Lecture 17: Visual Transformer, Masked auto encoder

Data C182 (Fall 2024). Week 11. Tuesday Nov 5th, 2024

Speaker: Eric Kim

Rev01: 2024-11-05 7:58 PM PST

Announcements

- HW03 ("Transformers + NLP") out! Due: Fri Nov 22nd 11:59 PM PST
  - Please start early!
- Midterm regrade requests due: Fri Nov 8th, 11:59 PM PST
  - Midterm stats: [link]
  - "Course Grade Guidance": [link]

Today's lecture

- Visual Transformer
- Masked auto encoder

Context: Transformers

- 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?

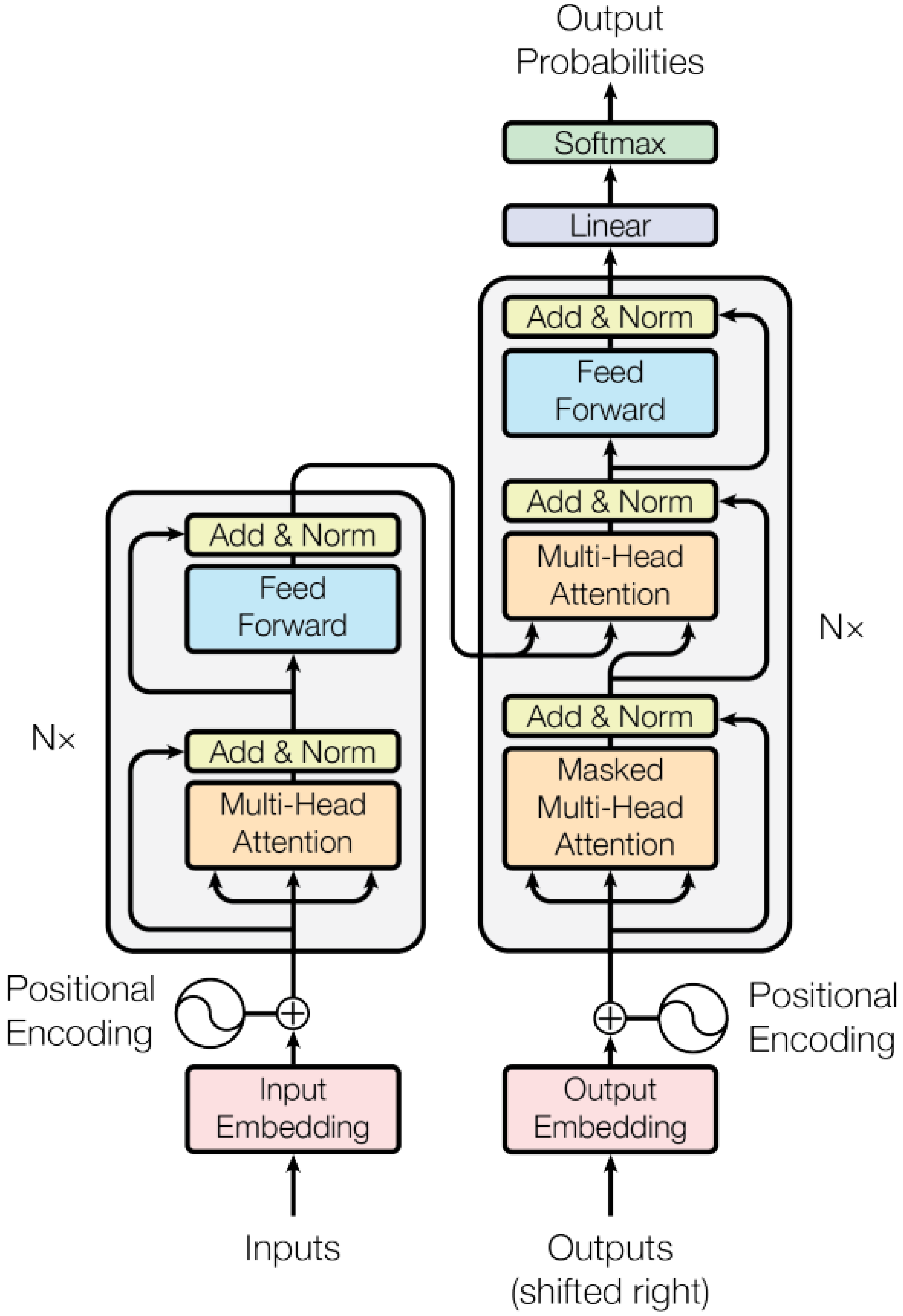
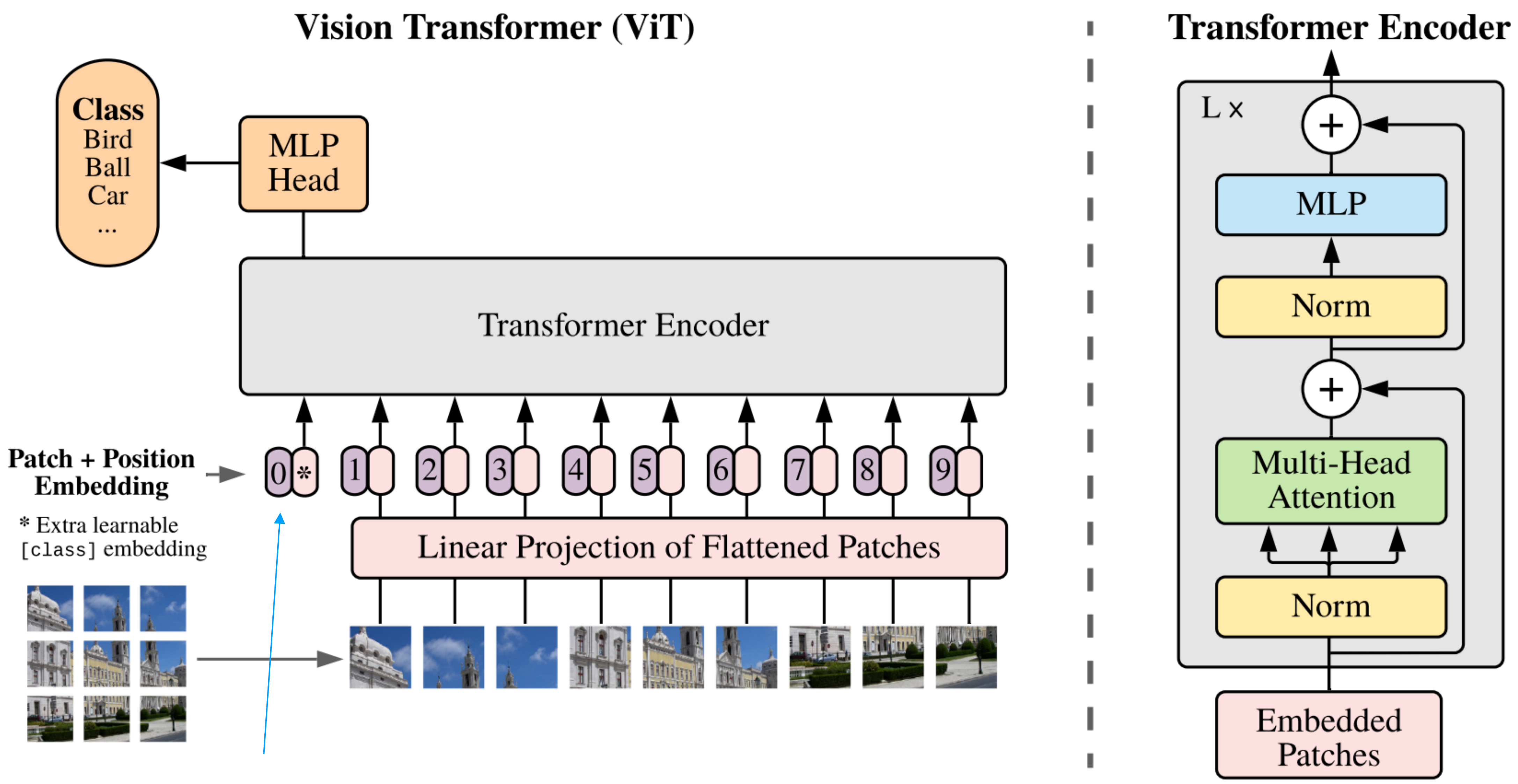


Figure 1: The Transformer - model architecture.

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

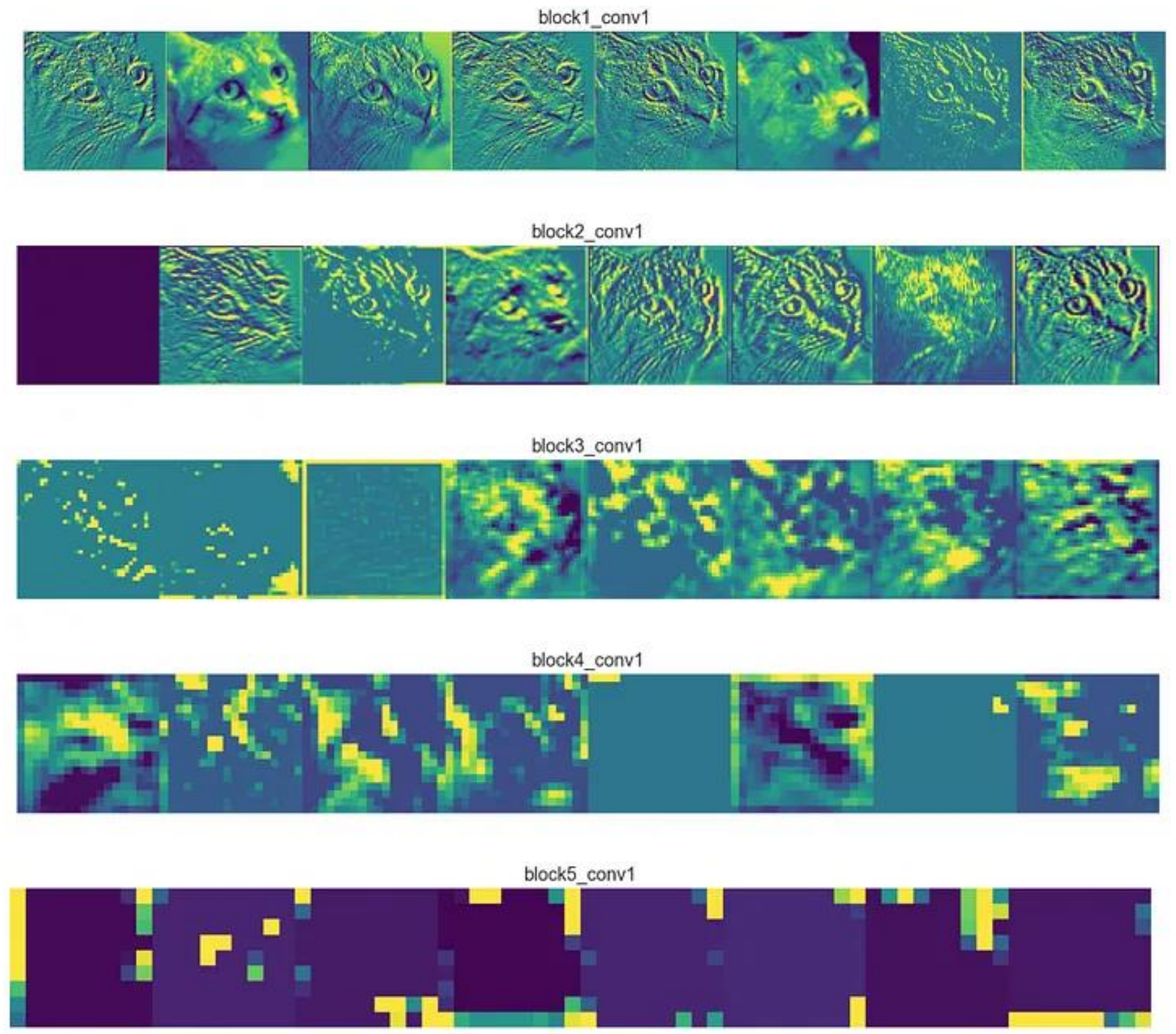


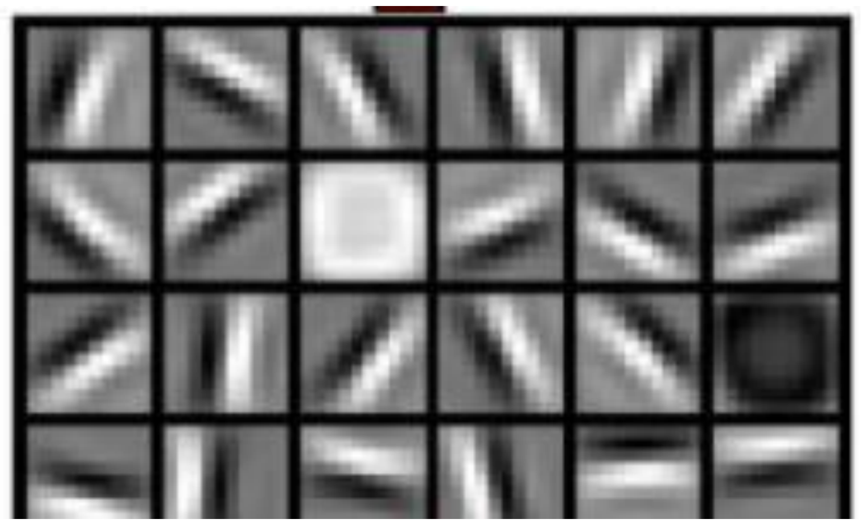
"0" token is basically BERT's CLS token!

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

Inductive bias

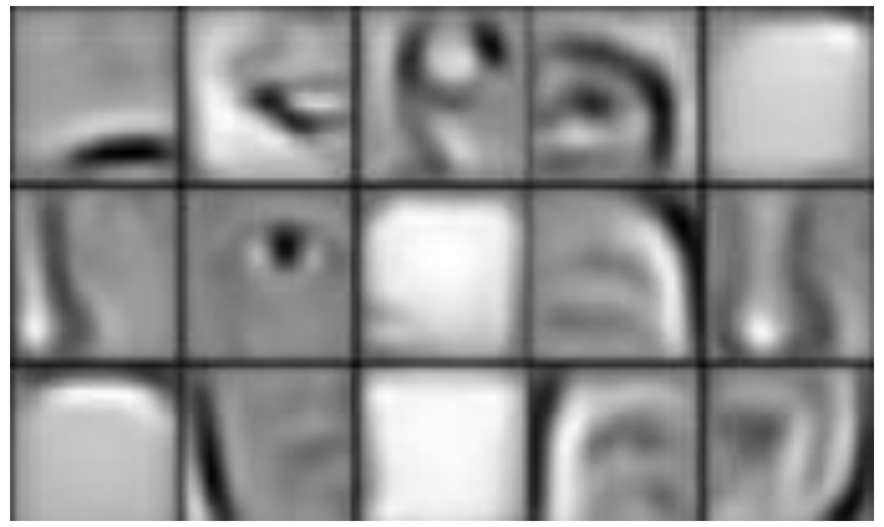
- Definition: "inductive bias" of a model is an **architecture-level inclinations** towards certain kinds of phenomenon/behavior.
- Example: CNN's have a strong inductive bias towards local, translation-invariant features, due to Conv2d being translation-invariant.
  - But: sometimes, we want our features 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?



