Data 188: Introduction to Deep Learning

Masked Autoencoders, large-scale pretraining

Speaker: Eric Kim

Lecture 19 (Week 12)

2026-04-09, Spring 2026. UC Berkeley.

Announcements

- HW3 ongoing: "Intro to Pytorch"
- HW4 will be released soon!
    - Transformers: Machine translation (NLP)

Today's lecture

Masked auto-encoder

Large-scale pretraining (computer vision)

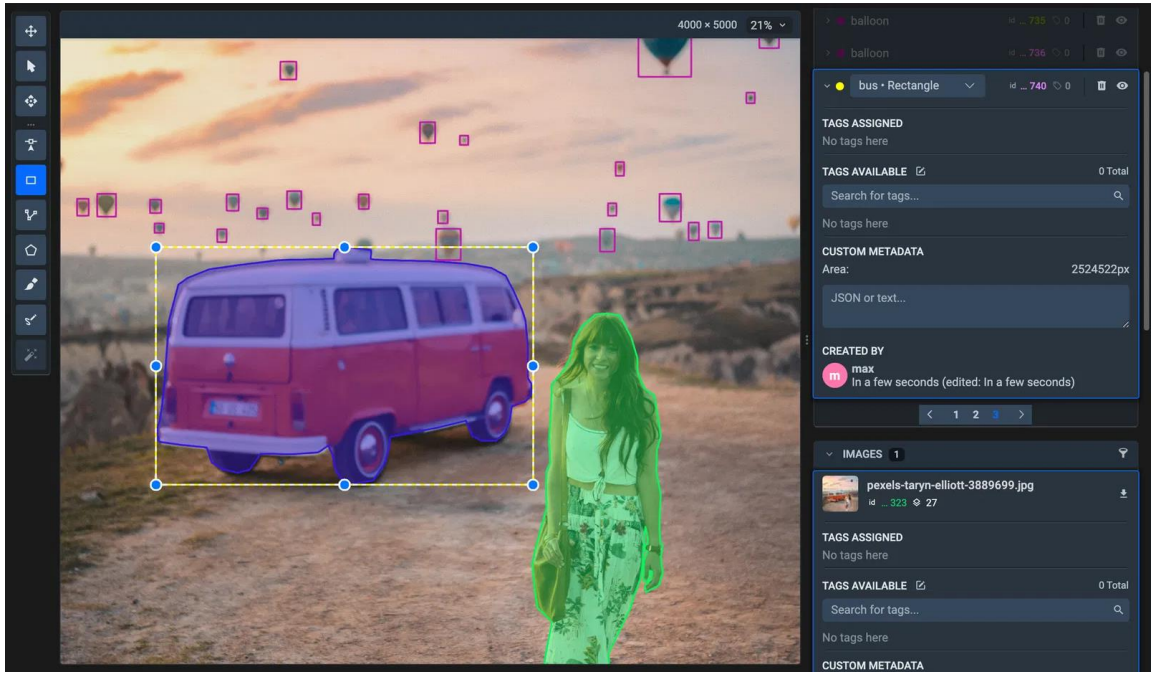
Scaling up datasets: challenges

The primary challenge with scaling up datasets like ImageNet: getting high-quality human labels at scale is very expensive, both in terms of $, time, and 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?**



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

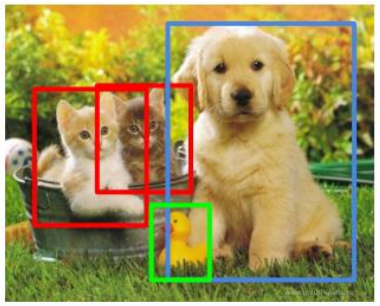
Learning paradigms

So far, we've focused on **supervised training**: each dataset 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

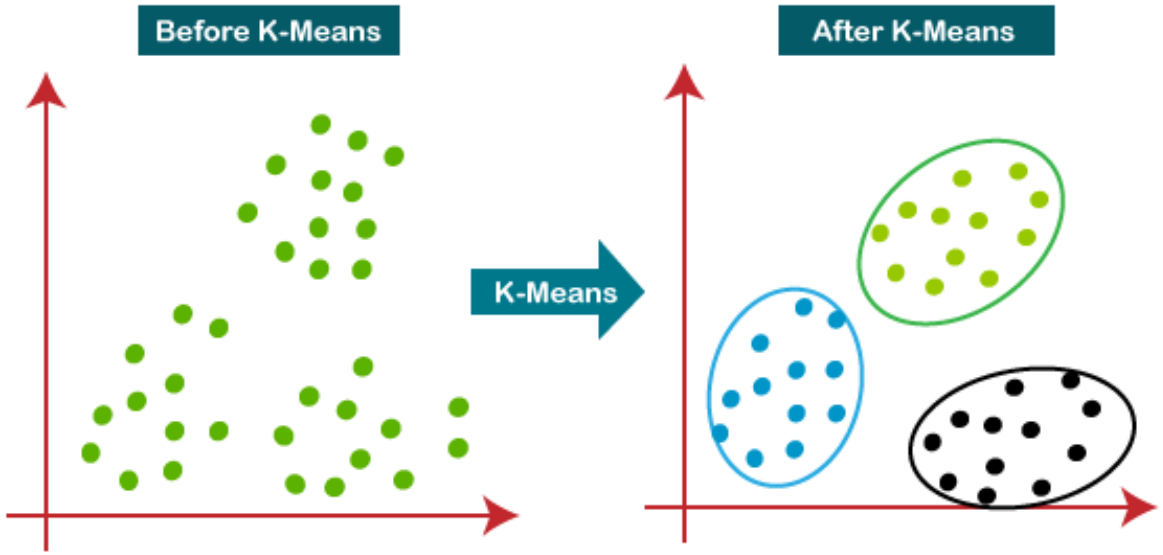
I am sleepy (EN) -> j'ai sommeil (FR)

