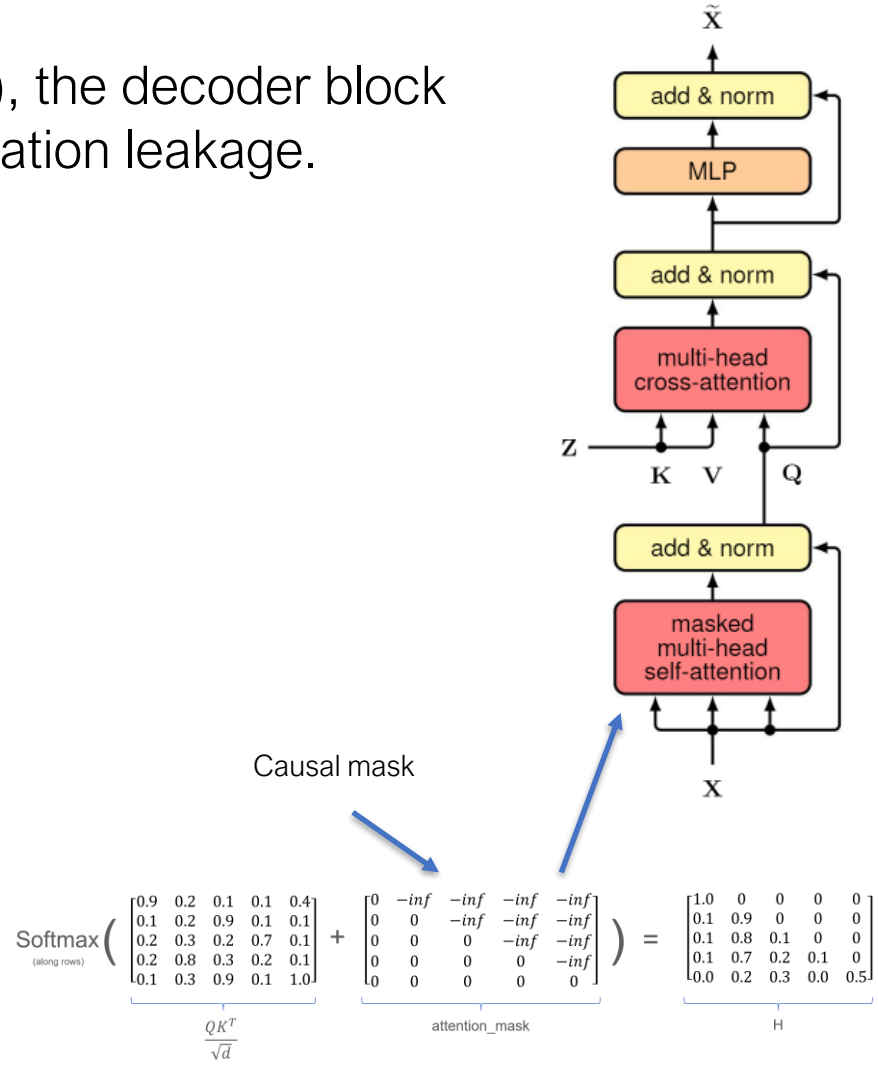
Most transformer functions accept both the input sequence and a corresponding padding mask.
Ex: `torch.nn.MultiheadAttention::forward(query, key, value, key_padding_mask)`.

MHA uses `key_padding_mask` to perform row-wise softmax only over valid tokens (skip the [PAD] tokens!).

Causal masks

Recall: for encoder+decoder models (ex: machine translation), the decoder block uses **causal masks** in its masked self attention to avoid information leakage.





Decoder: shift right

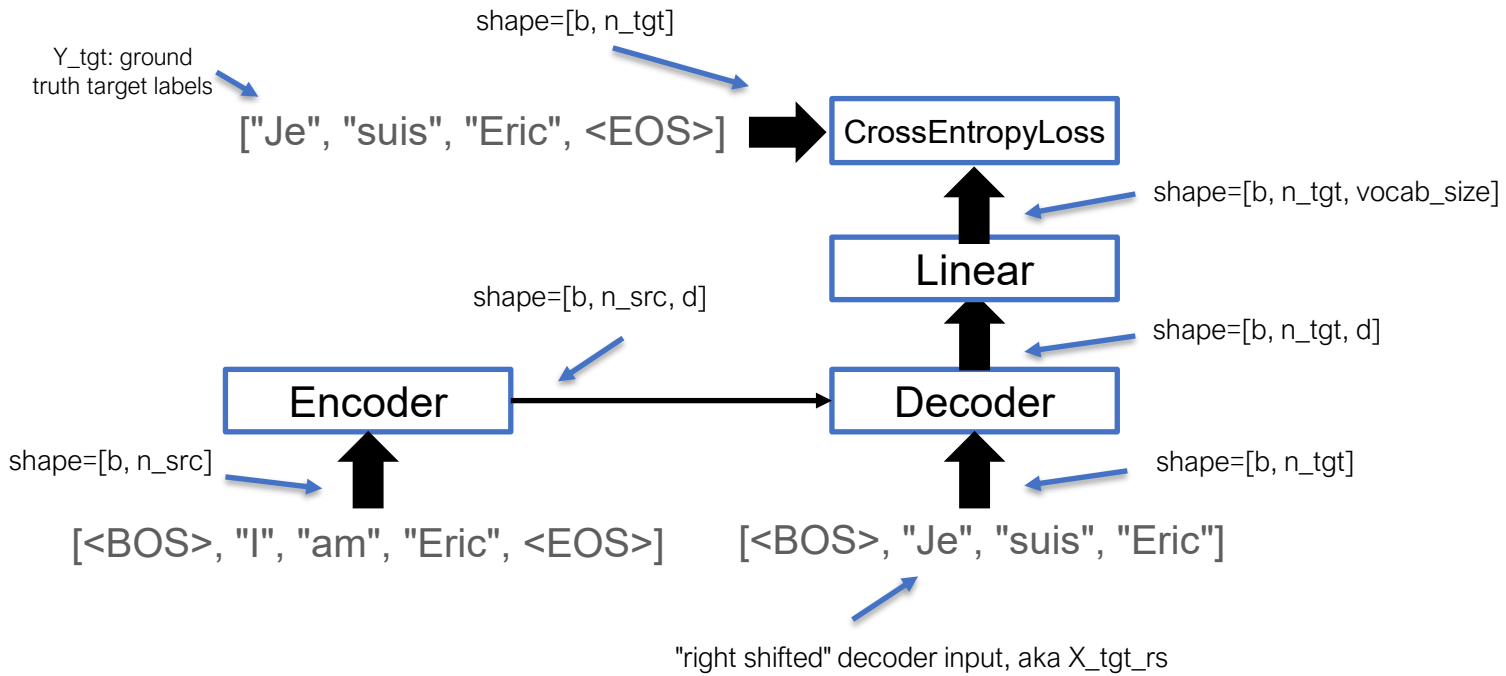
Most implementations will shift the target sequence right one token (prepend a [PAD] or [BOS] token), and use this right-shifted sequence as input to the decoder:

X_src = [<BOS>, "I", "am", "Eric", <END>]

Y_tgt = ["Je", "suis", "Eric", <END>]

X_tgt_rs = [<BOS>, "Je", "suis", "Eric"]

Tip: [BOS] means "beginning of sequence", [EOS] means "end of sequence"



With right-shifting, the next-token prediction task is organized nicely: decoder output position i predicts Y_tgt[:, i] (with causal masking to avoid information leakage))

```
Y_tgt = ["Je", "suis", "Eric", <EOS>]

X_tgt_rs = [<BOS>, "Je", "suis", "Eric"]
```

Given [<BOS>], predict: "Je"

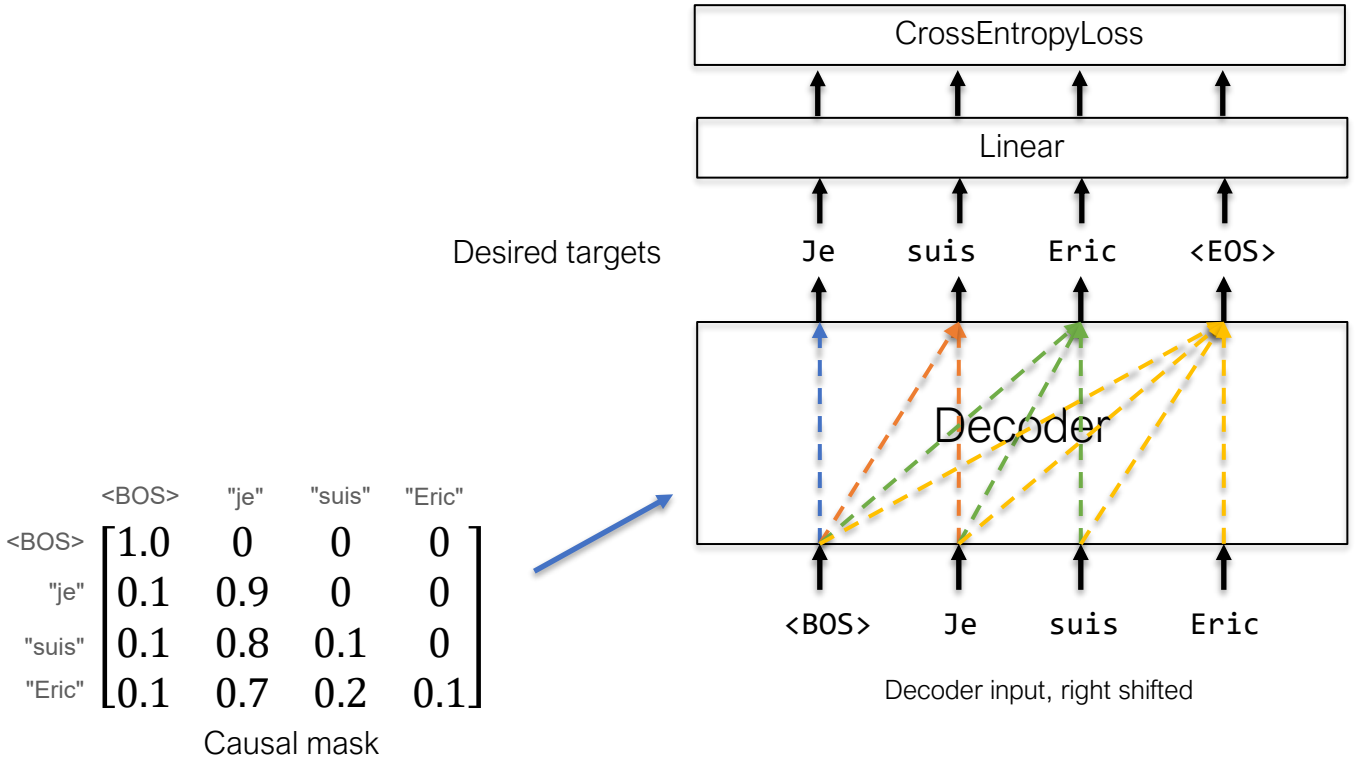
Given [<BOS>, "Je"], predict: "suis"

Given [<BOS>, "Je", "suis"], predict: "Eric"

Given [<BOS>, "Je", "suis", "Eric"], predict: <EOS>

Shift right + causal masks

With right-shifting, the next-token prediction task is organized nicely: decoder output position i predicts Y_tgt[:, i] (with causal masking to avoid information leakage):