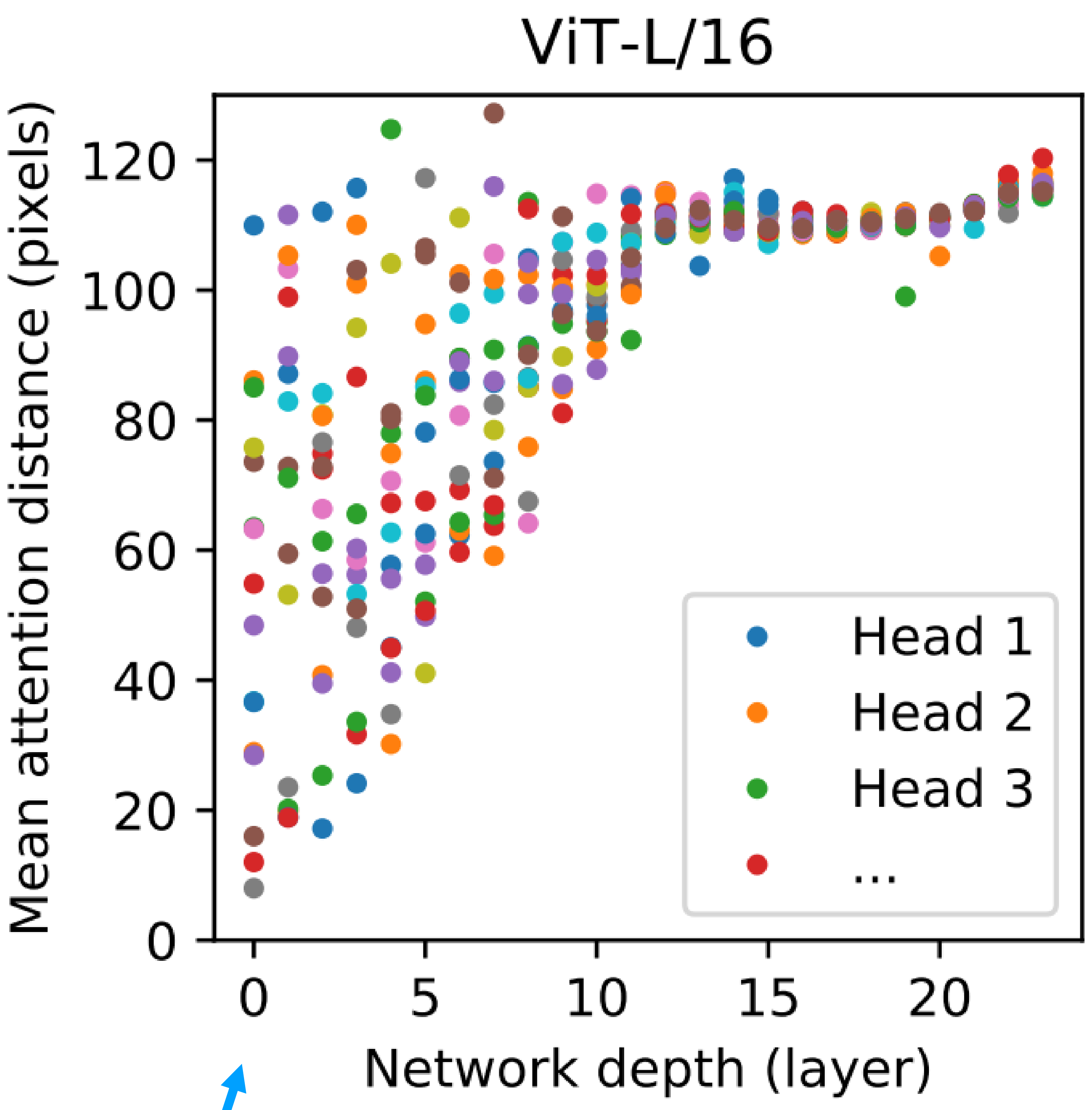
Filters @ Layer 2:

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 off of spatially local information ("receptive field")



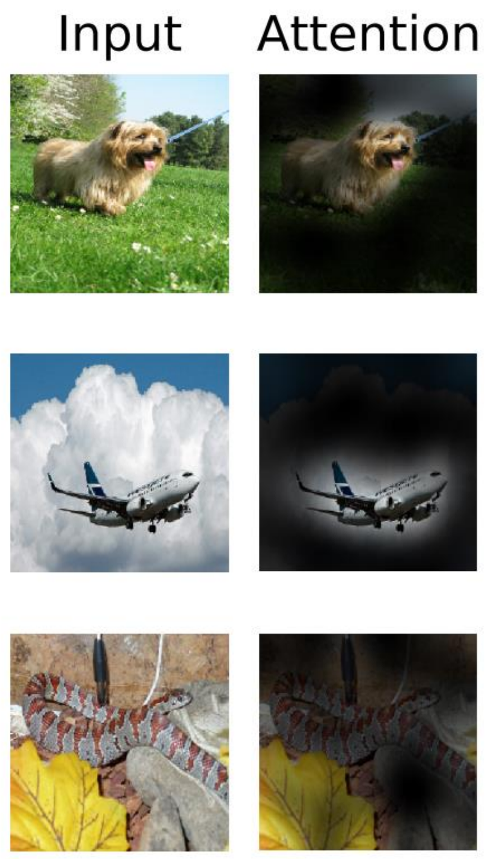


Figure 6: Representative examples of attention from the output token to the input space. See Appendix D.7 for details.

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

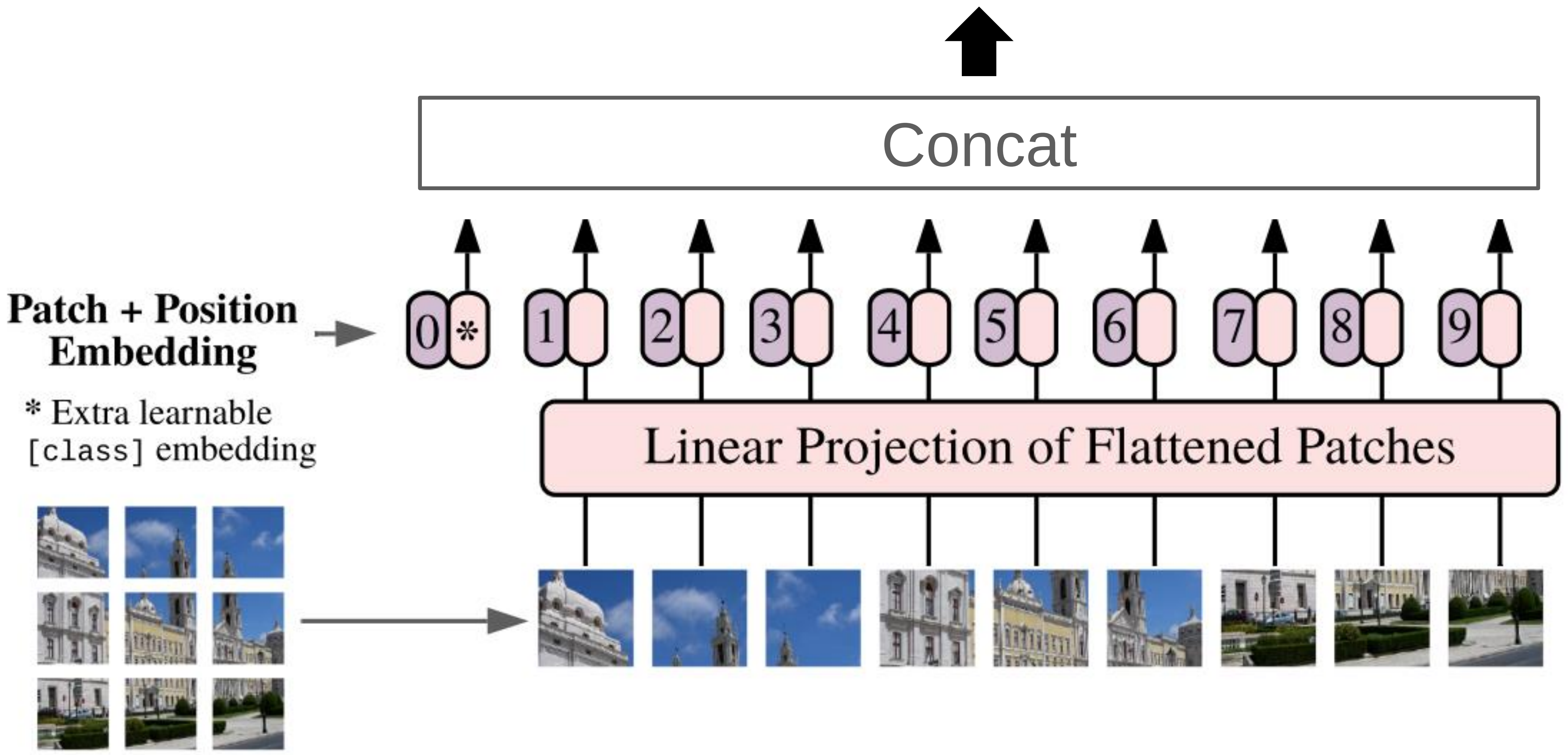
ViT Implementation details: "image_patchify()"

- Goal: represent an image [chans, height_img, width_img] as a sequence with shape [seq_len, dim].
- Idea: grid up image into patches!
- Tip: patch_size is typically 16x16. Generally, passing in larger image resolutions (eg 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. Ex: for a square image: seq_len = ceil(height_img / patch_size) ** 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.

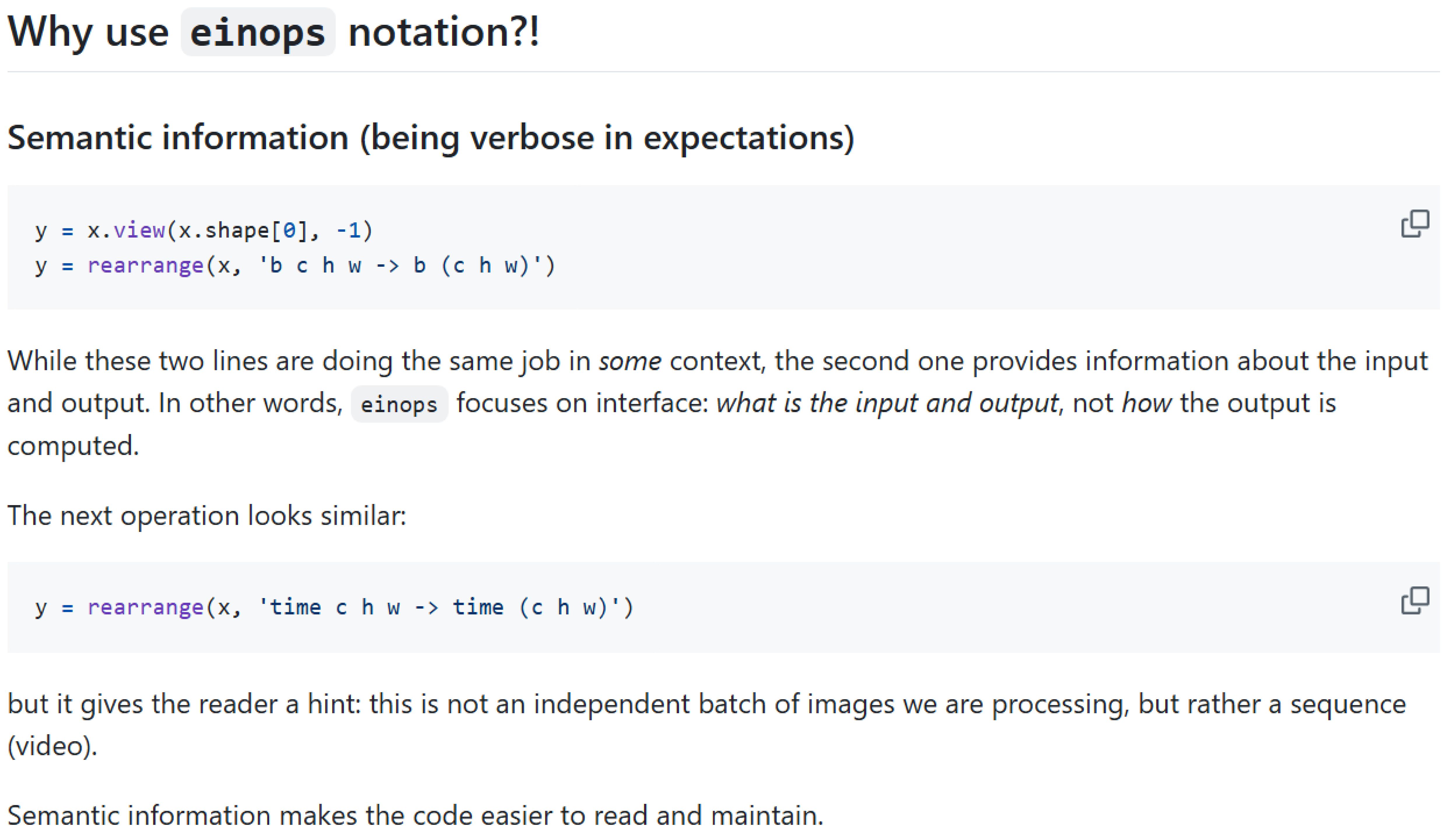

Concat
Patch + Position Embedding
* Extra learnable [class] embedding
0
*
1
2
3
4
5
6
7
8
9
Linear Projection of Flattened Patches