Translation: Text for language A and language B

Unsupervised learning

Unsupervised: dataset has NO labels

- Ex: clustering algorithms (k-means)
- Ex: classification (k nearest neighbors)



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. (aka "cloze" task [link])

Ex: "Today, I went to the _____ and bought some milk and eggs."

Target label: "store"

Pro: it's easy to scrape text data from the internet, and extremely easy to construct cloze examples. ("unlimited" training data for free! Ex: scrape wikipedia)

Why self-supervised?

Motivation: pretraining a model on large self-supervised datasets leads to a stronger "starting point" for downstream applications (eg classification)

Intuition: large-scale pretraining lets the model learn to "understand" the input distribution (ie visual/text world). Then, subsequent fine-tuning starts off from a "strong" starting point.

Very similar in spirit to **transfer learning**.

```
# 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)
```

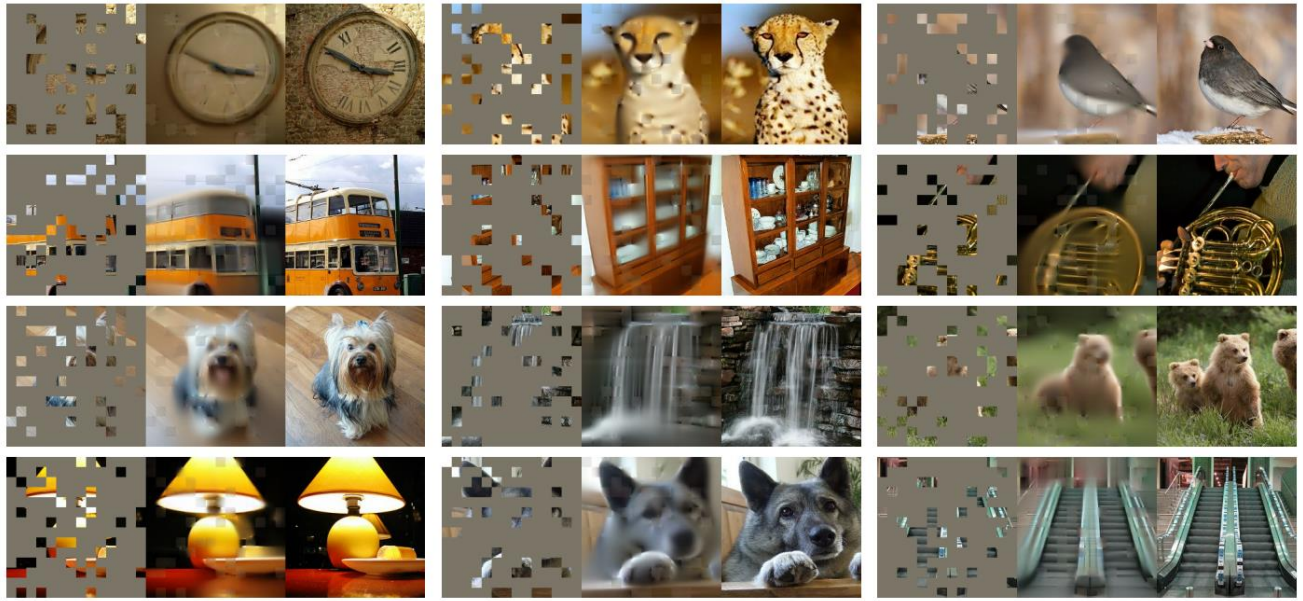
Pytorch documentation on model.save(), model.load(): [link]