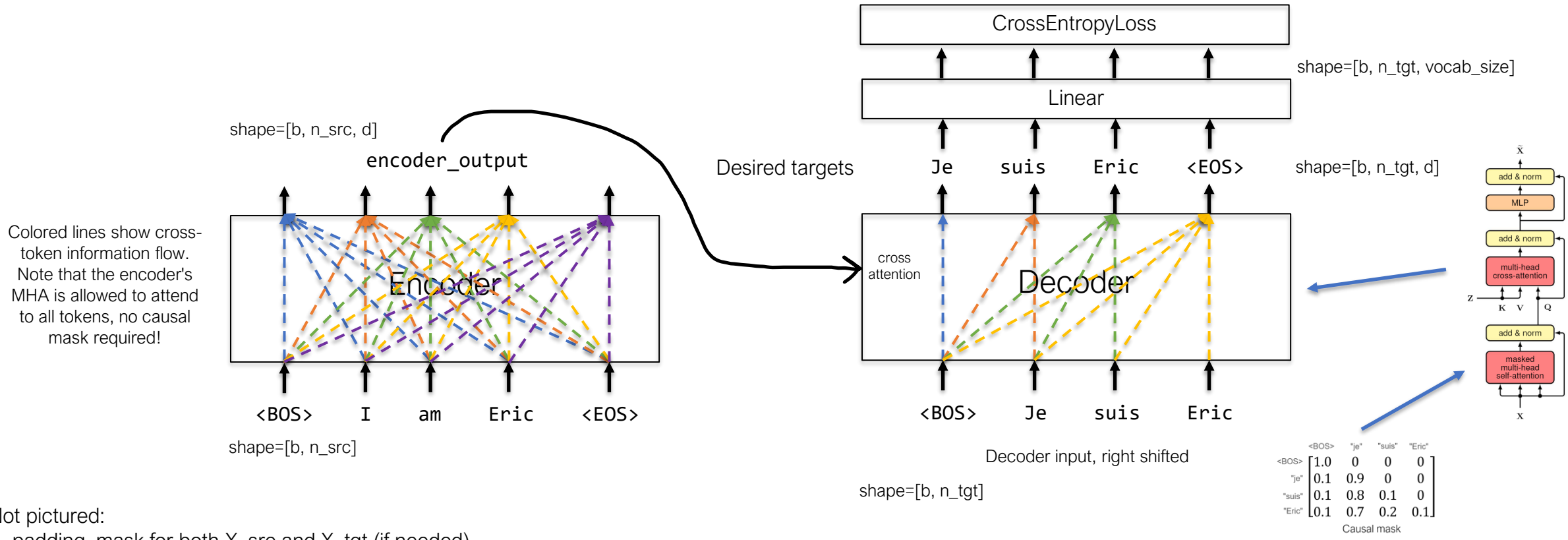


Colored lines show allowed cross-token information flow, enforced by causal mask in masked self attention block (to avoid information leakage)

Ex: when predicting "suis", decoder can only use information from: [<BOS>, "Je"]

Encoder+decoder train loss

Pictured: train loss for a sequence-to-sequence model (ex: machine translation).



Not pictured:

- padding_mask for both X_src and X_tgt (if needed)
- Position encoding
- Tokenizer and token embedding table

Transformer: information flow

Aside from MHA, all operations on token embeddings are done point-wise:

MHA: learn cross-token interactions

The rest (Linear, residual connection, FFN, LayerNorm): learn per-token transformations ("point-wise")

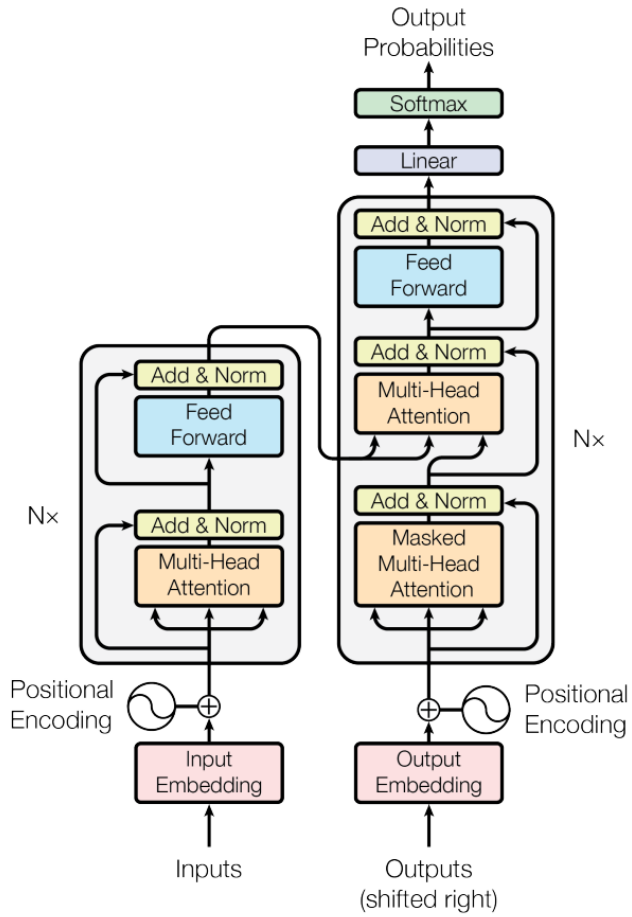


Figure 1: The Transformer - model architecture.