(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)')
```

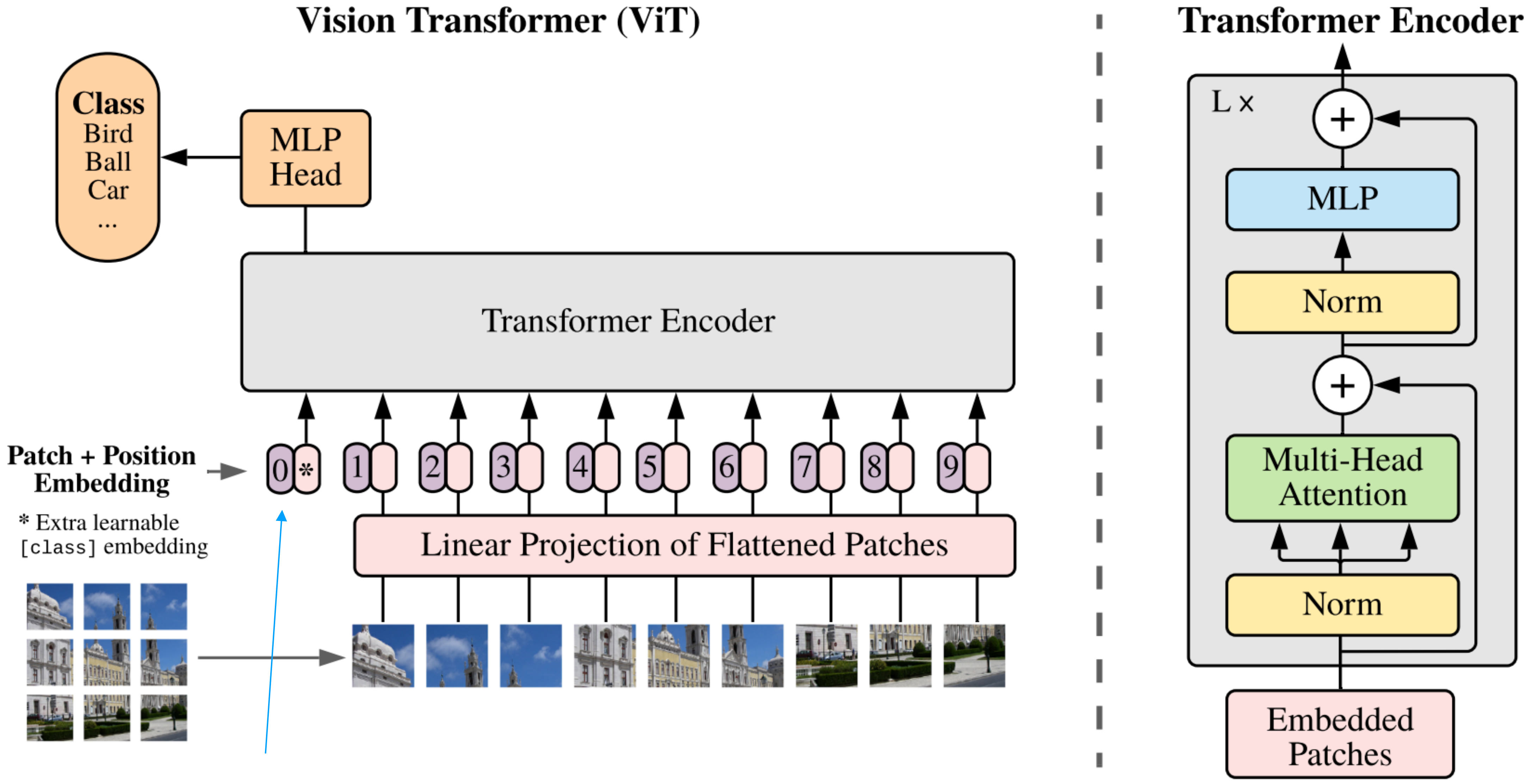
but it gives the reader a hint: this is not an independent batch of images we are processing, but rather a sequence (video).

Semantic information makes the code easier to read and maintain.

(Will be useful for HW04!)

ViT: Training methodology

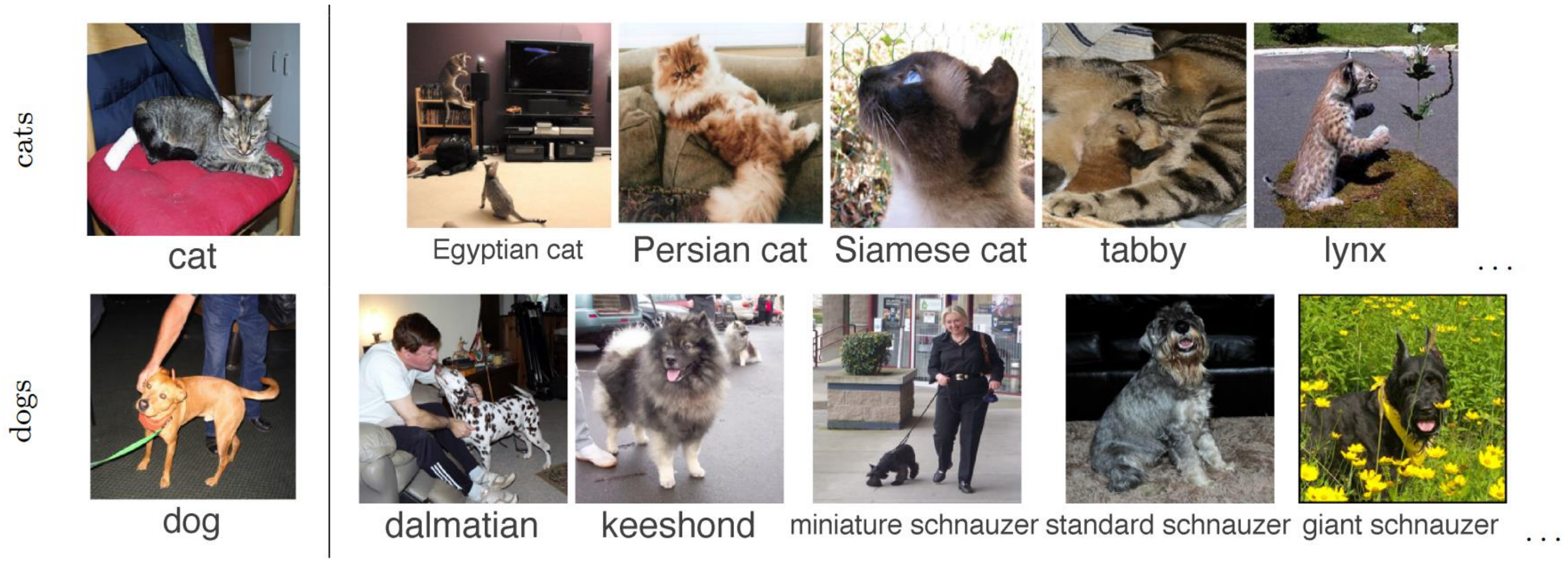
- At this point: you know how to mechanically implement the ViT model
  - Image_patchify() + transformer encoder classifier
- It turns out: this is only part of the story!
- The remaining "secret sauce": the training methodology + dataset.
- A recent trend in academia and industry: improvements in dataset quality/scale often trumps architecture tweaks
  - Followup: massively scaling up both dataset size and model capacity = wins!



"0" token is basically BERT's CLS token!

Context: 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 prime 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 :(

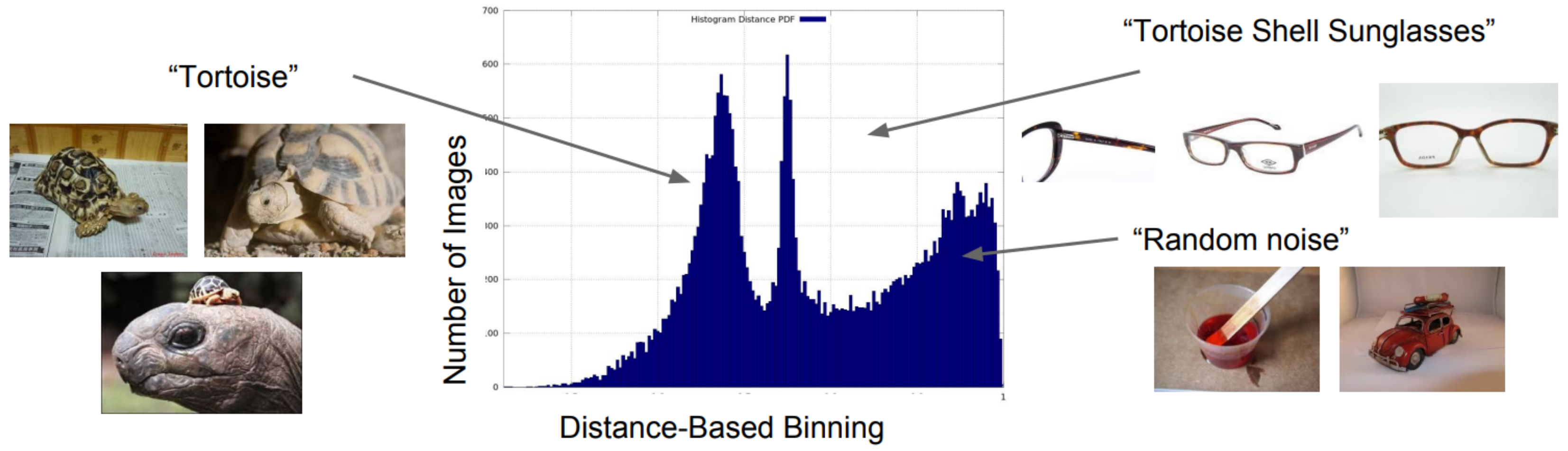


Figure 2. JFT-300M dataset can be noisy in terms of label confusion and incorrect labels. This is because labels are generated via a complex mixture of web signals, and not annotated or cleaned by humans. x-axis corresponds to the quantized distances to K-Means centroids, which are computed based on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

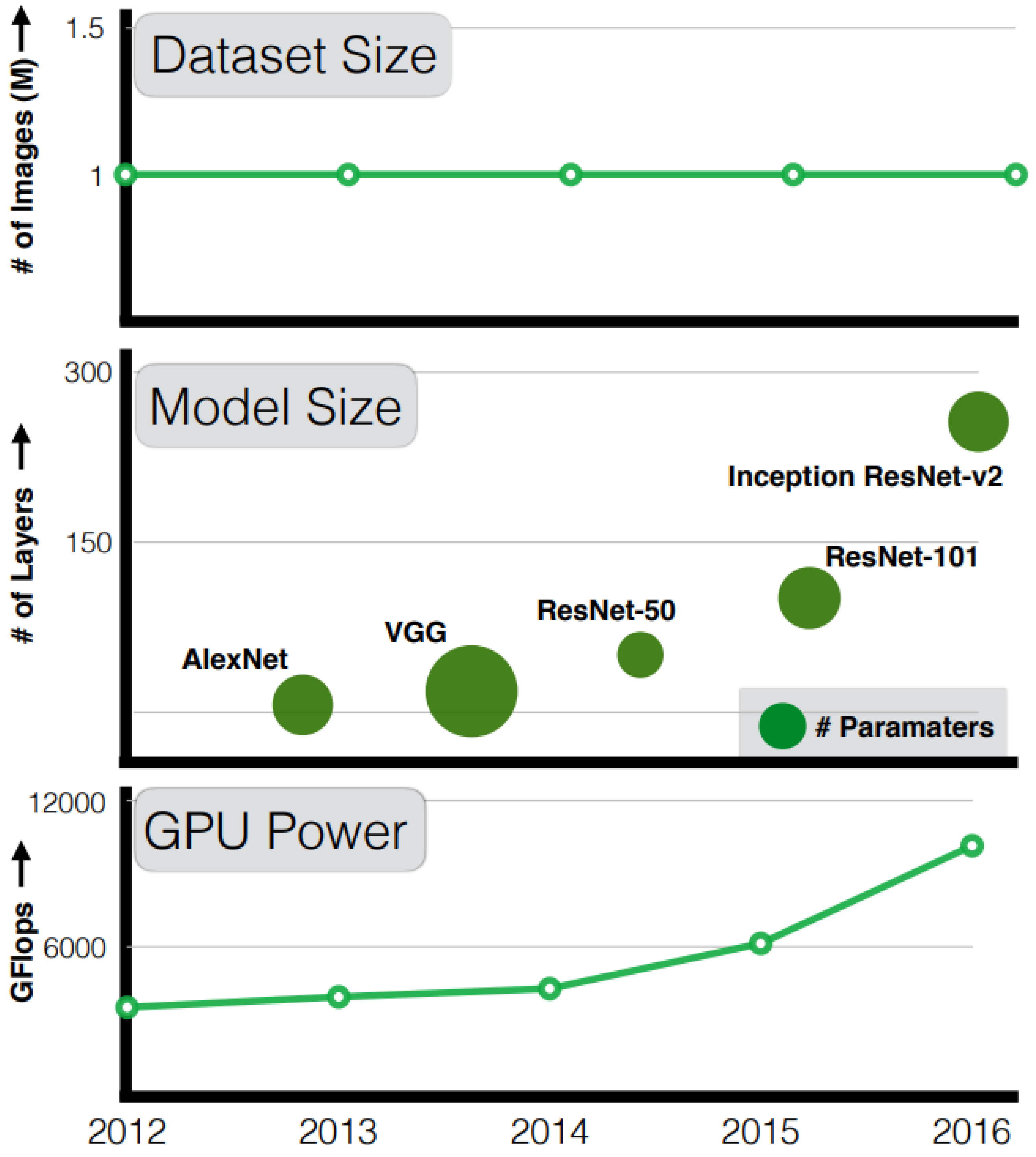


Figure 1. The Curious Case of Vision Datasets: While GPU computation power and model sizes have continued to increase over the last five years, size of the largest training dataset has surprisingly remained constant. Why is that? What would have happened if we have used our resources to increase dataset 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 ImageNet-21k (and, JFT-300M) and testing on ImageNet-1k: **ViT outperforms CNNs**
- **Takeaway**: transformers (like ViT) are most effective when trained on LOTS of data

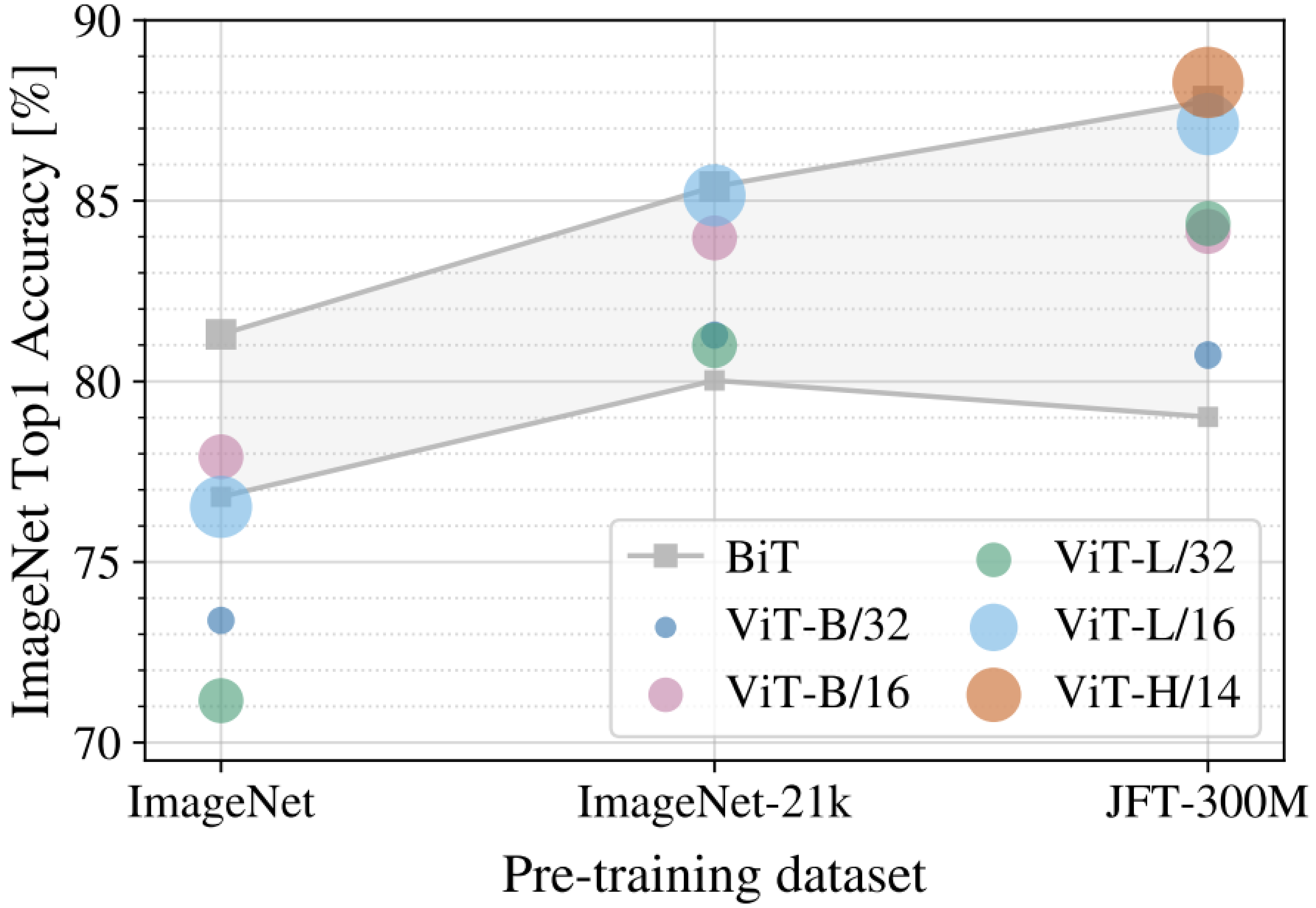
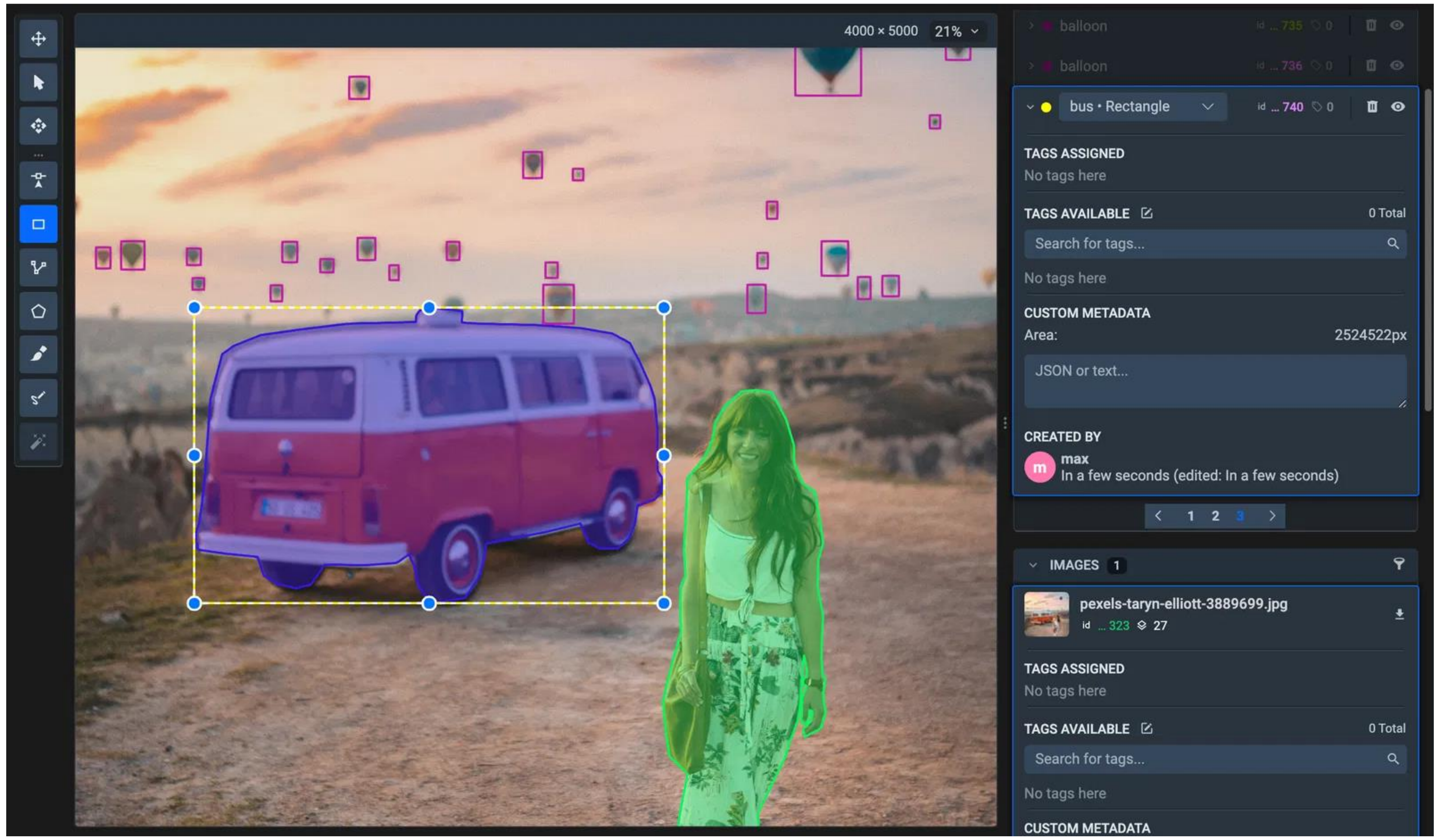


Figure 3: Transfer to ImageNet. While large ViT models perform worse than BiT ResNets (shaded area) when pre-trained on small datasets, they shine when pre-trained on larger datasets. Similarly, larger ViT variants overtake smaller ones as the dataset grows.

Scaling up datasets: challenges

- The primary challenge with scaling up datasets like ImageNet is: getting high-quality human labels at scale is very expensive, both in terms of $ and time+effort.