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

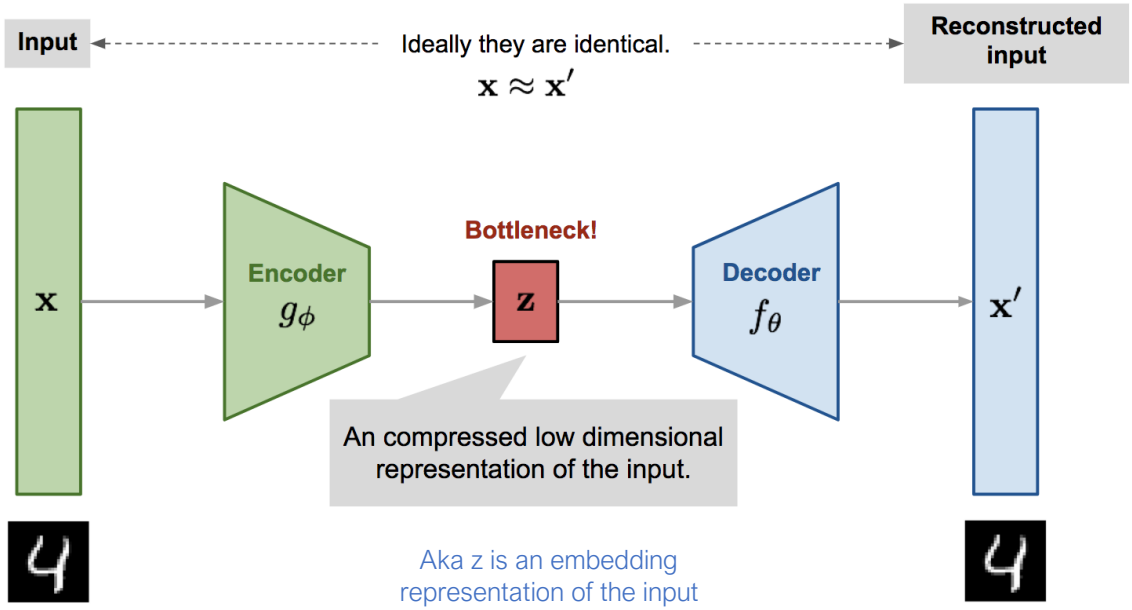
Background: Autoencoders

Autoencoders are a standard technique in AI/ML.

Idea: transform an input (ie image, text) into a latent representation (aka embedding), and then reconstruct the input from the latents

Encoder: Given input, transform into an embedding(s) (aka latent)

Decoder: Given latent representation, reconstruct original input.



Aka z is an embedding representation of the input image

This sounds like transformer encoder/decoder!

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

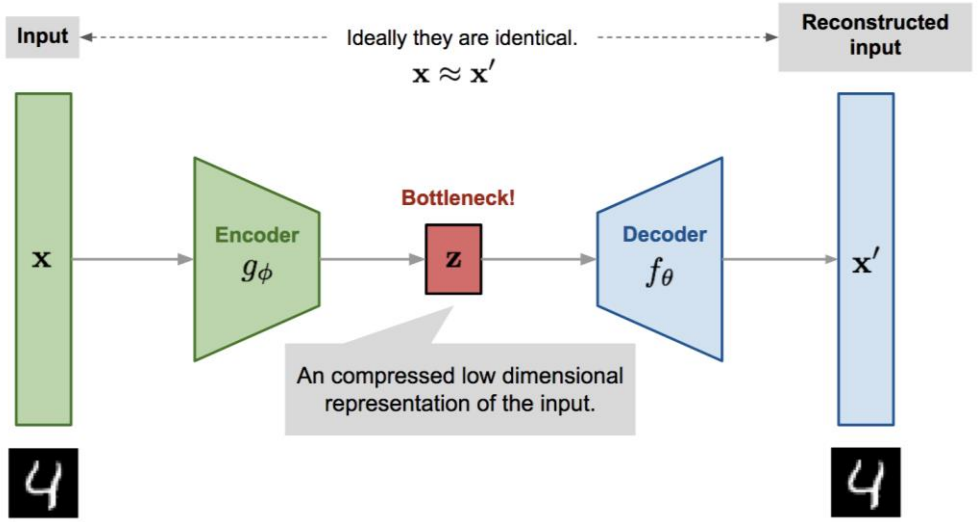
Background: Autoencoders

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]



Aka z is an embedding representation of the input image

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

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

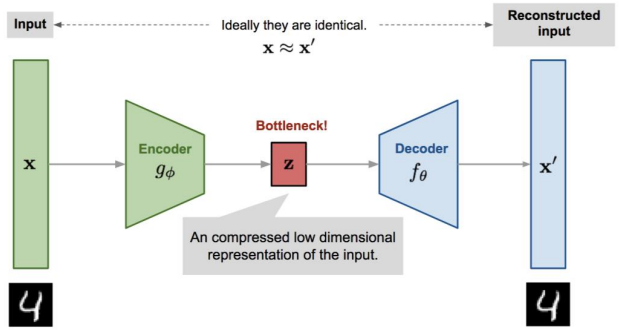
https://lilianweng.github.io/posts/2018-08-12-vae/