Attention: quadratic cost

MHA computation bottleneck: calculating $softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$

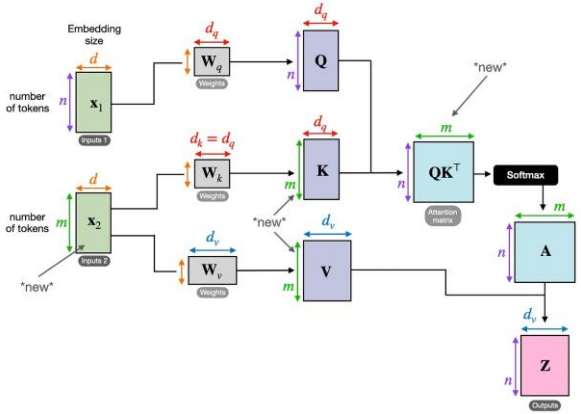
For Q.shape=[n, d], K.shape=[m, d]:

Time and memory complexity: O(n*m*d)

$\Rightarrow$ **Cost is quadratic in terms of sequence length.**

Ex: doubling sequence length leads to 4x longer compute and memory usage.

Bad news for LLM's that want very long context windows (aka longer sequence lengths)!



https://magazine.sebastianraschka.com/p/understanding-and-coding-self-attention

Flash Attention (2022)

As of 2026, FlashAttention (2022) is a popular algorithm that reduces memory usage to O(n + m), and accelerates MHA by 7.6x, enabling use of longer sequences.

Key idea: naive MHA implementations don't effectively utilize compute hardware, such as GPU's. Rewrite MHA to better utilize compute hardware ("IO aware": slow HBM vs fast SRAM)

Takeaway: modern large-scale deep learning requires performant implementations, often optimizing for specific hardware (ex: GPU's). Requires strong systems-level skills!

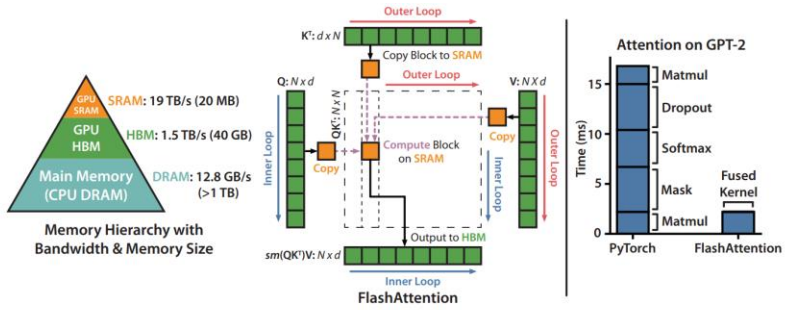


Figure 1: **Left:** FLASHATTENTION uses tiling to prevent materialization of the large $N \times N$ attention matrix (dotted box) on (relatively) slow GPU HBM. In the outer loop (red arrows), FLASHATTENTION loops through blocks of the **K** and **V** matrices and loads them to fast on-chip SRAM. In each block, FLASHATTENTION loops over blocks of **Q** matrix (blue arrows), loading them to SRAM, and writing the output of the attention computation back to HBM. **Right:** Speedup over the PyTorch implementation of attention on GPT-2. FLASHATTENTION does not read and write the large $N \times N$ attention matrix to HBM, resulting in an 7.6× speedup on the attention computation.

"FlashAttention trains Transformers faster than existing baselines: **15% end-to-end wall-clock speedup on BERT-large (seq. length 512)...3× speedup on GPT-2 (seq. length 1K), and 2.4× speedup on long-range arena (seq. length 1K-4K)**"

This is an example of "**operator fusion**": rather than calculate $softmax(\frac{QK^T}{\sqrt{d_k}})V$ as separate matmult/softmax operations, fuse them into a single operator (optimized for GPU hardware).

(optional) here's a neat explanation of FlashAttention if you're curious: "ELI5: FlashAttention" (2023)

Transformers for computer vision

Context: it's 2021. CNN's are still the dominant computer vision model arch.

"Attention is all you need" (Vaswani et al [link]) came out in NIPS 2017. Its "transformers" model architecture is causing a revolution in NLP

Natural question: can we apply transformers to the computer vision domain? Say, image classification?

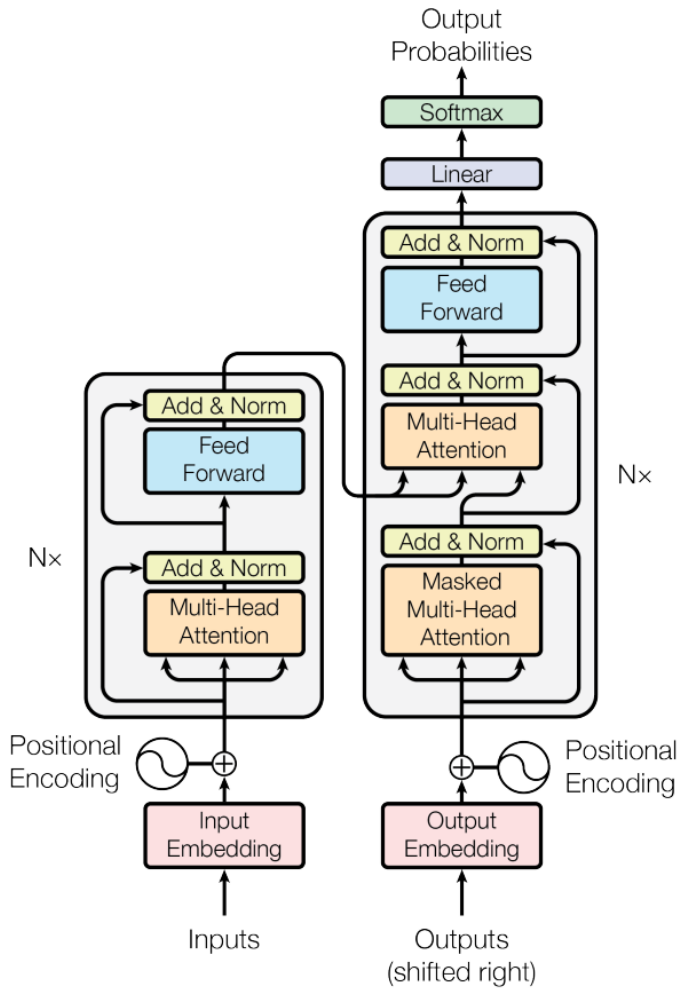


Figure 1: The Transformer - model architecture.

