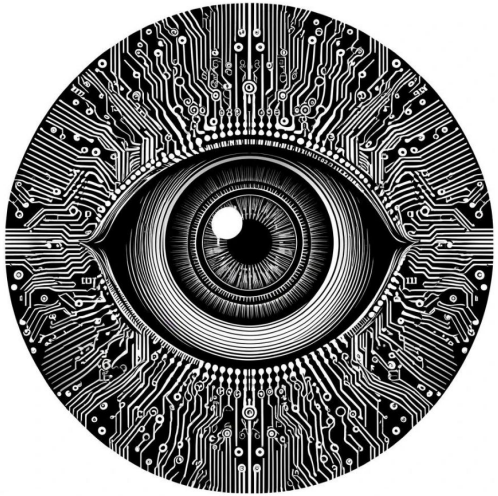
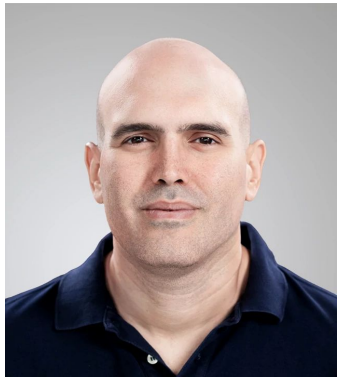
Workshop 10 - Image Embeddings, VIT, and CLIP



Antonio Rueda-Toicen

About me

- AI Researcher at Hasso Plattner Institute, AI Engineer & DevRel for Voxel51
- Organizer of the Berlin Computer Vision Group
- Instructor at Nvidia's Deep Learning Institute and Berlin's Data Science Retreat
- Preparing a MOOC for OpenHPI (now open for registration)



LinkedIn

How to use our Discord channel during the workshop

- Our channel is **#practical-computer-vision-workshops.** Please ask all questions there instead of the Zoom chat. Through Discord we can have better and more detailed discussions.

- **Step 1 - Use the Discord invite on the Voxel51 website** https://discord.com/invite/fiftyone-community
- **Step 2 - Access channel** (use the direct link below or search) http://bit.ly/3YmvPXG

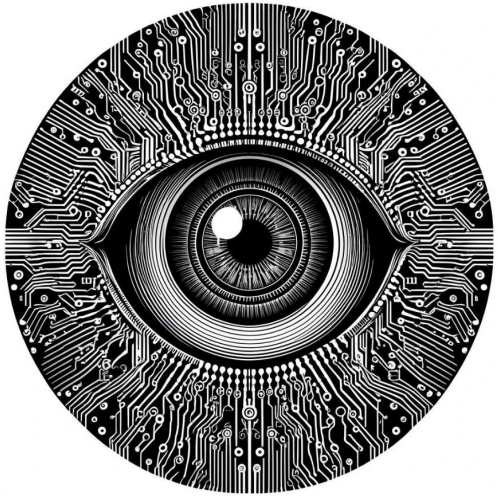
Agenda

- Image Embeddings
- Vision Transformer
- Contrastive Language-Image Pretraining (CLIP)

Notebooks

- Creating Embeddings from ResNet34
- Visualizing and Clustering Embeddings with FiftyOne
- Zero-shot classification with CLIP

Image Embeddings

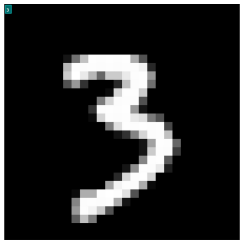
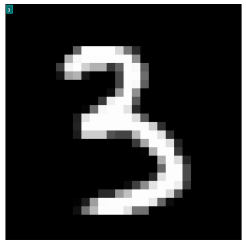


Antonio Rueda-Toicen

Learning goals

- Understand dense vector embeddings as learned representations
- Describe how image embeddings are produced in neural networks
- Extract image embeddings out of a pretrained Resnet50
- Inspect image neighborhoods using the cosine similarity metric

Sparse vs dense vector representations

$\mathbf{x}_1 =$  $\mathbf{x}_2 =$ 

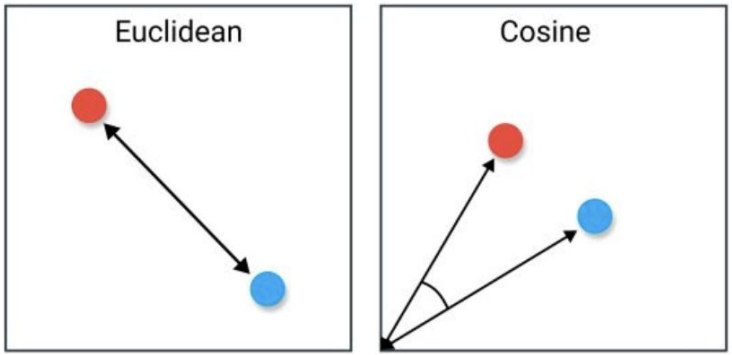
28 x 28 pixels = 784 values

$$\mathbf{x}_1\text{sparse} = \mathbf{x}_2\text{sparse} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^\top$$

$$\mathbf{x}_2\text{dense} = \begin{bmatrix} 0.10 & -0.75 & 0.35 & 0.41 & -0.20 & 0.89 & -0.60 & 0.03 & 0.59 & -0.50 \end{bmatrix}^\top$$

$$\mathbf{x}_1\text{dense} = \begin{bmatrix} 0.12 & -0.85 & 0.37 & 0.44 & -0.22 & 0.90 & -0.63 & 0.01 & 0.58 & -0.49 \end{bmatrix}^\top$$