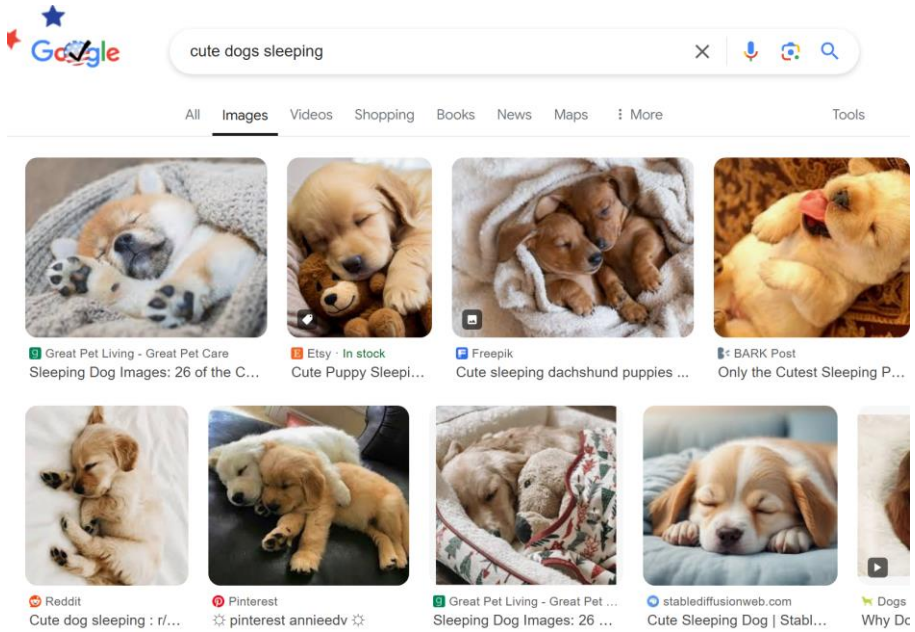
Training Autoencoders

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.

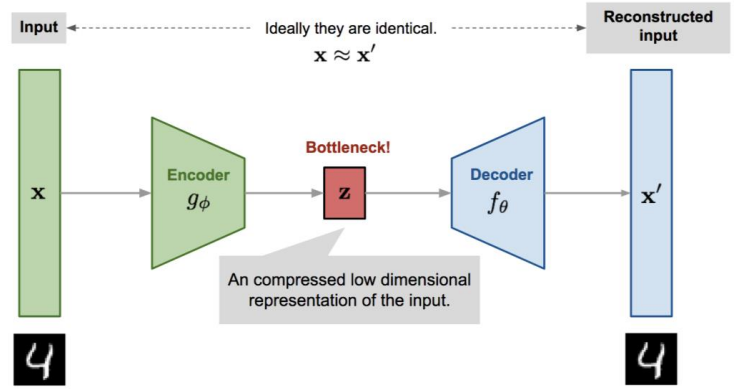
Given an image, the target is the image itself! Very nice.





A simple autoencoder model architecture

Question: design a model architecture that given an image (say, a 224x224 RGB image), implements the autoencoder idea.



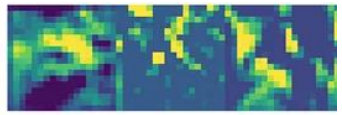
Answer: many possible, but here's one:

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]

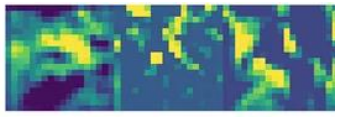
Encoder

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!