Visual Transformer (2021)

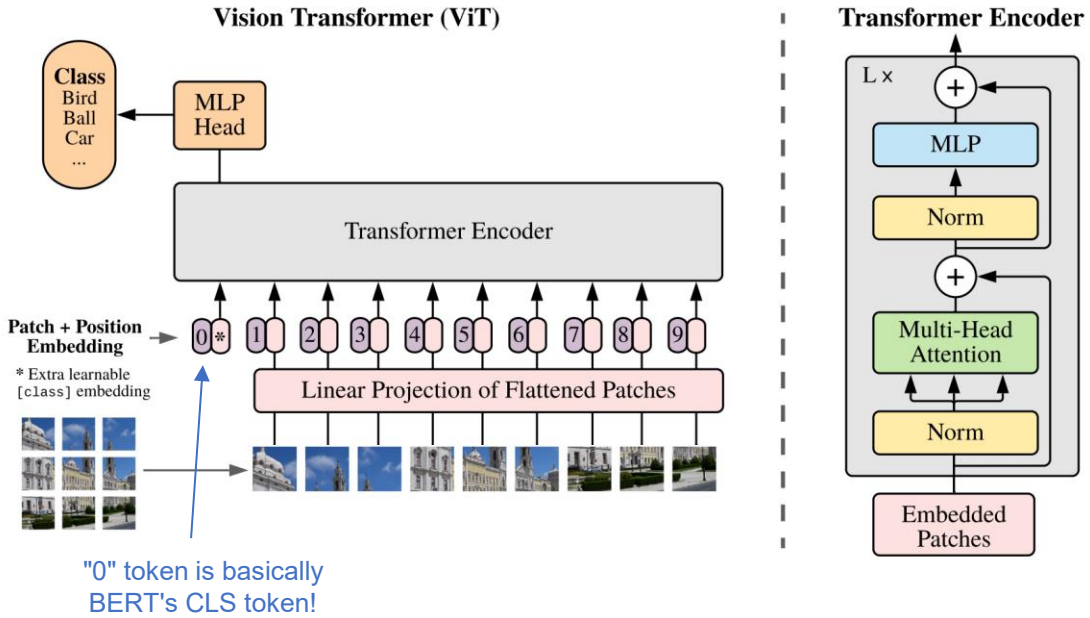
Enter: the Visual Transformer (ViT)

- "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale" (Dosovitskiy, Beyer, Kolesnikov, Weissenborn, Zhai et al) [link]. ICLR 2021

Idea: represent an image as a sequence of [16x16] patches, left-to-right, top-to-bottom ("raster" order),

- Pass this image sequence to a **transformer encoder**, and train an image classifier on top of it!

Results: achieved state-of-the-art results on ImageNet-1k vs CNN-based methods like ResNets



Earlier, we've discussed (in detail!) the mechanical details of how transformer encoders work. Now, let's dig into the training methodology!

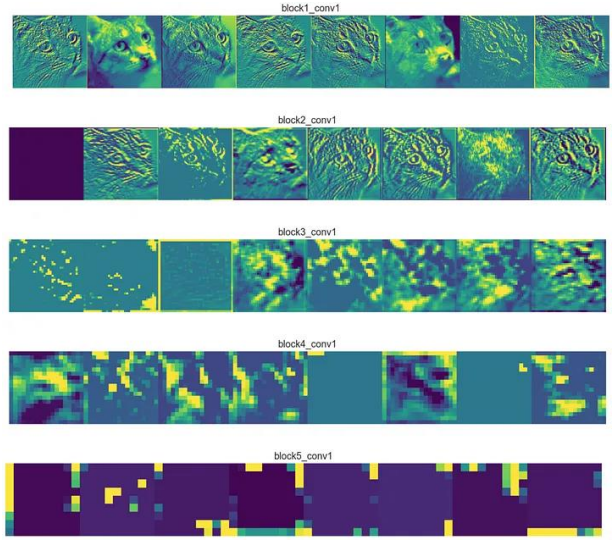
Inductive bias

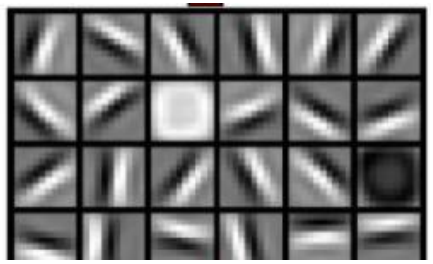
Definition: "inductive bias" of a model is an **architecture-level inclination** towards certain kinds of phenomenon/behavior.

Example: CNN's have a strong inductive bias towards local, translation-invariant features, due to Conv2d being translation-invariant.

But: sometimes, we want our features to be a mix of local+global

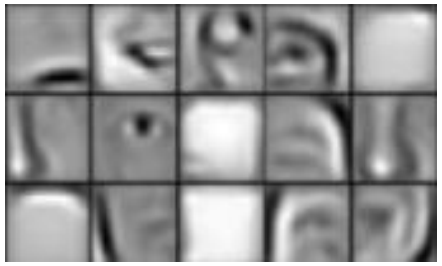
Also: the high-level semantic CNN features tend to also have poor spatial resolution (due to the feature-map downsampling after each Conv block)





Filters @ Layer 1:

edge detectors?