- However, we've seen that transformer model architectures (like ViT) are data hungry, and require lots of training data to realize its potential
- If collecting human annotations is too expensive, what are our alternatives?
- Idea: can we create an image dataset **without any human annotations?**



TAGS ASSIGNED
No tags here

TAGS AVAILABLE 0 Total
Search for tags...
No tags here

CUSTOM METADATA
Area: 2524522px
JSON or text...

CREATED BY
max
In a few seconds (edited: In a few seconds)

pexels-taryn-elliott-3889699.jpg
id ... 323 27

TAGS ASSIGNED
No tags here

TAGS AVAILABLE 0 Total
Search for tags...
No tags here

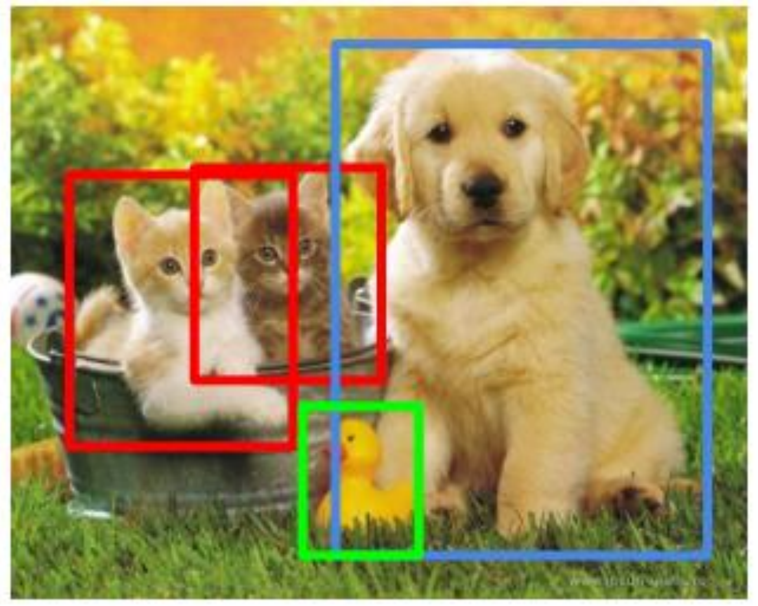
CUSTOM METADATA

https://supervisely.com/labeling-toolbox/images/

Tangent: learning paradigms

- So far in this class, we've focused on supervised training: Given a dataset, each row has an input and a ground-truth label.
  - Ex: image classification, object detection, machine translation
- However, there are other training paradigms!

Classification: Image + label



Detection: Image + boxes

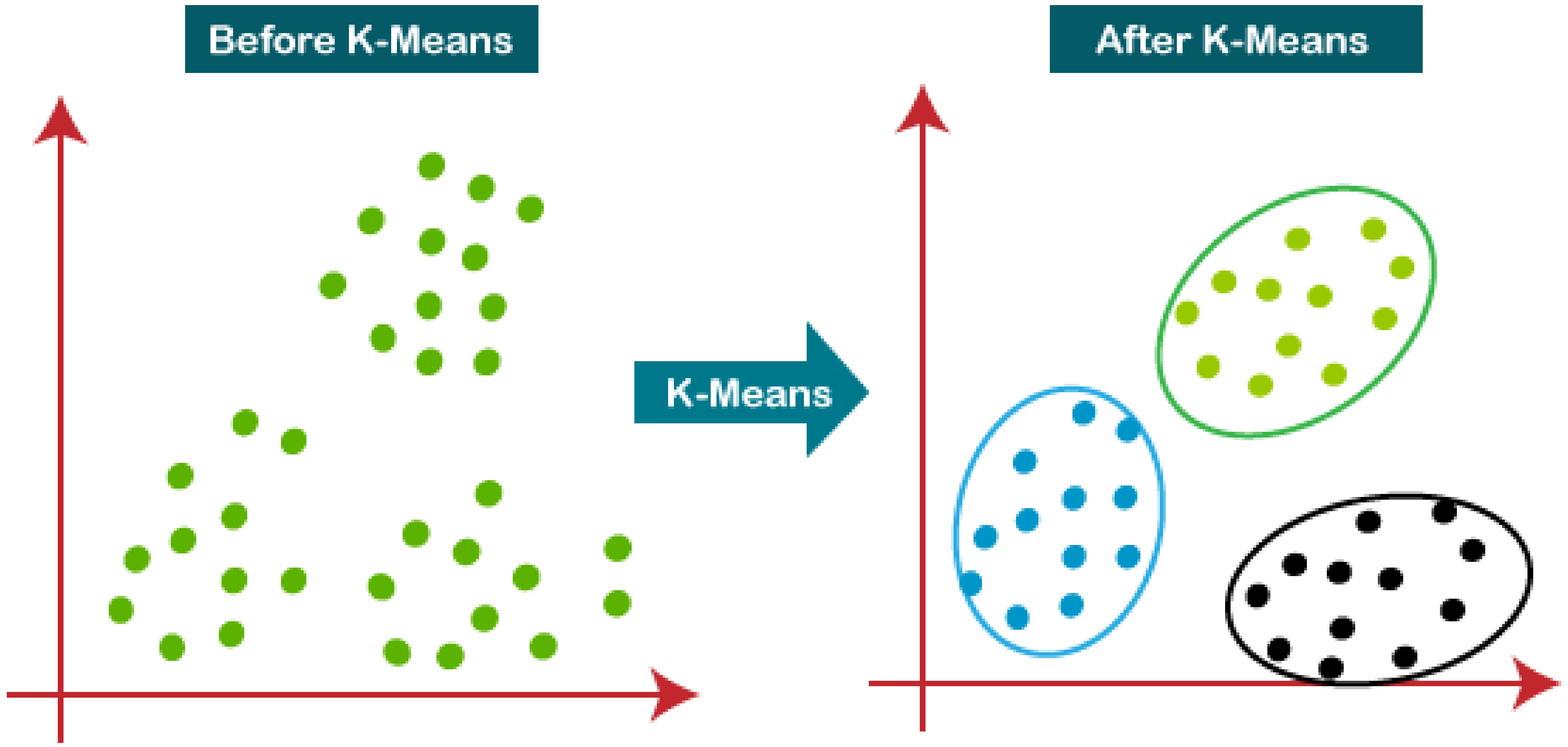
Segmentation: Image + masks

I am sleepy (EN) -> j'ai sommeil (FR)

Translation: Text for language A and language B

Unsupervised learning

- Unsupervised: dataset has NO labels
  - Ex: clustering algorithms (k-means)



https://keytodatascience.com/k-means-clustering-algorithm/

Self-supervised learning

- Self-supervised: create our own labels based on the input (no human labeling required!)
- NLP "fill in the blank": given a sentence, randomly blank out some words. Ask text model to predict the removed words. ("cloze" task [link])
    - Ex: "Today, I went to the ______ and bought some milk and eggs."
        - Target: "store"
    - Pro: it's really easy to scrape tons of text data from the internet, and very easy to construct cloze examples. ("unlimited" training data for free!)

Why self-supervised?

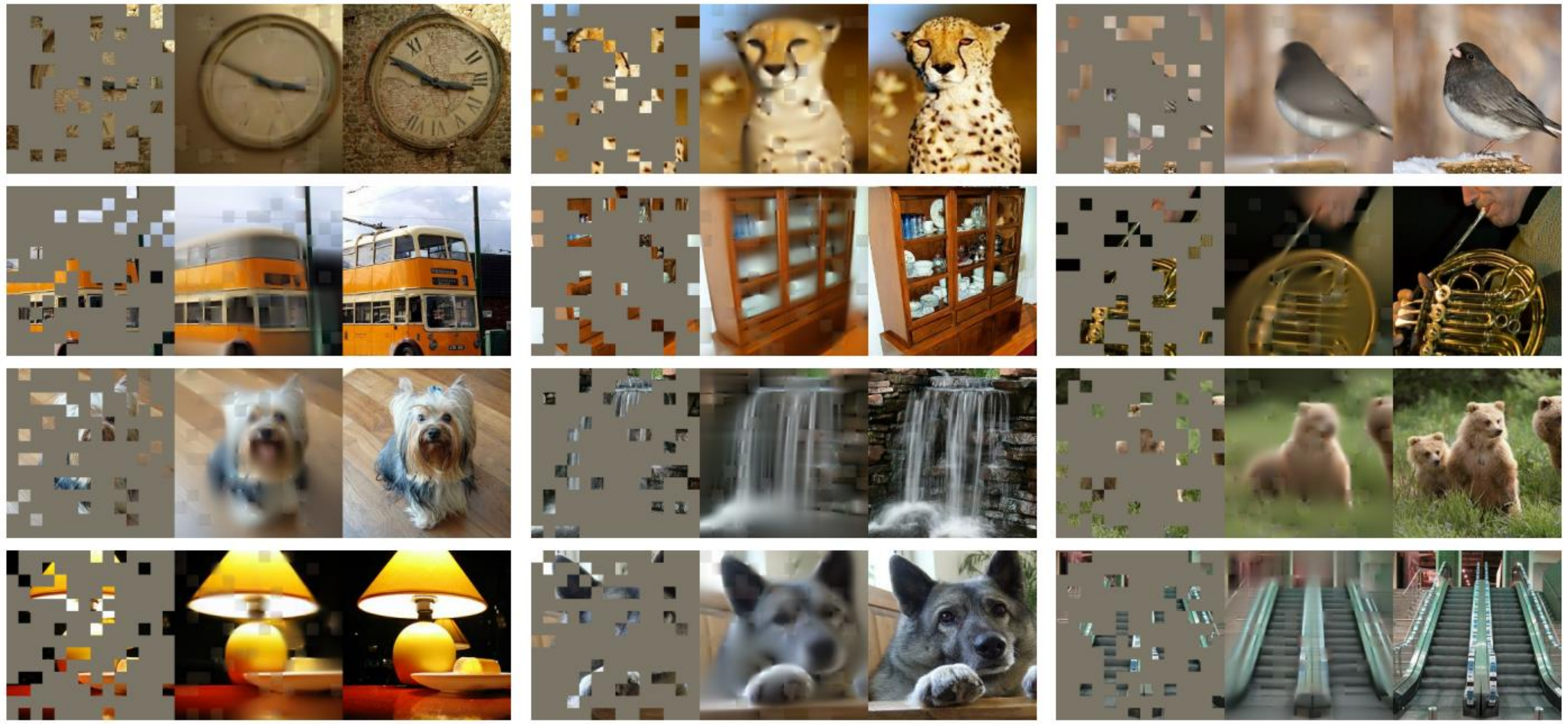
- **Motivation ("the dream")**: training a model on large self-supervised datasets leads to a stronger "starting point" for downstream applications (eg classification)
- **Intuition**: pretraining a model on self-supervised tasks lets the model learn "something" about the visual/text world. Then, fine-tuning starts off from a "strong" starting point (doesn't have to start from scratch!)