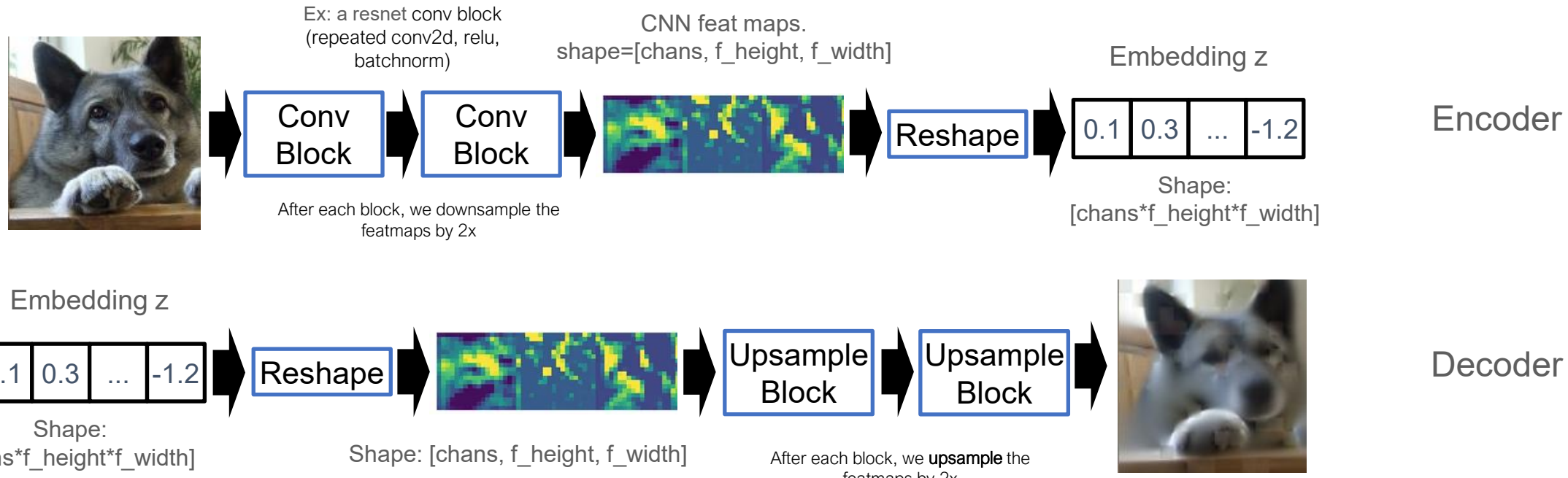
Decoder

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])



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

A simple autoencoder model architecture (pytorch)

```
class Autoencoder(nn.Module):
    def __init__(self):
        super().__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
```

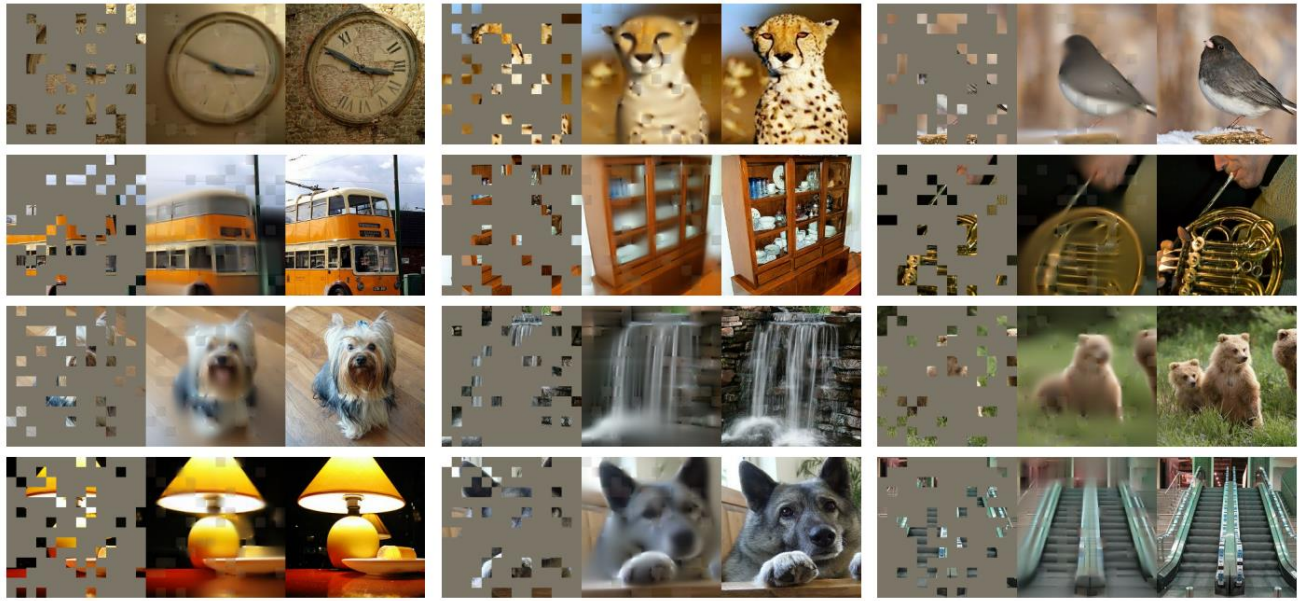
Fun fact: we can also do non-learnable upsampling (eg standard 2d interpolation like nearest-neighbor, linear-interp, etc) in pytorch, and even backprop through them! https://pytorch.org/docs/stable/generated/torch.nn.Upsample.html