Filters @ Layer 2:

ears? noses?

https://medium.com/@saba99/feature-map-35ba7e6c689e

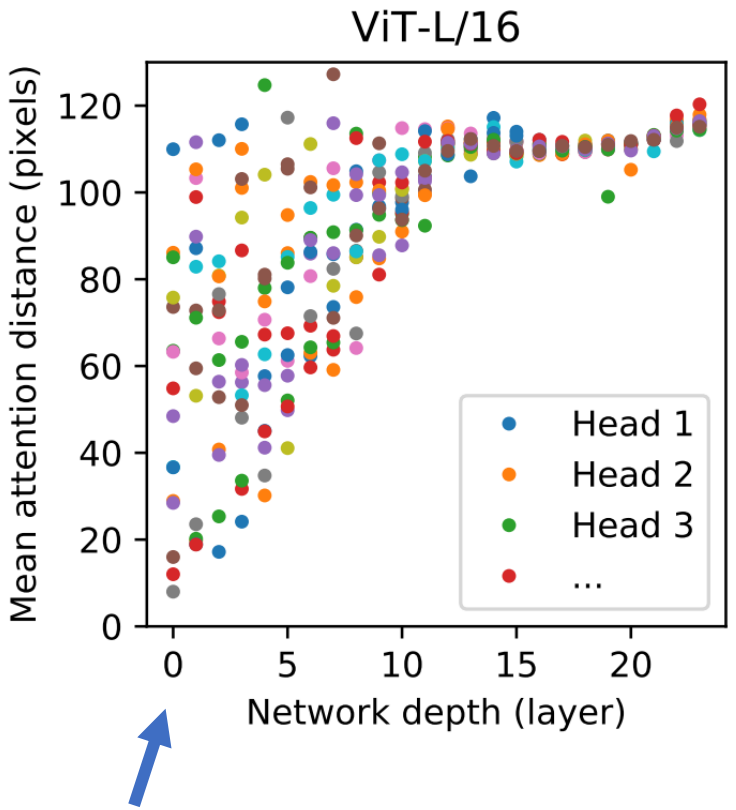
ViT: less inductive bias

A selling point for ViT is that it has less inductive bias than CNN's

Ex: at each transformer encoder block, each token (aka image patch) can interact with (aka "attend to") every other image patch in the image

Implication: this means that ViT can, at every transformer layer, learn features that involve information from **any part of the image.**

In contrast: CNN's can only learn features based on spatially local information ("receptive field")



Interesting observation: in early transformer layers, some heads use global information, and some use mostly local information.
CNN's can only use local info!

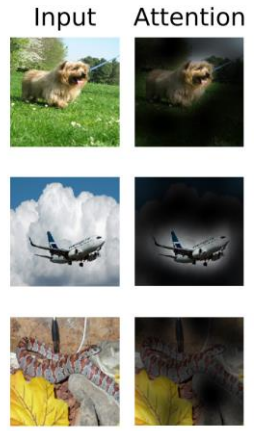


Figure 6: Representative examples of attention from the output token to the input space. See Appendix D.7 for details.

ViT: `image_patchify()`

Goal: represent an image [chans, height_img, width_img] as a sequence with shape [seq_len, d].

Idea: grid up image into patches!

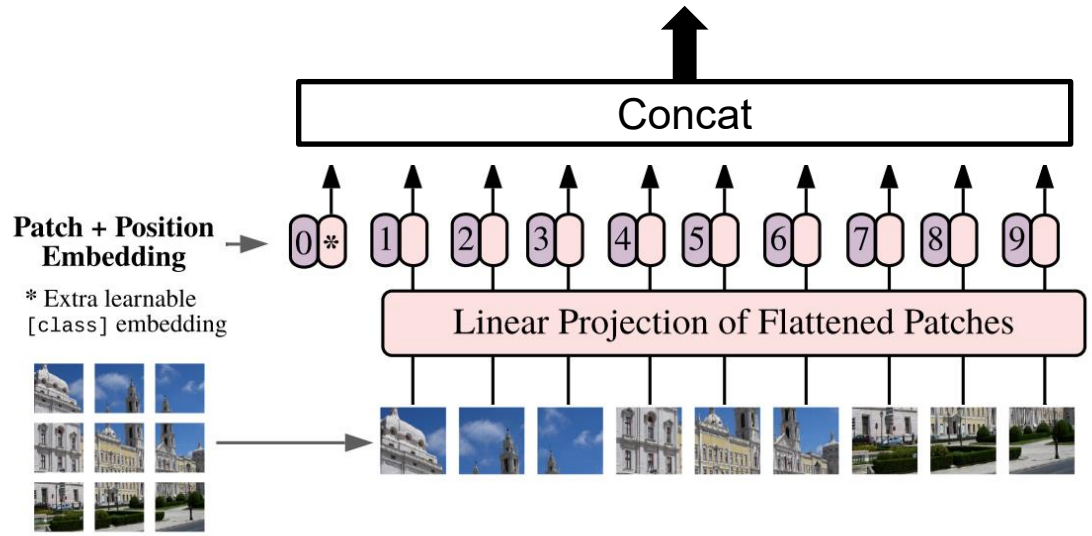
Tip: patch_size is typically 16x16. Generally, passing in larger image resolutions (ie longer seq lens) leads to better task performance, but is more expensive/slower to train/inference on (tradeoff!)

seq_len is total number of patches.

For a square image: seq_len = $ceil\left(\frac{height\_img}{patch\_size}\right)^2$

Ex: for a 224x224 image and patch_size=16: seq_len = 196.

(3) Concat all patch embeddings into a single [seq_len, dim] tensor.



(1) grid up the image into [patch_size x patch_size] patches

(2) Extract an embedding for each patch (with dimensionality `dim`)

Tangent: einops

ViT implementations often use a library called "**einops**" for image_patchify()

einops: "Einstein-Inspired Notation for operations"

(for you physics fans) notation is loosely inspired by Einstein summation [link] (ex: einops.einsum)

Purpose: make it easier (and safer/more-explicit) to do certain operations with multidimensional tensors (like reshaping)

Useful tutorials: [link1] [link2] [link_github]

Why use `einops` notation?!