```
# Create model (randomly init'd)
model = create_model()
# Download pretrained model weights
# Ex: pretrained on self-supervised task
model_pretrained_weights = download_pretrained_weights()
# Load pretrained weights into our model
model.load(model_pretrained_weights)
# Fine-tuning: train starting from pretrained weights
train_model(model, dataset)
```

Pytorch documentation on model.save(), model.load(): [link]

Self-supervised: computer vision

- Question: how do we apply the "Fill in the blank" task from NLP to computer vision?
- Answer: "Fill in the pixels!"
- How do we design a DNN to predict pixel values (rather than classification labels)?



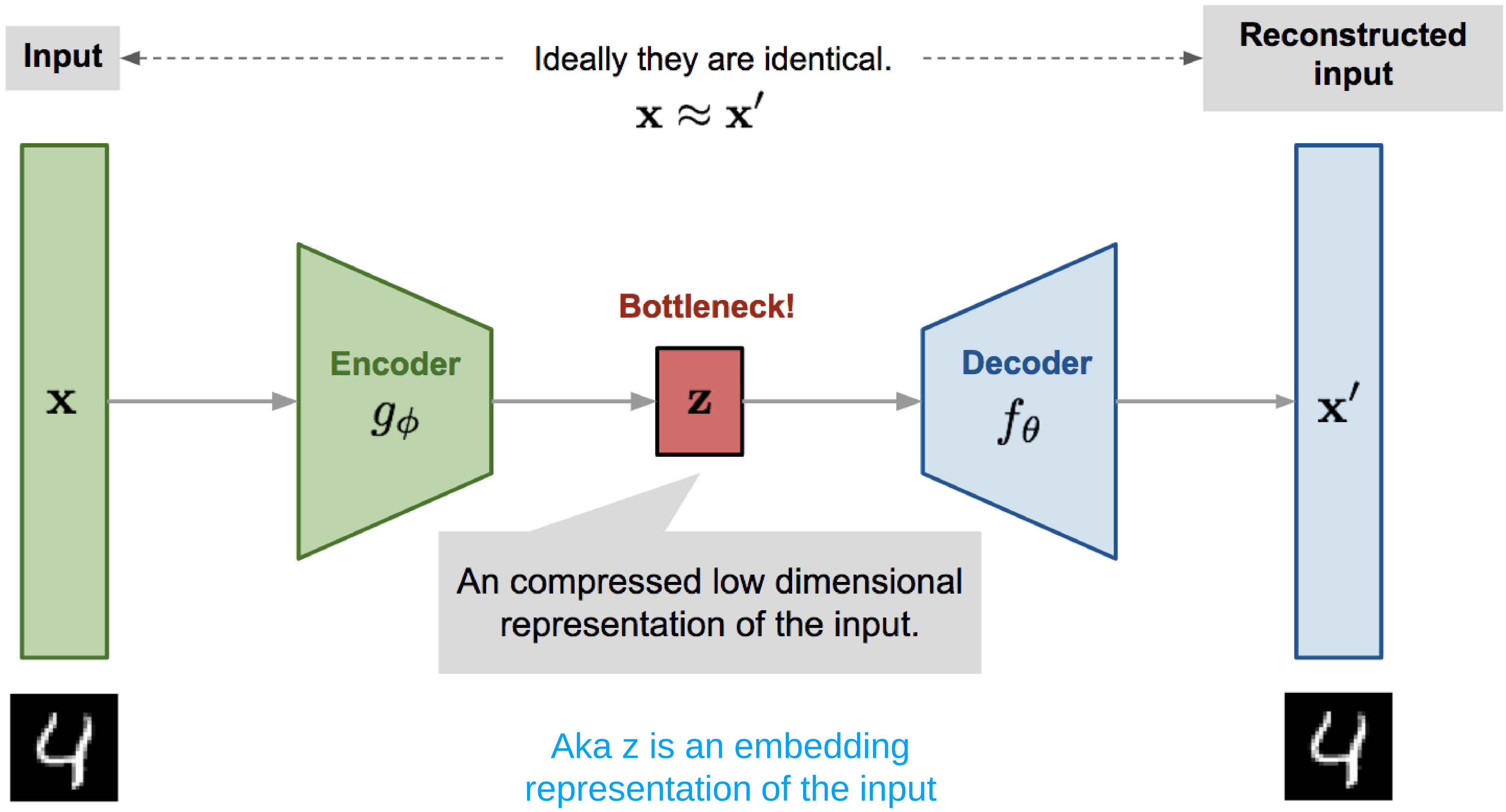
For each group: (**left**) image with blanked-out pixels (**middle**) model predictions (**right**) ground truth image

Ex: "Today, I went to the _______ and bought some milk and eggs."
Target: "store"

https://arxiv.org/abs/2111.06377

Background: Autoencoders

- Autoencoders are a classic, well-studied technique
- Idea: transform an input (eg image, text) into a latent representation (aka embedding), and then reconstruct the input from the latents
  - **Encoder**: Given image, transform into an embedding(s) (aka latent)
  - **Decoder**: Given latent representation, reconstruct original image


Input
Ideally they are identical.
$\mathbf{x} \approx \mathbf{x}'$
Reconstructed input
$\mathbf{x}$
Encoder
g_ϕ
Bottleneck!
$\mathbf{z}$
Decoder
f_θ
$\mathbf{x}'$
An compressed low dimensional representation of the input.

Aka z is an embedding representation of the input image

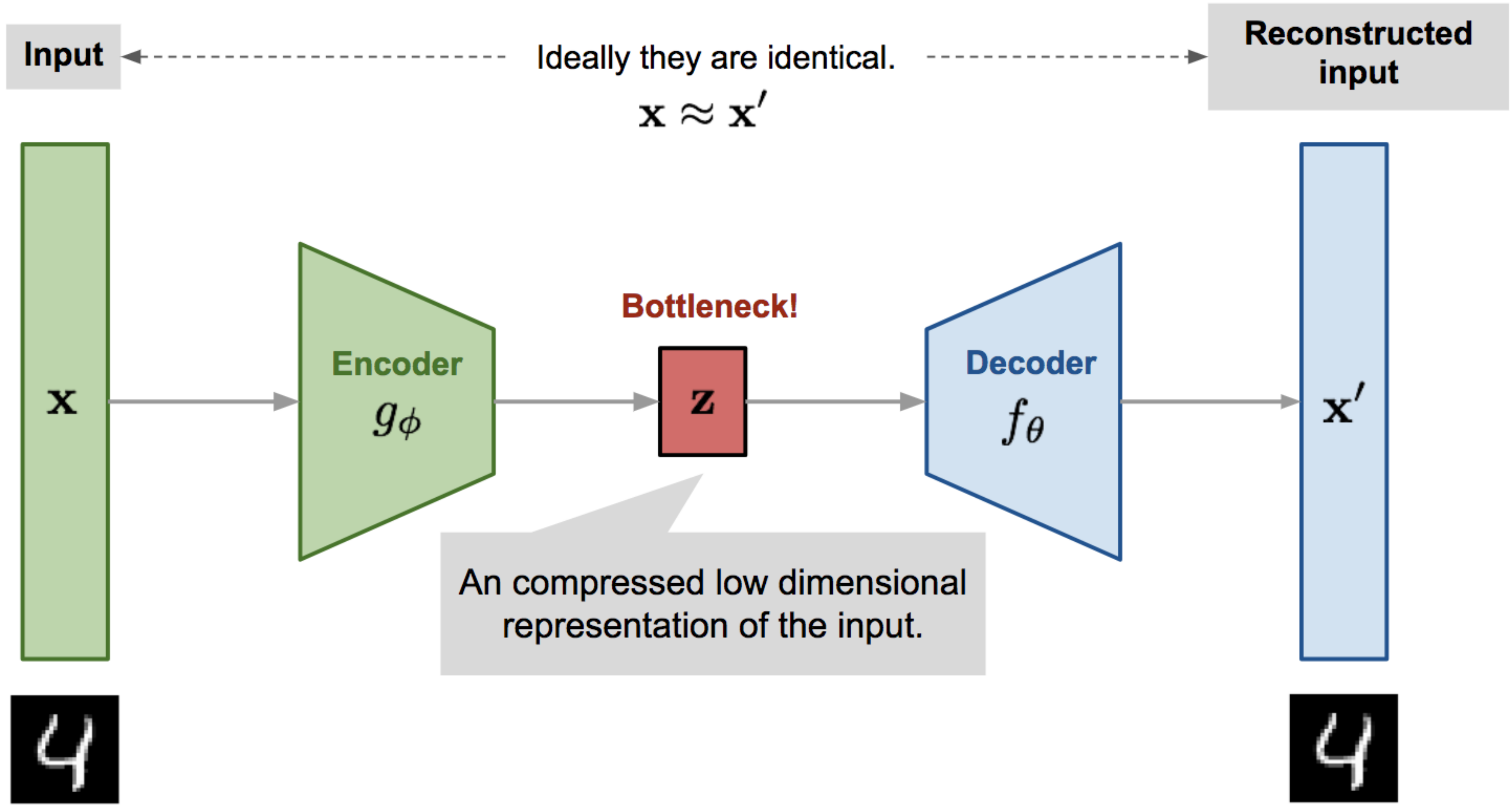
Foreshadowing: this sounds like transformer encoder/decoder!

https://lilianweng.github.io/posts/2018-08-12-vae/

Background: Autoencoders

- Question: suppose we had a model architecture like the one on the right. What is an appropriate loss function?
- Answer: mean-squared error between the input image and the reconstructed image (model output)! Aka pixel-error aka "reconstruction error".
  - Mean Squared Error (MSE) aka L2 norm
  - Mean Absolute Error (MAE): L1 norm

Fun fact: L1 norm tends to encourage sparsity in its output reconstruction errors (aka more 0 values, aka more exact matching), but L2 norm encourages overall fit. To learn more, see: [link]


Input
Ideally they are identical.
$\mathbf{x} \approx \mathbf{x}'$
Reconstructed input
$\mathbf{x}$
Encoder
g_ϕ
Bottleneck!
$\mathbf{z}$
Decoder
f_θ
$\mathbf{x}'$
An compressed low dimensional representation of the input.

Aka z is an embedding representation of the input image

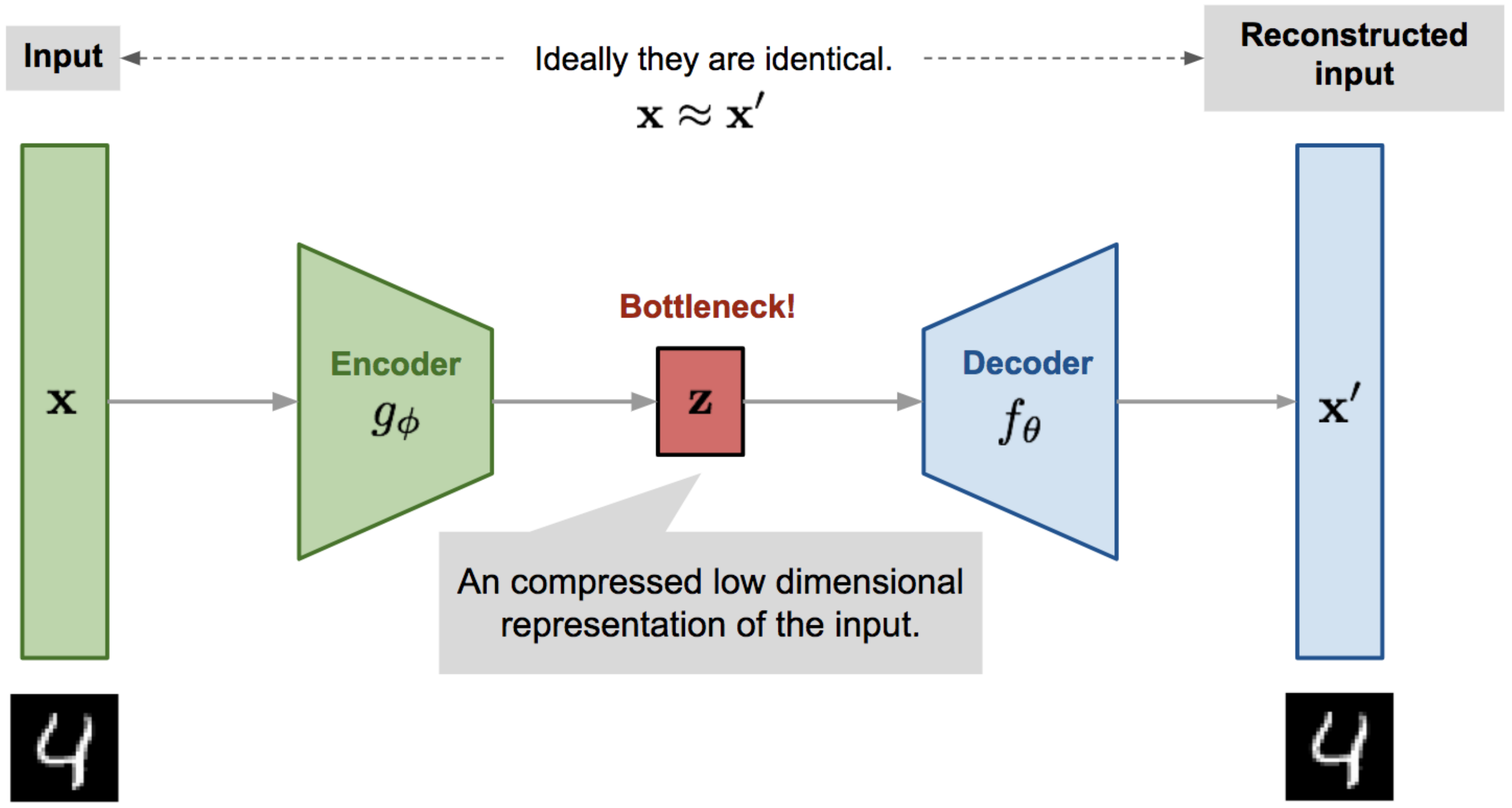
$$loss_{l2} = \|x - x'\|_2$$

$$loss_{l1} = \|x - x'\|_1$$

https://lilianweng.github.io/posts/2018-08-12-vae/

Training Autoencoders

- To train an autoencoder, we can easily construct a training dataset: we just need a source of images. No labeling required!
  - Ex: scrape Google Images / Pinterest / etc.
  - Given an image, the target is the image itself! Very nice.



Input

Ideally they are identical.

$\mathbf{x} \approx \mathbf{x}'$

Reconstructed input

$\mathbf{x}$

Encoder g_ϕ

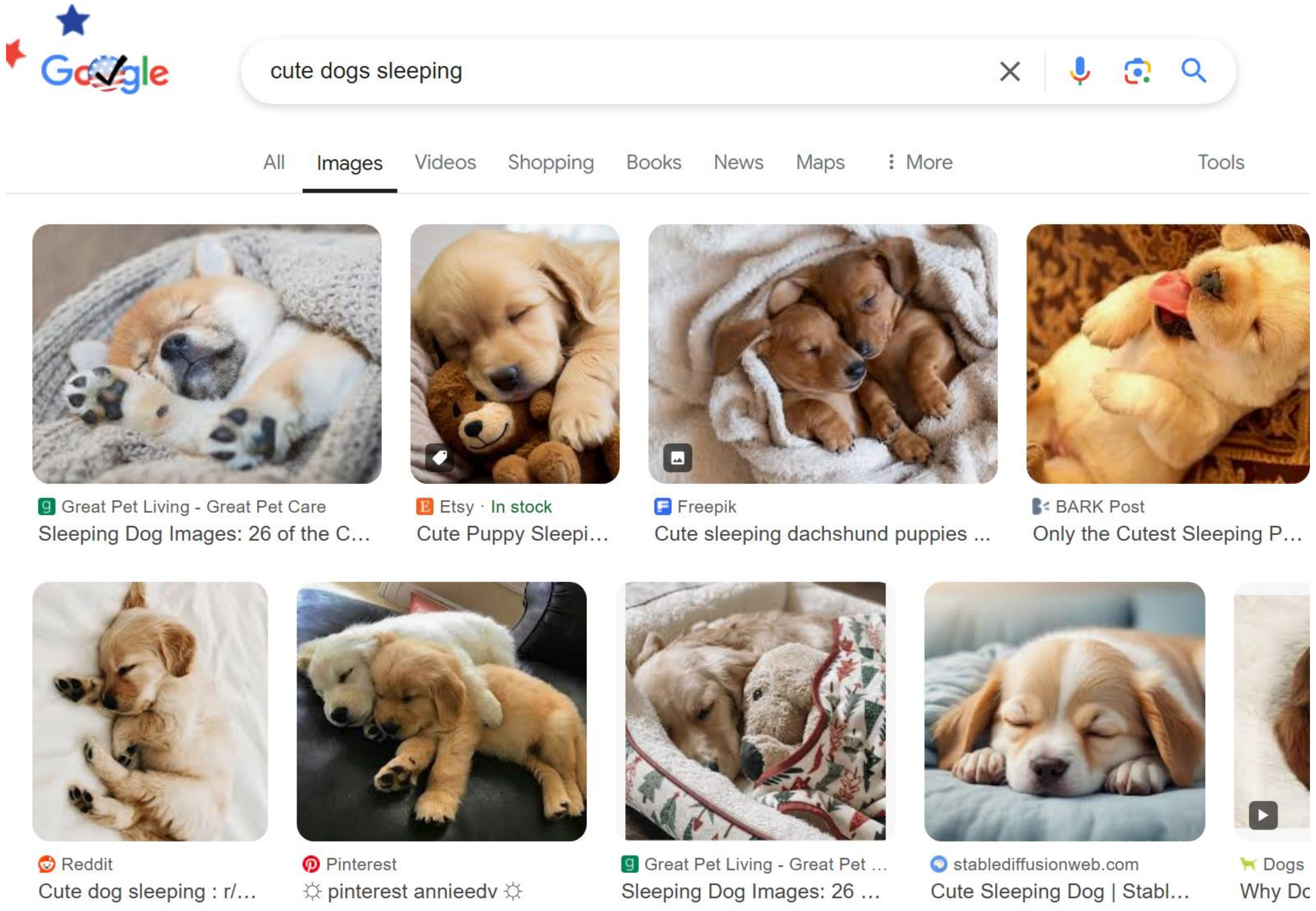
Bottleneck!

$\mathbf{z}$

Decoder f_θ

$\mathbf{x}'$

An compressed low dimensional representation of the input.



cute dogs sleeping

Great Pet Living - Great Pet Care
Sleeping Dog Images: 26 of the C...

Etsy · In stock
Cute Puppy Sleepi...

Freepik
Cute sleeping dachshund puppies ...

BARK Post
Only the Cutest Sleeping P...

Reddit
Cute dog sleeping : r/...

Pinterest
☼ pinterest annieedv ☼

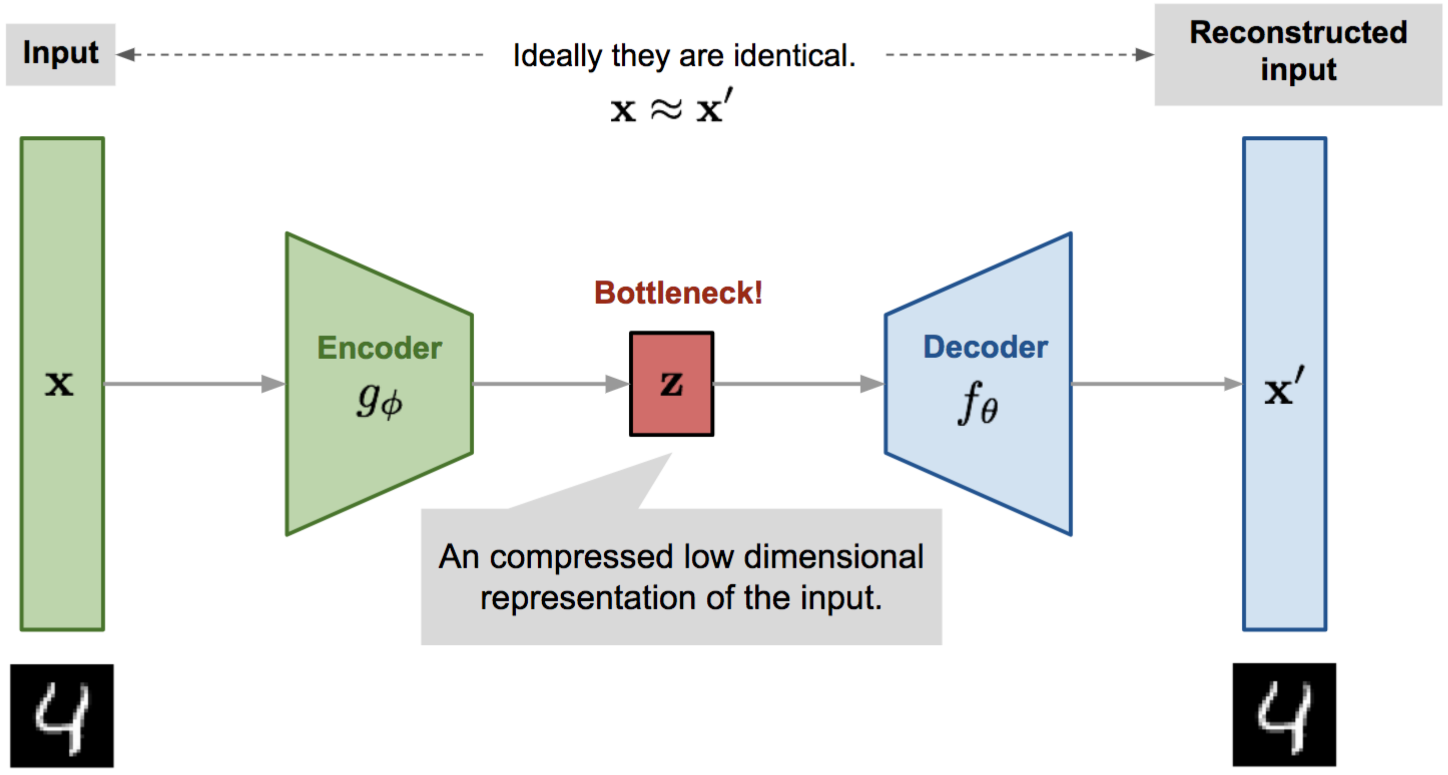
Great Pet Living - Great Pet ...
Sleeping Dog Images: 26 ...

stablediffusionweb.com
Cute Sleeping Dog | Stabl...

Dogs
Why Dc

A simple autoencoder model architecture

- Question: design a model architecture that given an image (say, a 224x224 RGB image), implements the autoencoder idea.
- Answer: many possible, but here's one:

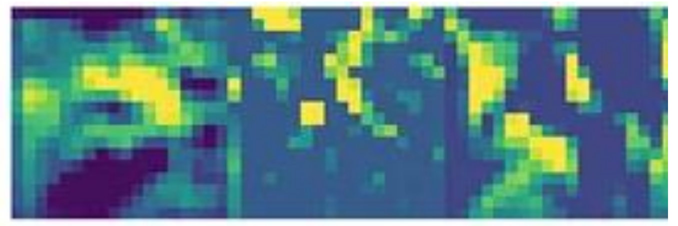

Input
Ideally they are identical.
$\mathbf{x} \approx \mathbf{x}'$
Reconstructed input
$\mathbf{x}$
Encoder
g_ϕ
Bottleneck!
$\mathbf{z}$
Decoder
f_θ
$\mathbf{x}'$
An compressed low dimensional representation of the input.

Ex: a resnet conv block (repeated conv2d, relu, batchnorm)

➡ Conv Block ➡ Conv Block ➡

After each block, we downsample the featmaps by 2x

CNN feat maps.
shape=[chans, f_height, f_width]



➡ Reshape ➡

Embedding z

0.1	0.3	...	-1.2

Shape:
[chans*f_height*f_width]

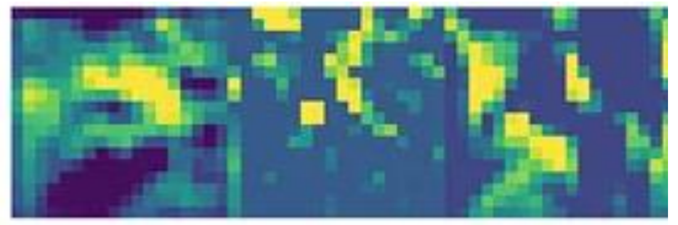
Encoder

Embedding z

0.1	0.3	...	-1.2

Shape:
[chans*f_height*f_width]

➡ Reshape ➡



Shape: [chans, f_height, f_width]

➡ Upsample Block ➡ Upsample Block ➡

After each block, we **upsample** the featmaps by 2x

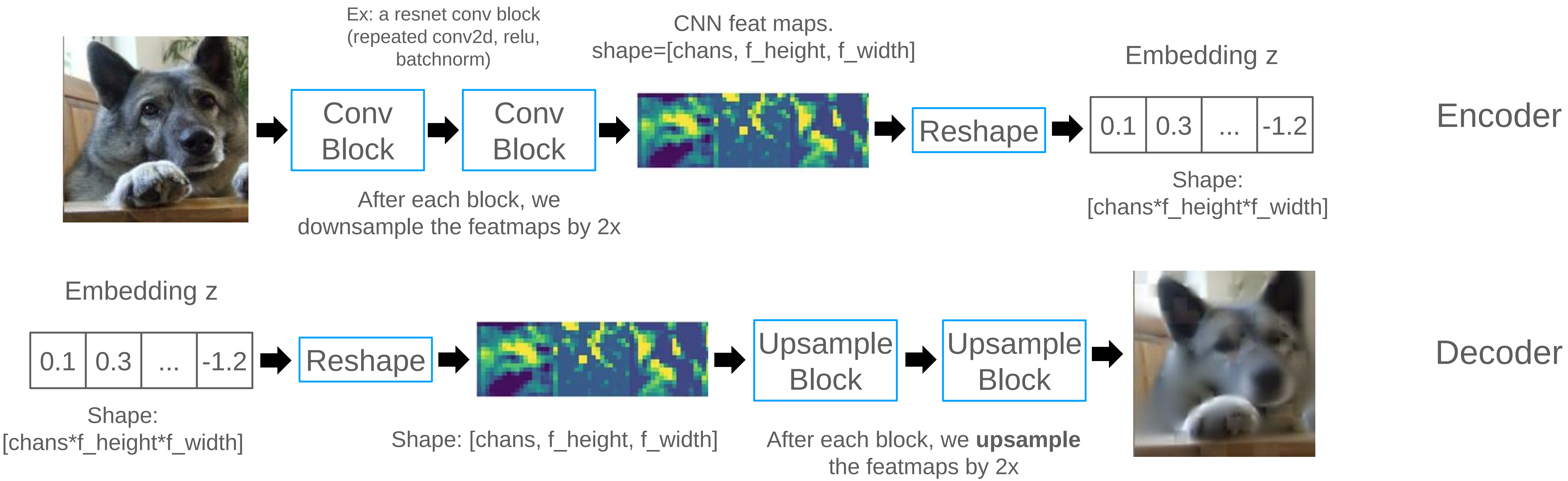
Our reconstructed output!

Decoder

Note: technically we don't need to do the Reshape, but I'll do it here to be consistent with the above figure

Transposed Convolution

- How to implement the "Upsample block"?
- Transposed Convolutions, aka "learned upsampling" (pytorch: torch.nn.ConvTranspose2d [link])



Ex: a resnet conv block (repeated conv2d, relu, batchnorm)

CNN feat maps. shape=[chans, f_height, f_width]

Embedding z

Conv Block → Conv Block → Reshape →

0.1	0.3	...	-1.2

Encoder

After each block, we downsample the featmaps by 2x

Shape: [chans*f_height*f_width]

Embedding z

0.1	0.3	...	-1.2

→ Reshape → Upsample Block → Upsample Block →

Decoder

Shape: [chans*f_height*f_width]

Shape: [chans, f_height, f_width]

After each block, we **upsample** the featmaps by 2x

More resources on Transposed Conv: [link1] [link2]

A simple autoencoder model architecture (pytorch)

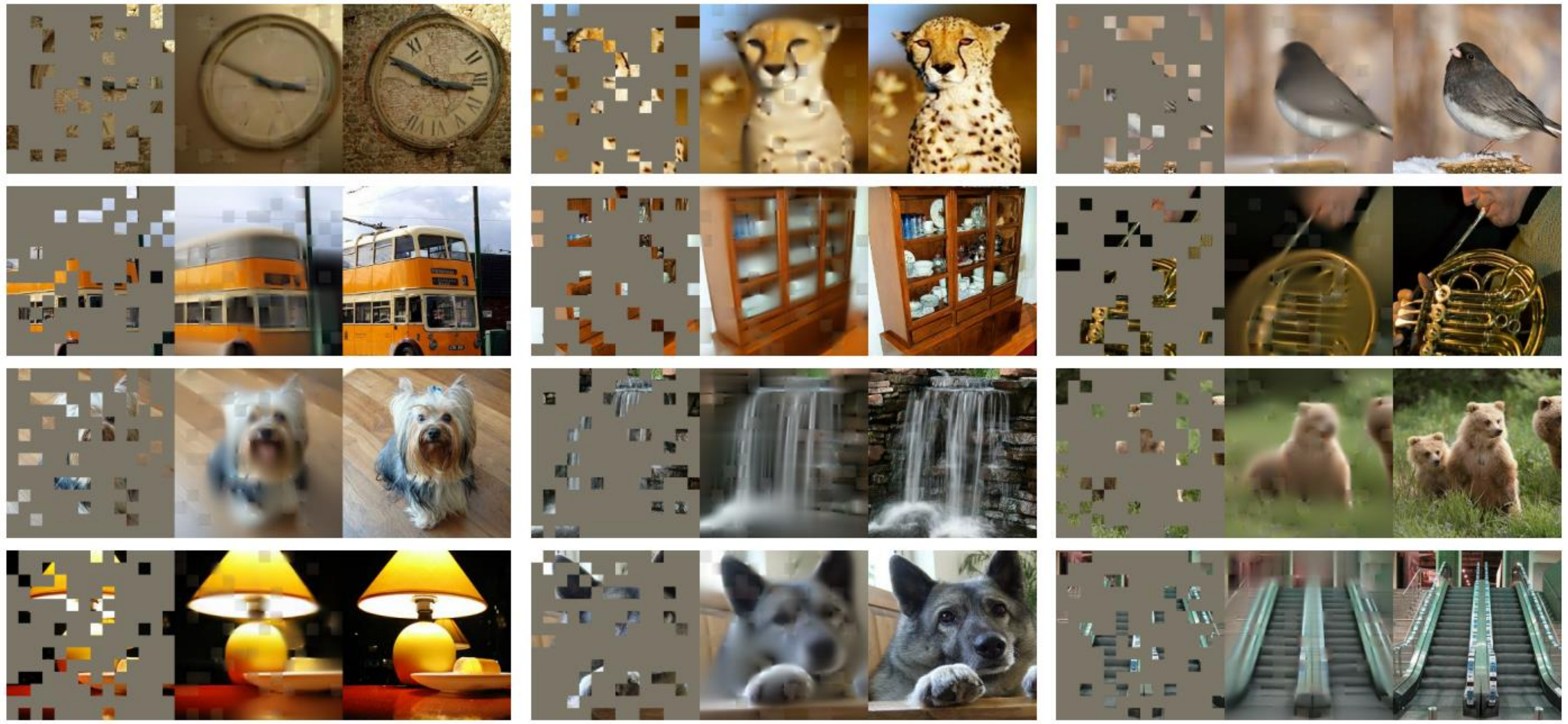
```python
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 16, 3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(16, 32, 3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(32, 64, 7)
        )
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(64, 32, 7),
            nn.ReLU(),
            nn.ConvTranspose2d(32, 16, 3, stride=2, padding=1, output_padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(16, 1, 3, stride=2, padding=1, output_padding=1),
            nn.Sigmoid()
        )