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

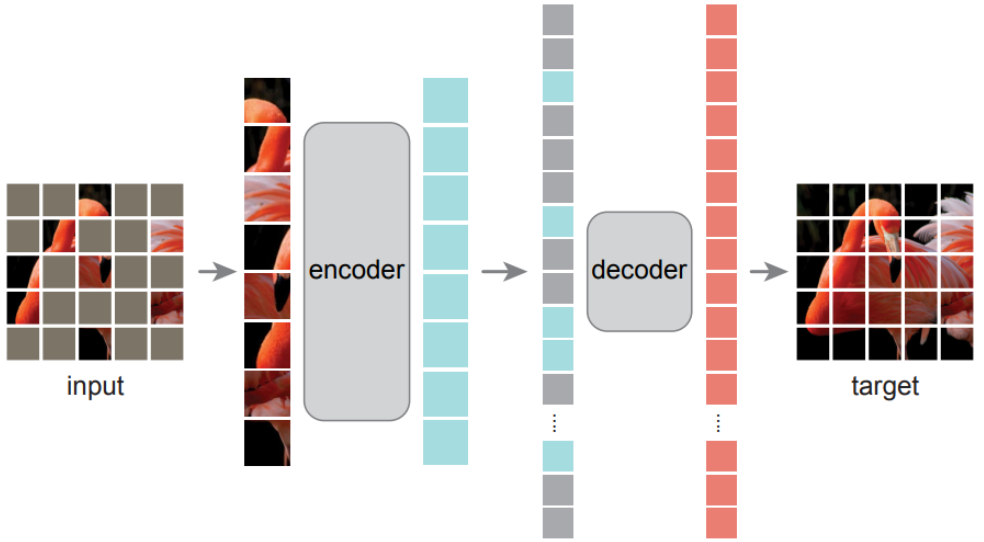


Figure 1. **Our MAE architecture**. During pre-training, a large random subset of image patches (*e.g.*, 75%) is masked out. The encoder is applied to the small subset of *visible patches*. Mask tokens are introduced *after* the encoder, and the full set of encoded patches and mask tokens is processed 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**

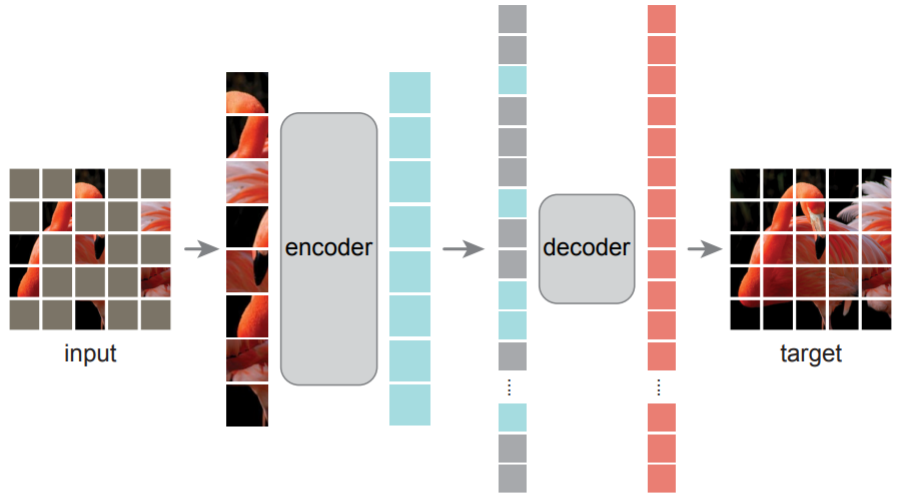


Figure 1. **Our MAE architecture**. During pre-training, a large random subset of image patches (*e.g.*, 75%) is masked out. The encoder is applied to the small subset of *visible patches*. Mask tokens are introduced *after* the encoder, and the full set of encoded patches and mask tokens is processed 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 (ie ImageNet-1k image classification).

- Intuition: rather than starting your model weights from scratch (ie 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