Semantic information (being verbose in expectations)

```
y = x.view(x.shape[0], -1)
y = rearrange(x, 'b c h w -> b (c h w)')
```

While these two lines are doing the same job in *some* context, the second one provides information about the input and output. In other words, `einops` focuses on interface: *what is the input and output*, not *how* the output is computed.

The next operation looks similar:

```
y = rearrange(x, 'time c h w -> time (c h w)')
```

but it gives the reader a hint: this is not an independent batch of images we are processing, but rather a sequence (video).

Semantic information makes the code easier to read and maintain.

ViT: Training methodology

The ViT model architecture is fairly straightforward:

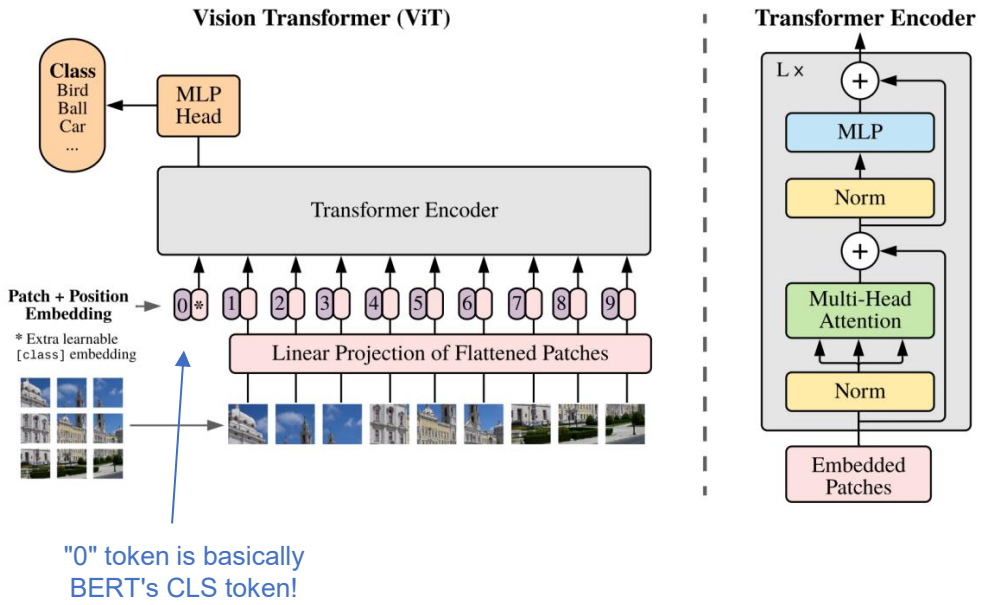
- image_patchify() + standard transformer encoder classifier

Turns out: this is only part of the story!

The remaining "secret sauce": the training methodology + dataset.

A recent trend in academia and industry: improvements in dataset quality/scale often trumps architecture tweaks

- Corrolary: massively scaling up both dataset size and model capacity = wins!



ImageNet-1K

ImageNet-1K [link] is the de-facto academic image classification dataset

1000 categories, 1,281,167 training images, 50,000 validation images and 100,000 test images

During its time, it was the largest-scale image classification dataset

History:

- 2006: Fei-Fei Li [link] began working on the idea for ImageNet
- 2009: ImageNet poster presentation (CVPR 2009 [link])
- 2010: First ImageNet Large Scale Visual Recognition Challenge

ImageNet-1k, ImageNet-21k, JFT-300M

ImageNet-1k (2010): main ImageNet release

- Human annotated
- **1.2M training images, 1000 categories**

ImageNet-21k (2010): superset of ImageNet-1k

- **14M training images, 21,841 categories**

JFT-300M (2017): image classification dataset from Google [link]

- **300M training images, 18,000 categories**
- Collected semi-automatically
- Downside: proprietary closed-source dataset private just to Google :(

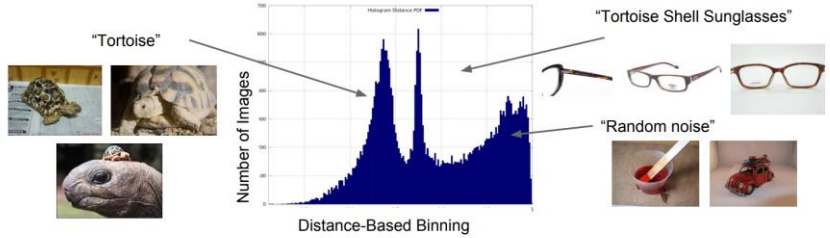


Figure 2. JFT-300M dataset can be noisy in terms of label confusion and incorrect labels. This is because labels are generated via a complex mixture of web signals, and not annotated or cleaned by humans. x-axis corresponds to the quantized distances to K-Means centroids, which are computed based on visual features.

Dataset trends

Observation: lots of work in scaling up image classification model architectures (eg CNNs like ResNet), but not a lot of work in scaling up datasets

- "Let's just use ImageNet-1k since everyone else uses it"

Idea: does anything change if we dramatically increase our training dataset size?

- Followup: what if we both increase the dataset size AND the model size? (...foreshadowing for ViT...)