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```

Fun fact: we can also do non-learnable upsampling (eg standard 2d interpolation like nearest-neighbor, linear-interp, etc) in pytorch, and even backprop through them! https://pytorch.org/docs/stable/generated/torch.nn.Upsample.html

https://www.cs.toronto.edu/~lczhang/321/lec/autoencoder_notes.html

Image masking

- Twist on the image reconstruction task: rather than reconstruct the entire image (as in classical autoencoders), let's do the "fill in the blank" task for images!
  - Aka "Masked autoencoder"



For each group: (**left**) image with blanked-out pixels (**middle**) model predictions (**right**) ground truth image

Masked autoencoders (MAE) (2022)

- "Masked Autoencoders Are Scalable Vision Learners" (Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollar, Ross Girshick), CVPR 2022
- Model arch: transformer encoder, decoder

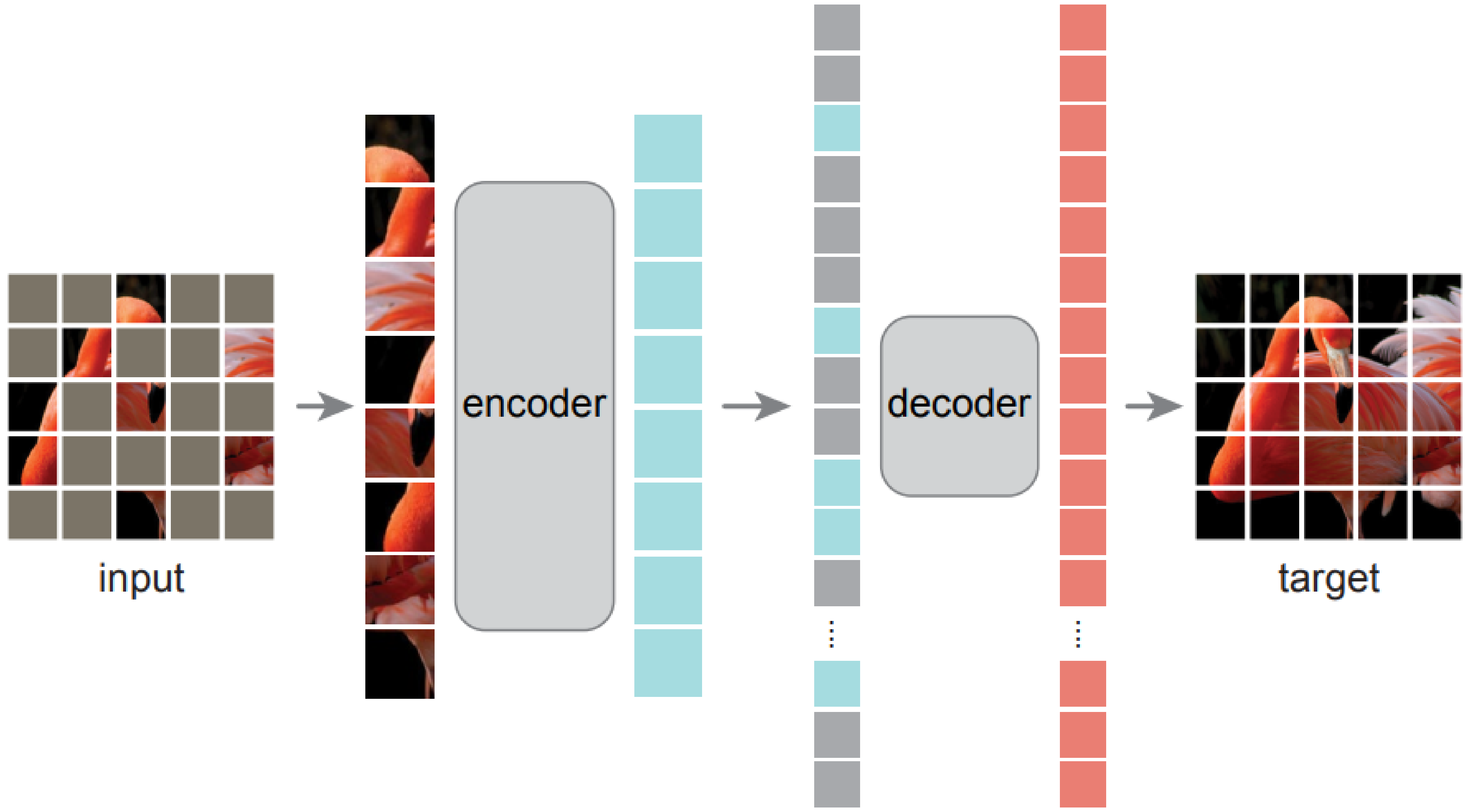


Figure 1. **Our MAE architecture**. During pre-training, a large random subset of image patches (*e.g.*, 75%) is masked out. The encoder is applied to the small subset of *visible patches*. Mask tokens are introduced *after* the encoder, and the full set of encoded patches and mask tokens is processed by a small decoder that reconstructs the original image in pixels. After pre-training, the decoder is discarded and the encoder is applied to uncorrupted images (full sets of patches) for recognition tasks.