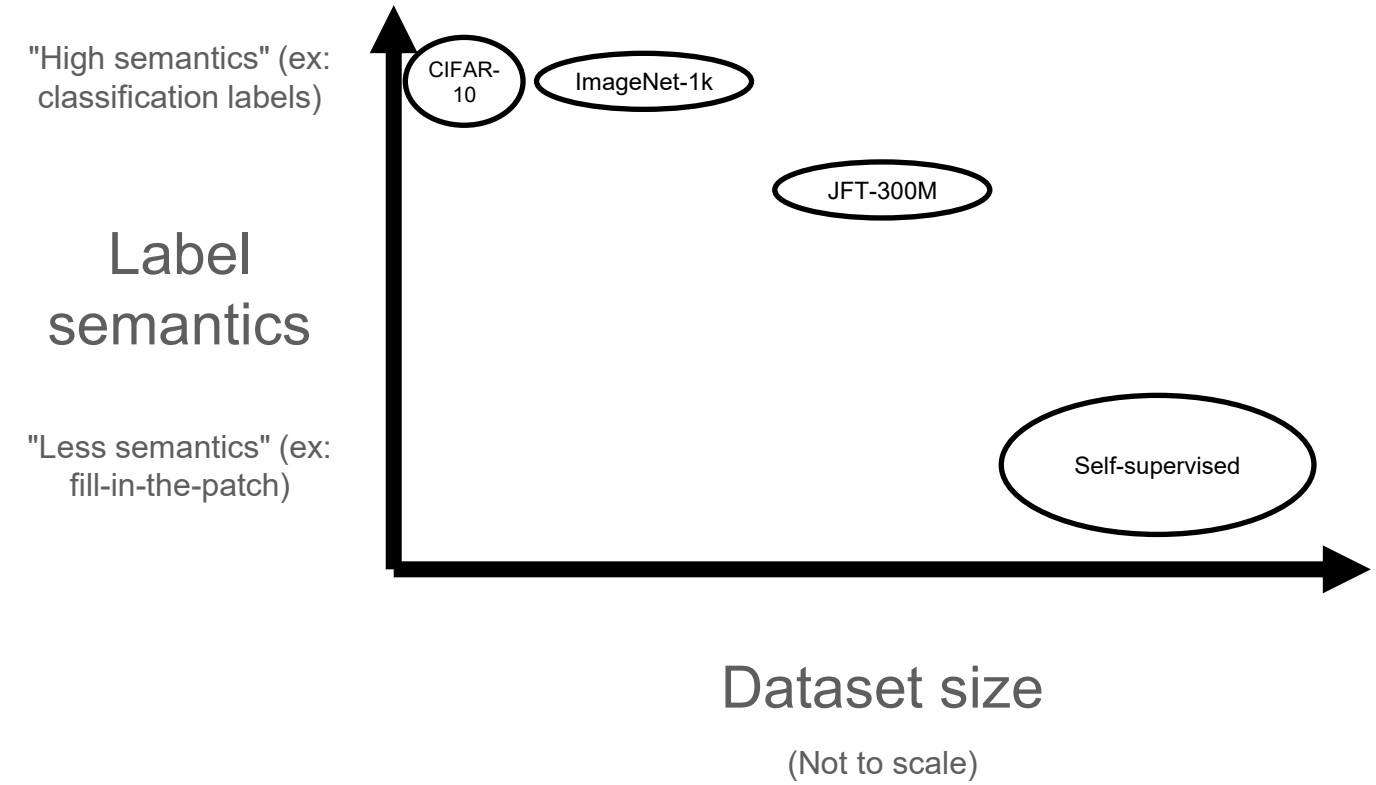
Figure 8: Example of retrieval results using the control (production) model, the ResNeXt-101 Annotations-1.3B model, and the ViT-B/16 Annotations-1.3B model. The ViT model generally matches more similar product results.

Dataset tradeoff: scale vs semantics?

What is more important: semantics, or scale?

Takeaway: for high-capacity deep learning models (eg transformer models), large-scale (possibly noisy) datasets are crucial for training foundation models.

But, for downstream tasks: you still need high-quality labels.

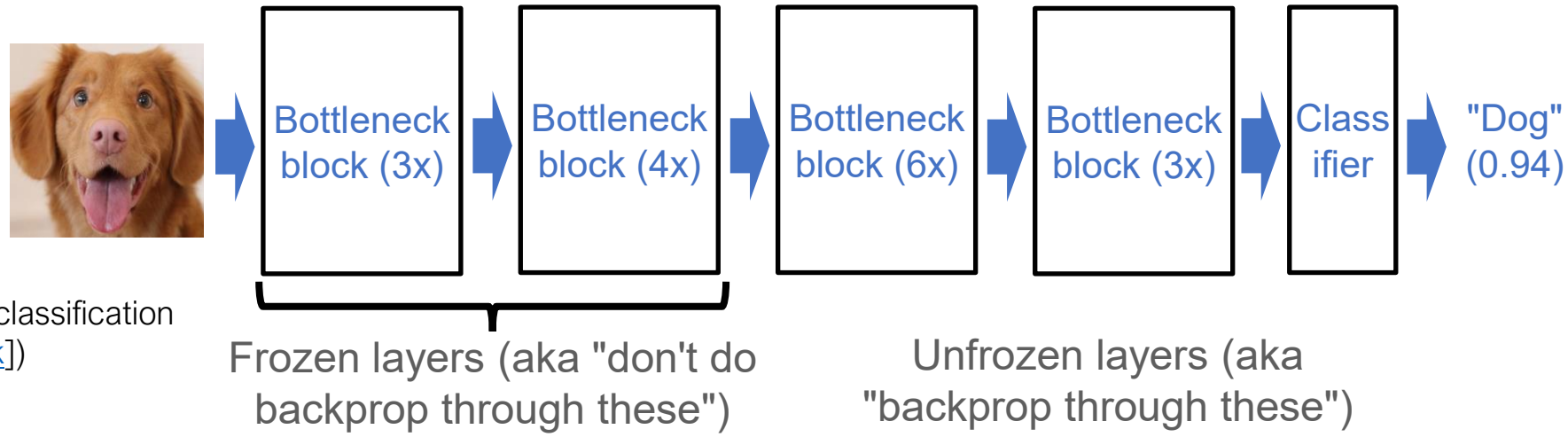


Tangent: model freezing

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)