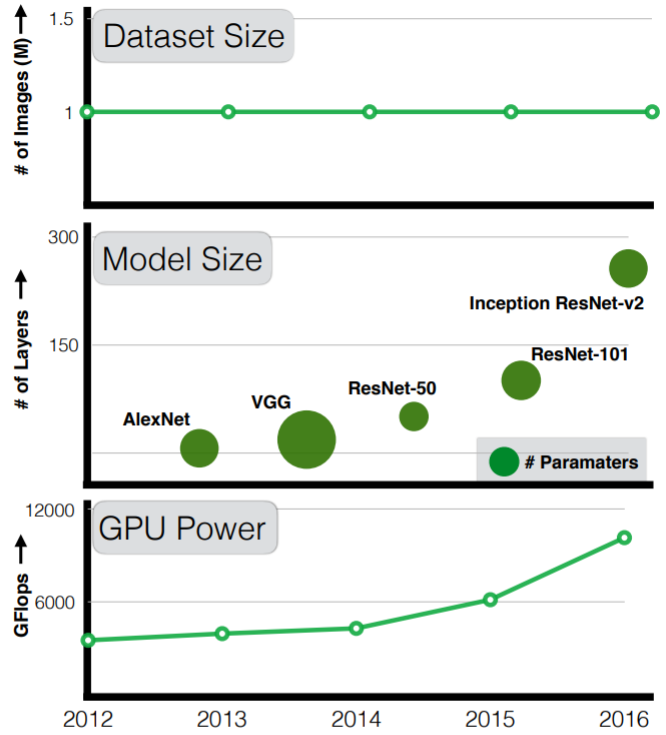


Figure 1. The Curious Case of Vision Datasets: While GPU computation power and model sizes have continued to increase over the last five years, size of the largest training dataset has surprisingly remained constant. Why is that? What would have happened if we have used our resources to increase dataset size as well? This paper provides a sneak-peek into what could be if the dataset sizes are increased dramatically.

https://arxiv.org/abs/1707.02968

ViT: dataset ablations

(back to ViT)

When training+testing on ImageNet-1k, **CNNs are better than ViT!**

- What?! I thought transformers were The Best Thing?

But: when training on larger datasets (ImageNet-21k, JFT-300M) and testing on ImageNet-1k: **ViT outperforms CNNs**

Takeaway: transformers (like ViT) are most effective when trained on LOTS of data.

Said another way: transformer models have more **model capacity** than CNN's: we can feed it more training data without performance plateauing.

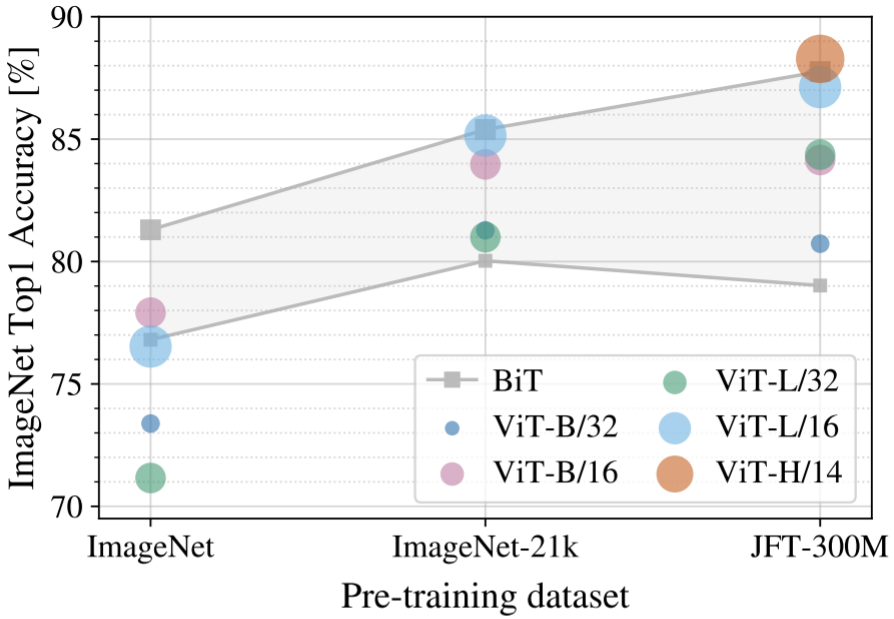


Figure 3: Transfer to ImageNet. While large ViT models perform worse than BiT ResNets (shaded area) when pre-trained on small datasets, they shine when pre-trained on larger datasets. Similarly, larger ViT variants overtake smaller ones as the dataset grows.

Tangent: Learning rate schedules

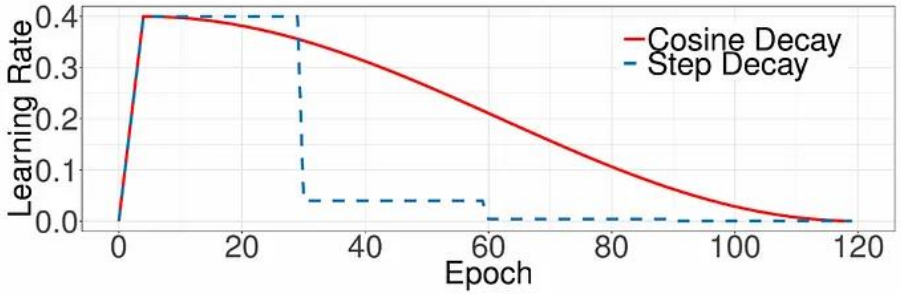
So far in this class, we've used a single learning rate. In practice, it's better to use learning rate schedules

Start learning rate small, then gradually ramp it up to a larger value (ie the first ~100 iterations)

- Intuition: starting learning rate too high often leads to training divergence (eg NaN losses). Thus, we start it low to get the model weights in a "healthy" region, then slowly increase the learning rate

Over the course of training, decay the learning rate

- Intuition: during early parts of training, model needs to make big steps (high LR). But, near the end of training, model is focusing on finer-grained details (small LR).



torch.optim.lr_scheduler.StepLR
torch.optim.lr_scheduler.CosineAnnealingLR

https://pub.towardsai.net/are-you-sure-that-you-can-implement-image-classification-networks-d5f0bffb242d