MAE training methodology

- First phase: self-supervised training on "fill in the patch" task on ImageNet-1k
  - Special care to deal with masking in encoder and decoder (read paper for details)
- Second phase: take transformer encoder from the first phase, and train it on image classification
  - Notably: discard the decoder!
- Result: achieved state-of-the-art results in ImageNet-1k **for models that have only seen ImageNet-1k**

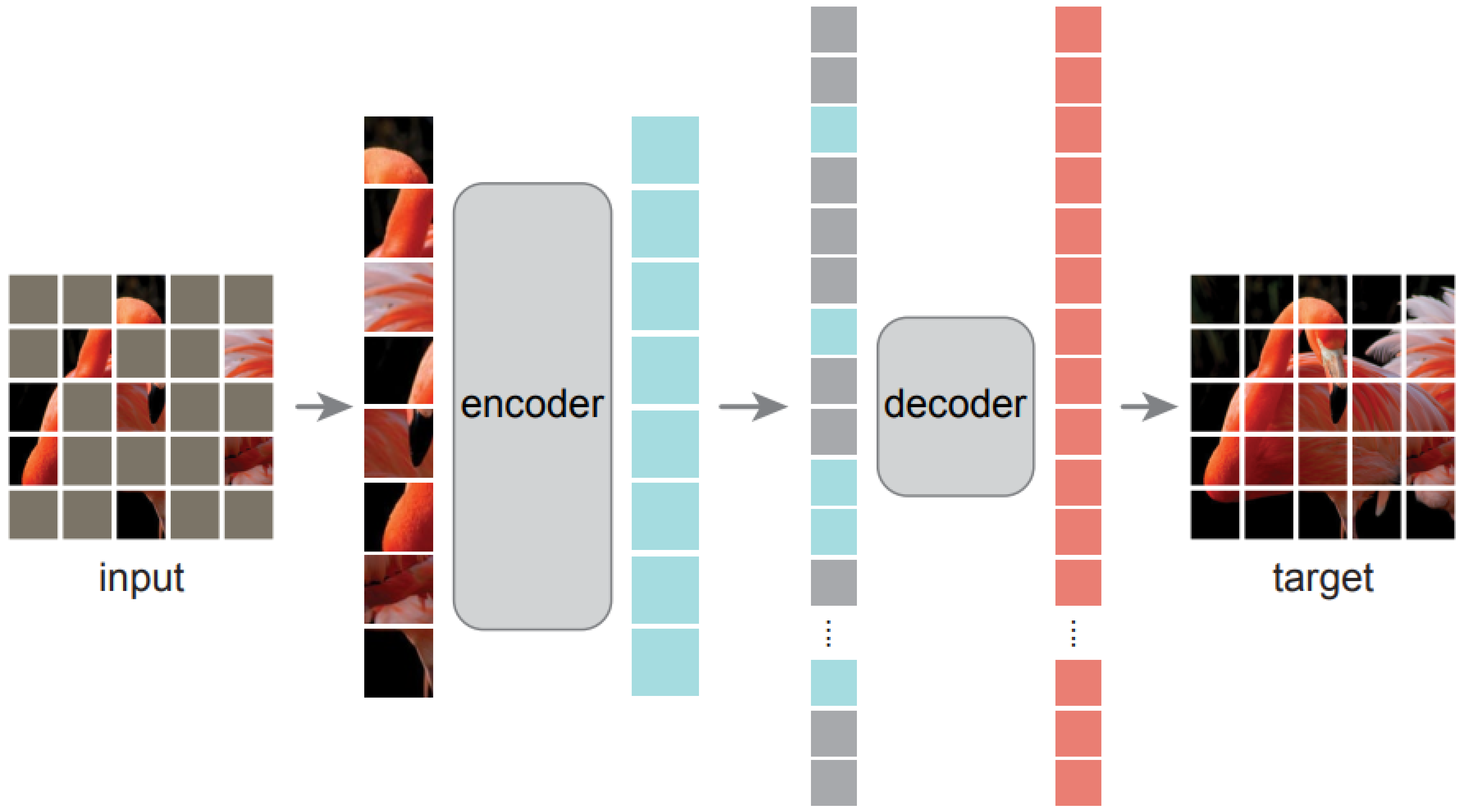


Figure 1. **Our MAE architecture**. During pre-training, a large random subset of image patches (*e.g.*, 75%) is masked out. The encoder is applied to the small subset of *visible patches*. Mask tokens are introduced *after* the encoder, and the full set of encoded patches and mask tokens is processed by a small decoder that reconstructs the original image in pixels. After pre-training, the decoder is discarded and the encoder is applied to uncorrupted images (full sets of patches) for recognition tasks.

This qualification is important, as other papers (like ViT) first pretrained on large external datasets like JFT-300M prior to fine-tuning on ImageNet-1k

MAE takeaways

- This paper justifies the following strategy:
- Large-scale pretraining. First, take a large-capacity model (eg ViT-L), and pretrain it on a gigantic self-supervised task like "fill in the patch"
  - Produces a "foundational" model, suitable for downstream usecases
  - Pro: easy to collect this dataset!
  - Con: large-scale pretraining requires a LOT of compute and $
- Task-specific fine-tuning. Take your resulting foundational model, and train it on your desired task (eg ImageNet-1k image classification).
  - Intuition: rather than starting your model weights from scratch (eg random init), we start the model weights from a strong starting point.
  - Tricks: to accelerate this stage, can freeze early model layers

Effectiveness of Large-scale pretraining

- Large-scale pretraining + fine-tuning is extremely effective, and is widely used in both academia and industry
- Pinterest: "Billion-Scale Pretraining with Vision Transformers for Multi-Task Visual Representations" (2022) (Josh Beal, Hao-Yu Wu, Dong Huk Park, Andrew Zhai, Dmitry Kislyuk) [link]
    - Use ML models to automatically generate large "weakly labeled" image dataset
- Facebook: "Exploring the Limits of Weakly Supervised Pretraining" (Mahajan et al) (2018) [link]
    - Hashtag prediction on Instagram images

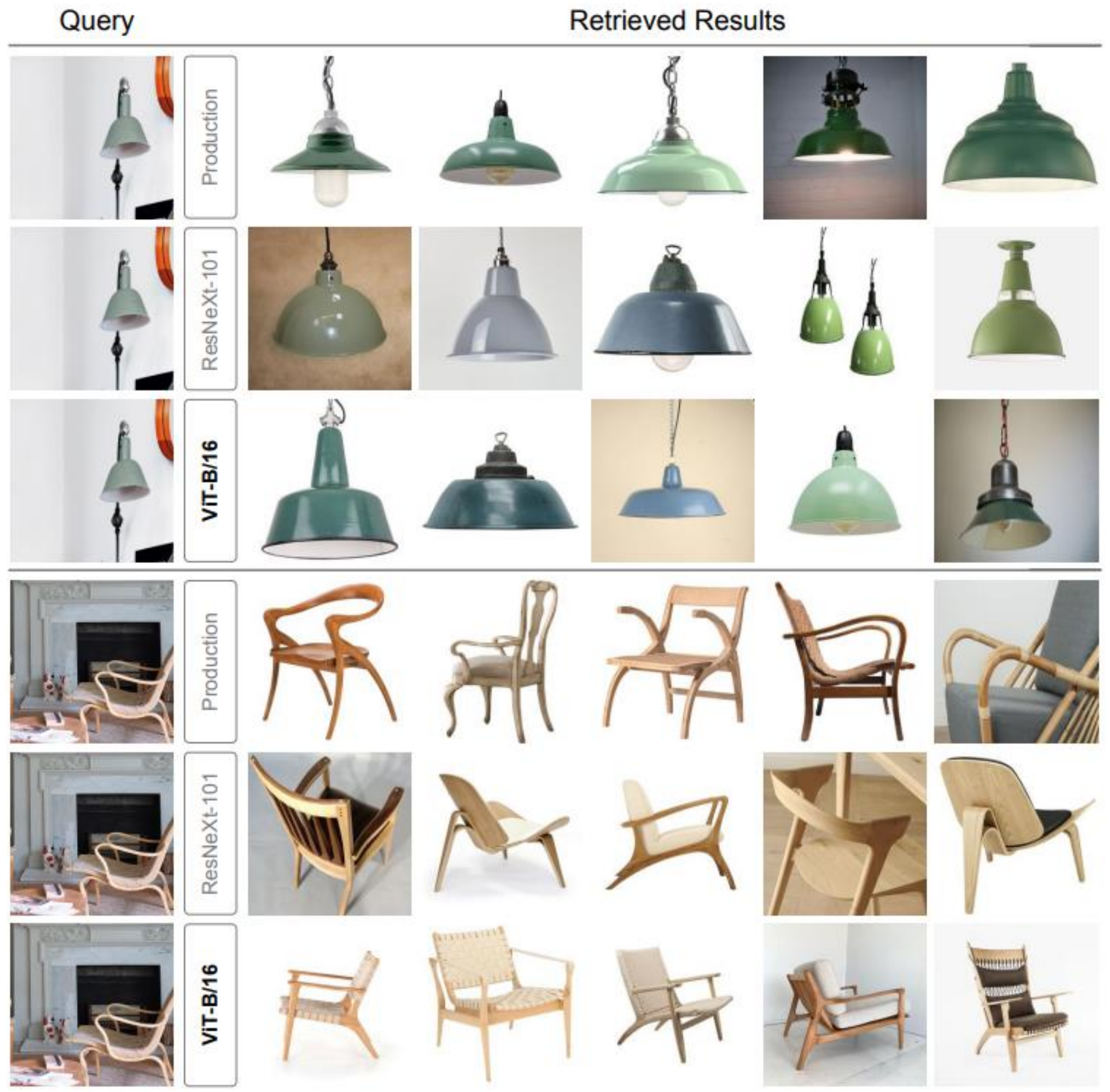


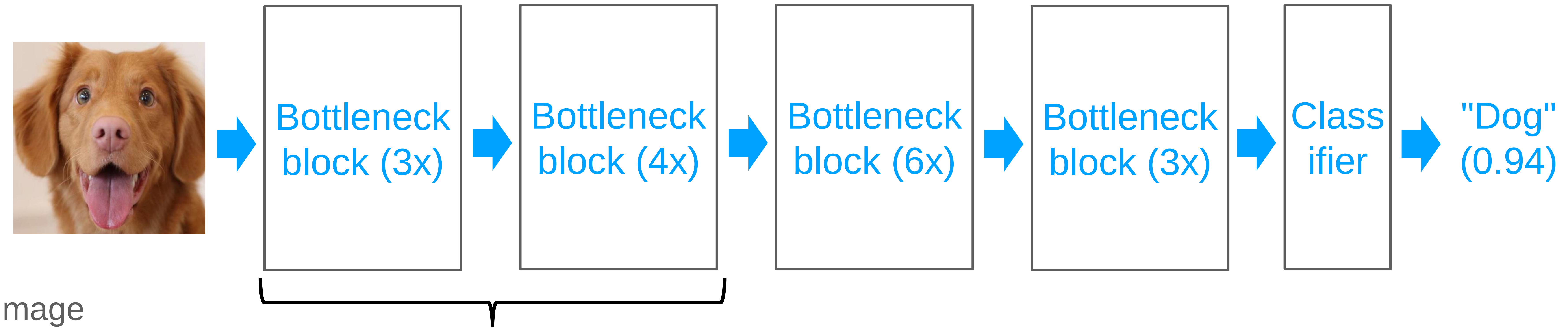
Figure 8: Example of retrieval results using the control (production) model, the ResNeXt-101 Annotations-1.3B model, and the ViT-B/16 Annotations-1.3B model. The ViT model generally matches more similar product results.