ResNet50 (Image classification arch [link])

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 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...

```
import torch
import torchvision.models as models

# Load a pre-trained ResNet model
model = models.resnet50(pretrained=True)

# Example to print the names of each layer block
for name, layer in model.named_children():
    print(name)

# Freeze layers up to (and including) `layer2`
for name, layer in model.named_children():
    if name in ['conv1', 'bn1', 'layer1', 'layer2']:
        for param in layer.parameters():
            param.requires_grad = False

# important: only give trainable (non-frozen) params
# to optimizer
trainable_params = [
    p for p in model.parameters() if p.requires_grad
]
optimizer = torch.optim.SGD(
    trainable_params, lr=0.001, momentum=0.9
)
```

https://medium.com/we-talk-data/guide-to-freezing-layers-in-pytorch-best-practices-and-practical-examples-8e644e7a9598

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.

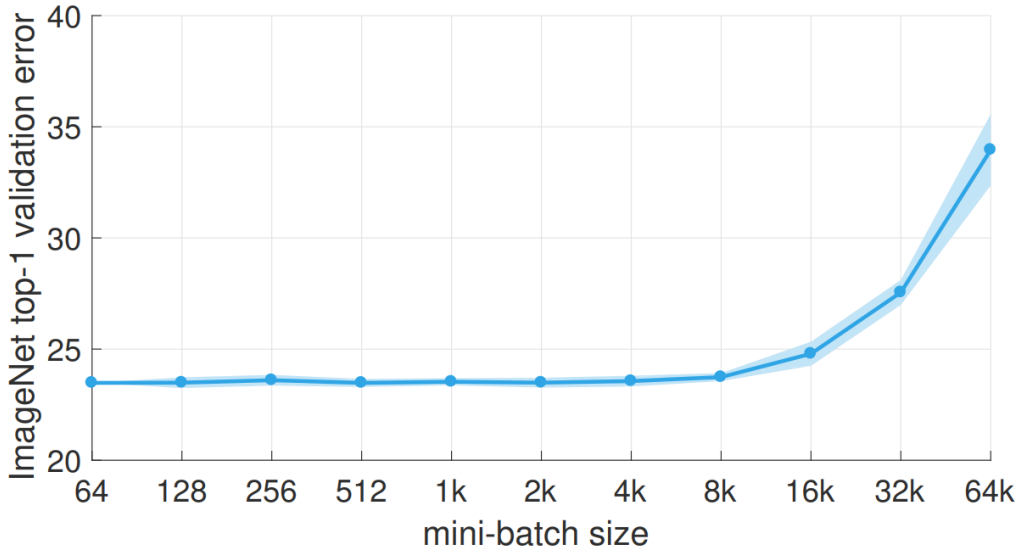


Figure 1. **ImageNet top-1 validation error *vs.* minibatch size.** Error range of plus/minus *two* standard deviations is shown. We present a simple and general technique for scaling distributed synchronous SGD to minibatches of up to 8k images *while maintaining the top-1 error of small minibatch training*.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

Tangent: "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 2026, 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**.

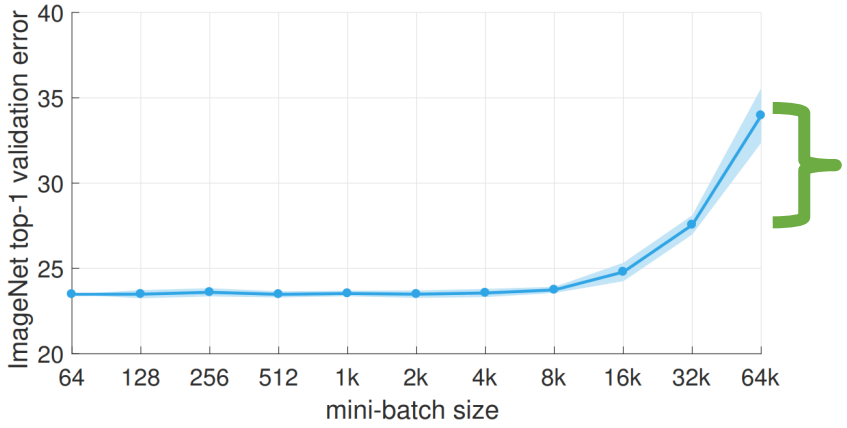


Figure 1. **ImageNet top-1 validation error *vs*. minibatch size.** Error range of plus/minus *two* standard deviations is shown. We present a simple and general technique for scaling distributed synchronous SGD to minibatches of up to 8k images *while maintaining the top-1 error of small minibatch training*.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