Aside: Transfer learning

- You'll see the term "transfer learning" used a lot in this context. Transfer learning (in DNN's) generally means: let's train a DNN on TaskA (ex: image classification, or fill-in-the-blank), and then fine-tune it on TaskB.
- Transfer learning is a successful technique that predates large-scale pretraining (which can be seen as a version of transfer learning)

Tangent: model freezing

- When we load a pretrained model and train it on another task ("fine tune"), we can decide which layers of the model to learn, and which to keep fixed ("frozen")!
- **Intuition**: in CNN's, the early Conv layers are responsible for low-level image features (edges, etc). When fine-tuning for image classification we can get away with freezing the early Conv layers and only learning the final few Conv blocks.
    - In practice: this leads to substantial training speedup with little downstream performance drop! (can often train for fewer epochs, and each epoch is faster + requires less GPU memory)



Bottleneck block (3x)

Bottleneck block (4x)

Bottleneck block (6x)

Bottleneck block (3x)

Classifier

"Dog" (0.94)

ResNet50 (Image classification arch [link])

Frozen layers (aka "don't do backprop through these")

Unfrozen layers (aka "backprop through these")

Tangent: model freezing (pytorch)

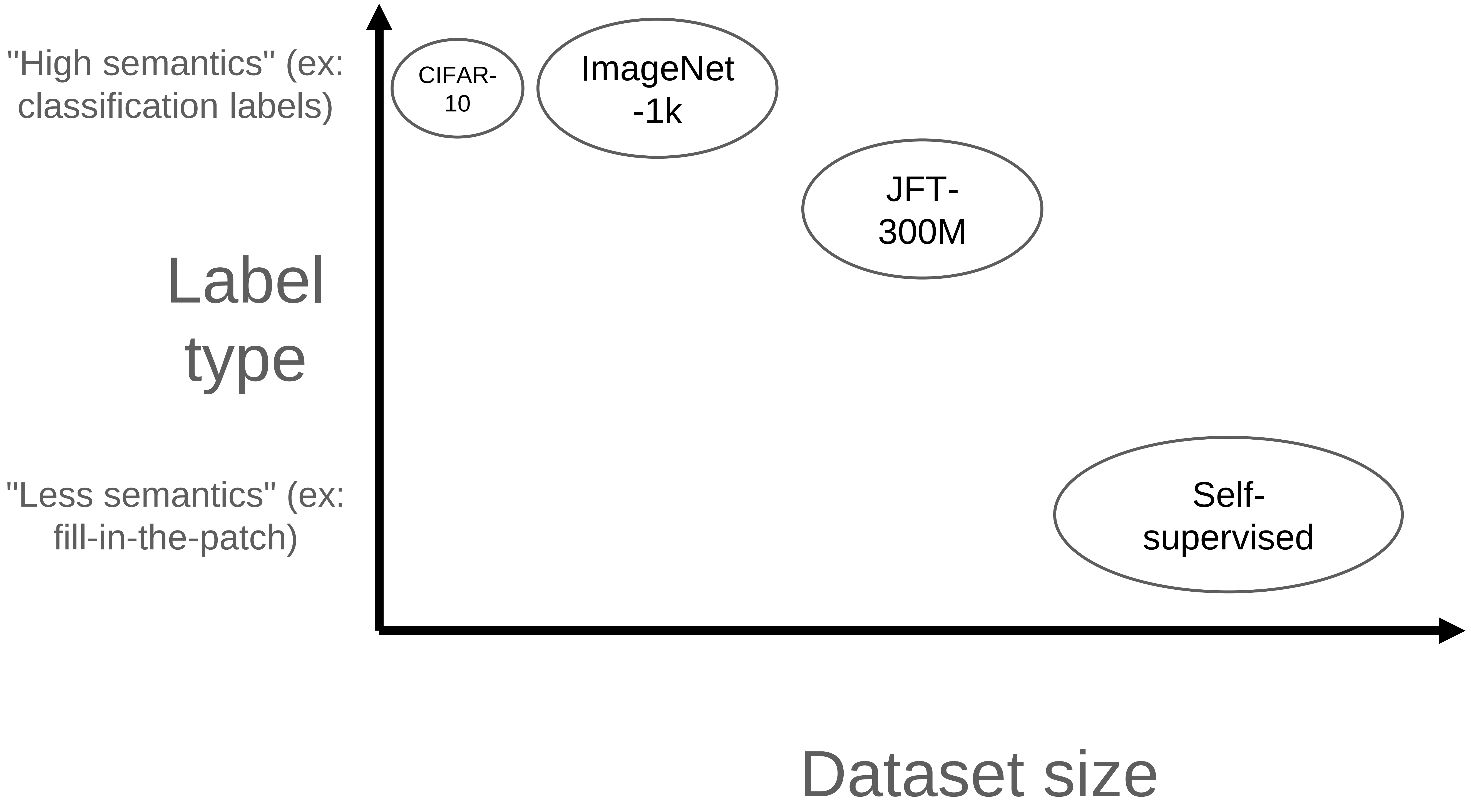
- Fortunately, in pytorch freezing layer(s) of a model is easy!
- Implementation tip: by organizing your torch.nn.Module's nicely (eg composing Modules together), you can make it easier to freeze each layer, vs manually iterating over all model parameters and doing annoying bookkeeping to determine which parameter belongs to which layer, etc...

```
for param in model.parameters():
    # Freeze, aka don't learn this parameter
    param.requires_grad = False
```

Note: CIFAR-10 has 50k training images [link]

(unused) Dataset tradeoff: scale vs semantics

- What is more important: semantics (eg classification labels)


"High semantics" (ex: classification labels)
CIFAR-10
ImageNet -1k
JFT-300M
Label type
"Less semantics" (ex: fill-in-the-patch)
Self-supervised
Dataset size

(Not too scale)

(unused) Self-supervised + Contrastive learning

- Another way: Contrastive learning
- Idea: take image X, corrupt it via X', and learn an image embedding representation that is robust to corruption, eg `dist(f(X), f(X'))` is small
- Ex: [link]

Tangent: "Train ImageNet in 1 hour"

- In "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour" (2018) [link] the authors, using a large distributed GPU cluster, trained a ResNet-50 model on ImageNet-1k "from scratch" in one hour. A neat accomplishment for that time!
  - Hardware: 256 GPUs ("Big Basin" [link] GPU cluster internal to Facebook, 16GB GPU mem per card. Nvidia Tesla P100).
- Learning: when scaling up the number of GPUs (aka increasing the effective batchsize), one must adjust the learning rate accordingly ("linear scaling rule").
  - Rule: double the batchsize -> double the learning rate.

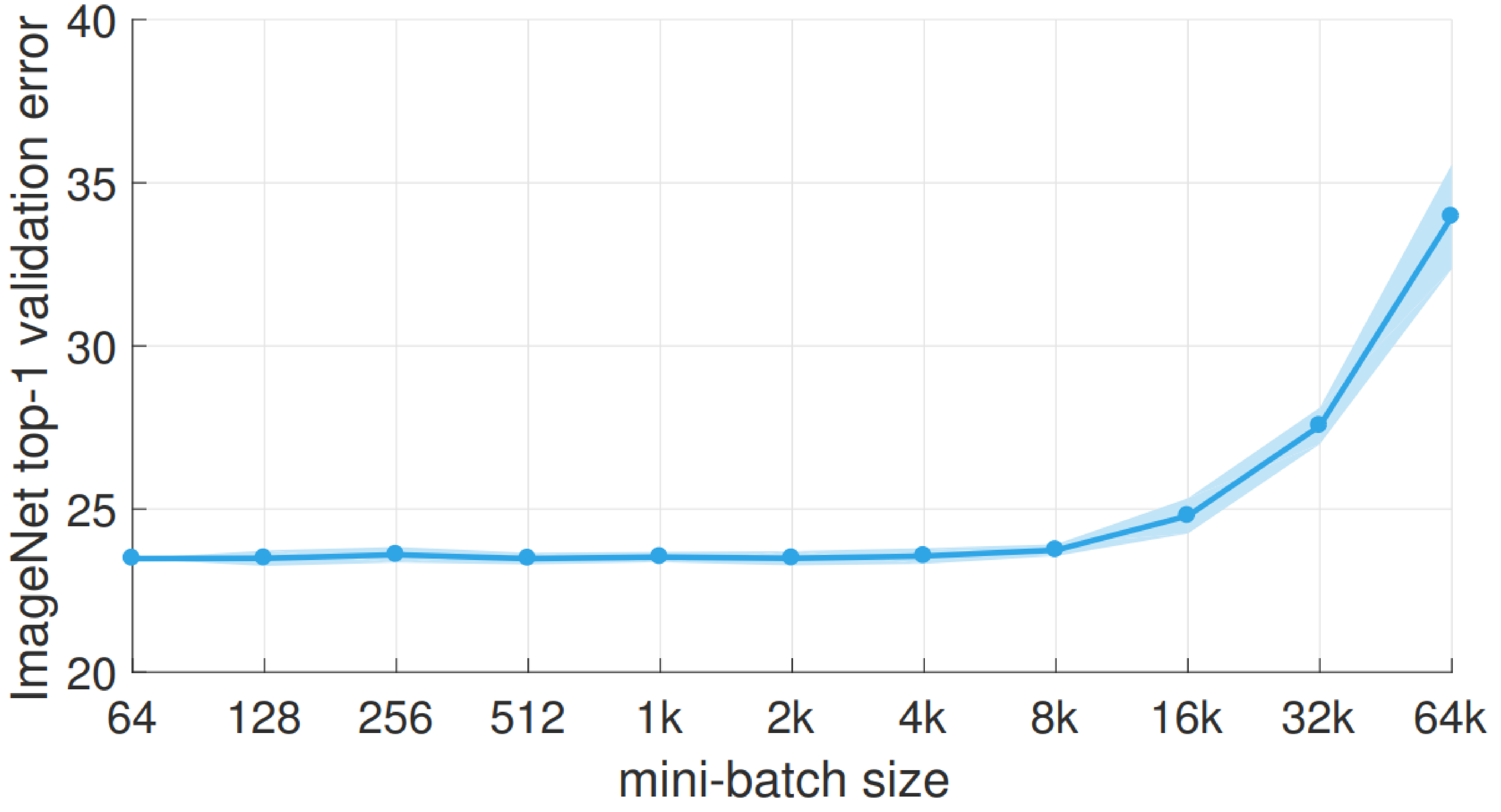
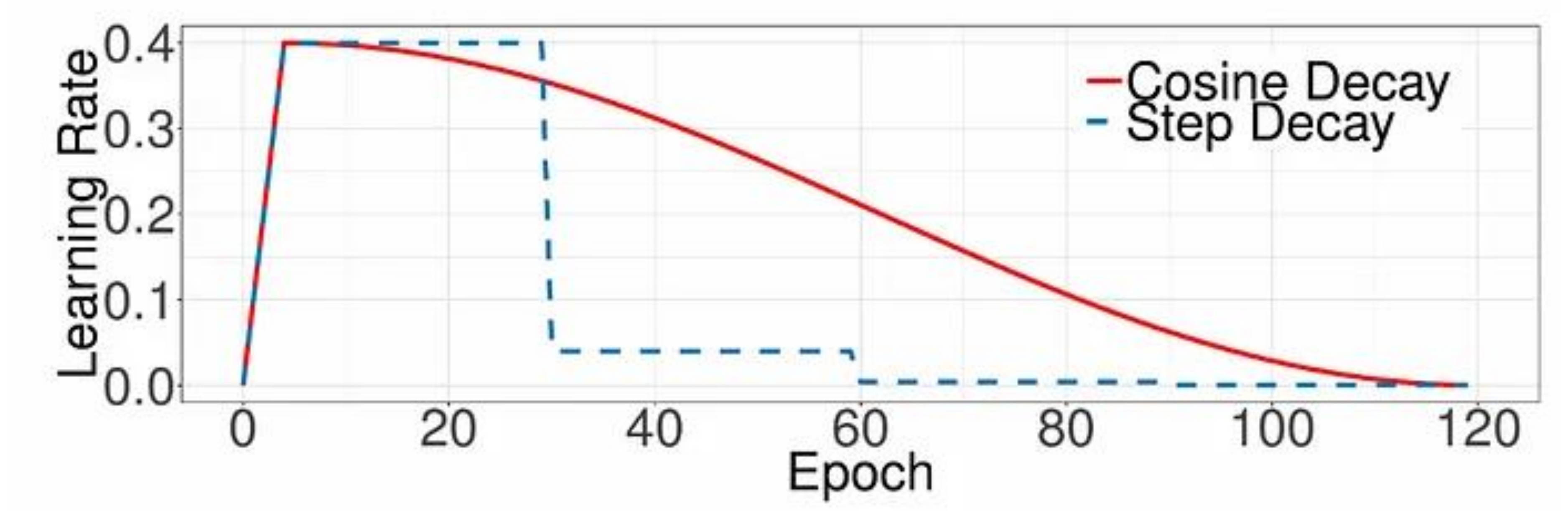


Figure 1. **ImageNet top-1 validation error *vs*. minibatch size.** Error range of plus/minus *two* standard deviations is shown. We present a simple and general technique for scaling distributed synchronous SGD to minibatches of up to 8k images *while maintaining the top-1 error of small minibatch training*. For all minibatch

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 (eg 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).



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

Tangent: "Train ImageNet in 1 hour"

- A sign that GPU hardware (and DNN libraries + distributed training frameworks) is advancing quickly
- **...And, a hint that ImageNet-1k is starting to feel small!**
- (later in Aug 2018, someone showed we can train ImageNet in 18 mins for $40 using AWS cloud! [link])
- In 2024, I bet things are even faster + cheaper! Technology marches on...

(Tangent tangent) gradient quality vs num steps?

- Observation: if you keep the number training epochs fixed, then increasing the batchsize leads to fewer model updates.
  - Higher batchsize -> higher quality gradient updates, but fewer parameter updates
  - Lower batchsize -> noisier gradient updates, but more parameter updates.
- What is best? Paper's answer: higher batchsize AND higher learning rate.
  - ...to a point. Beyond batchsize=8k, classification error **starts increasing**.

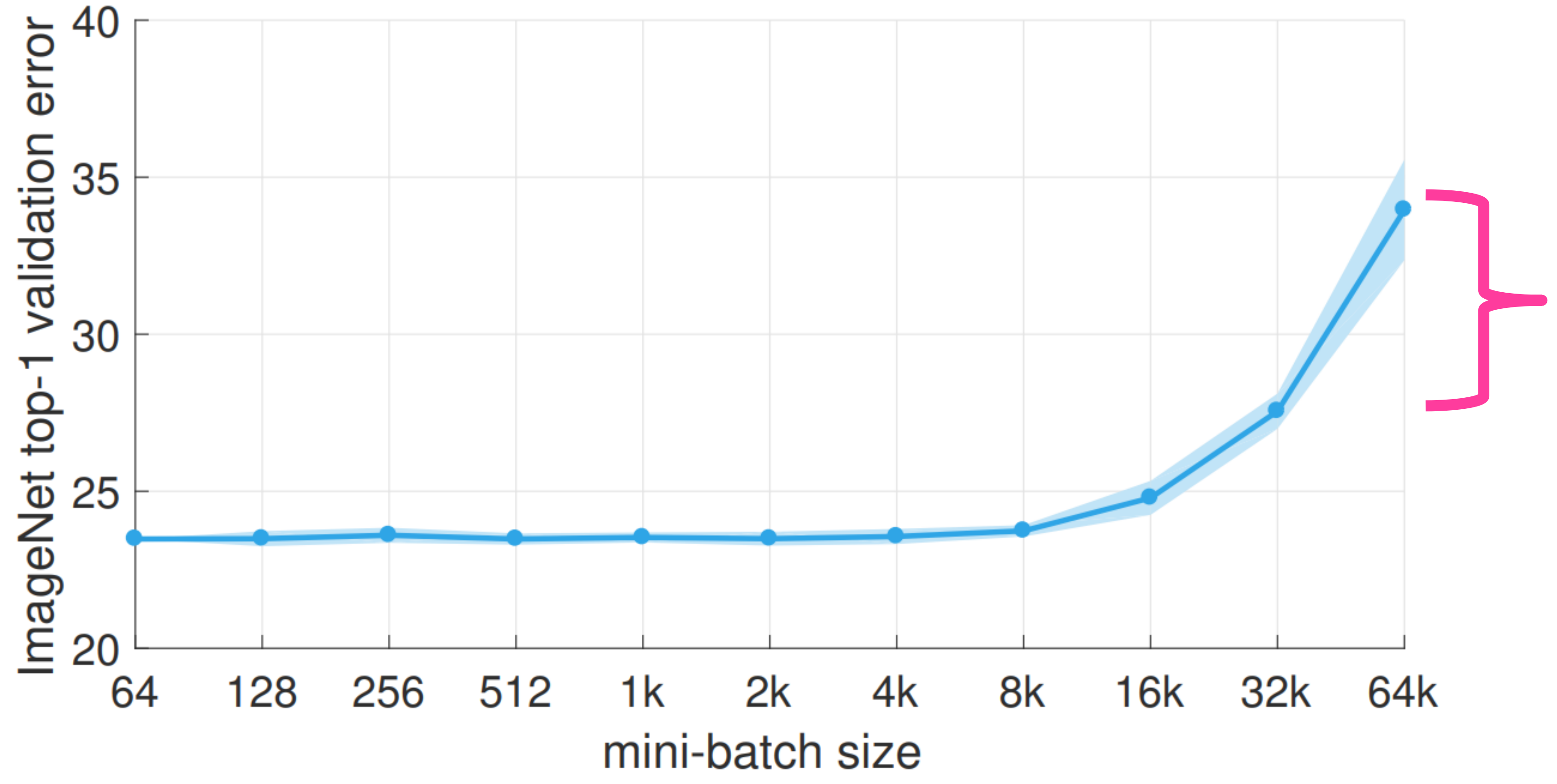


Figure 1. **ImageNet top-1 validation error *vs.* minibatch size.** Error range of plus/minus *two* standard deviations is shown. We present a simple and general technique for scaling distributed synchronous SGD to minibatches of up to 8k images *while maintaining the top-1 error of small minibatch training*. For all minibatch