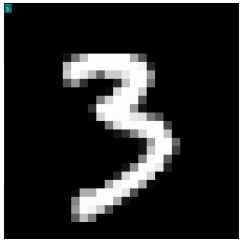
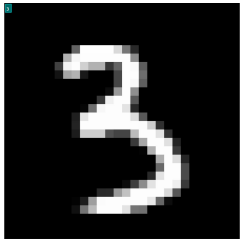
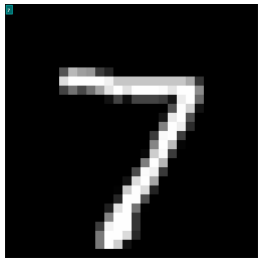
Common metrics to compare embeddings



$$d_{\text{euclid}}(x_1, x_2) = \sqrt{\sum_{i=1}^{n}(x_{1i} - x_{2i})^2}$$

$$\text{cosine similarity}(x_1, x_2) = \frac{x_1 \cdot x_2}{\|x_1\| \, \|x_2\|}$$

Similar objects have similar embeddings

$\mathbf{x}_1 =$  $\mathbf{x}_2 =$  $\mathbf{x}_3 =$ 

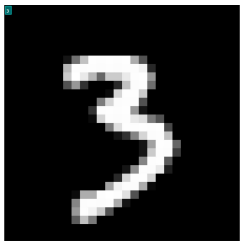
Similar digits (X_1, X_2): High cos similarity (≈ 0.98), Low Euclidean distance (≈ 0.3)

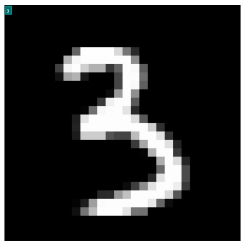
Different digits (X_1, X_3): Low cos similarity (≈ 0.4), High Euclidean distance (≈ 2.1)

Embeddings as representations learned during training

```
import torch.nn as nn


conv_linear_embedder = nn.Sequential(
    nn.Conv2d(1, 32, kernel_size=3),  # 26x26x32
    nn.ReLU(),
    nn.Conv2d(32, 64, kernel_size=3),  # 24x24x64
    nn.ReLU(),
    nn.Flatten(),  # 24*24*64
    # This layer can be extracted as 64-dimensional
    # embedding
    nn.Linear(24*24*64, 64),
    nn.ReLU(),
    # This layer can be a 10-dimensional embedding
    nn.Linear(in_features=64, out_features=10)
)
```

$$H(p,q) = -\sum_i p(i) \log q(i)$$

$\mathbf{x}_1 =$ 

$\mathbf{x}_2 =$ 

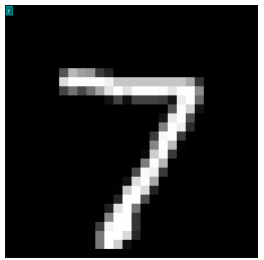
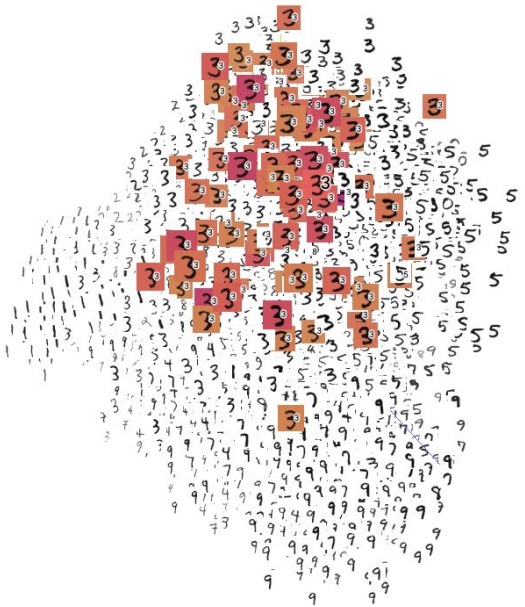
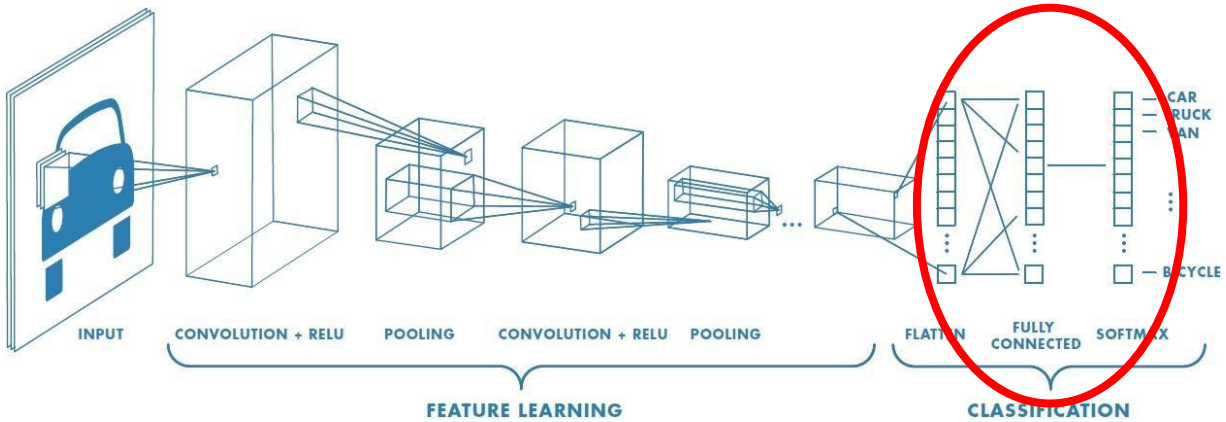
$\mathbf{x}_3 =$ 

Image embeddings as a byproduct of training

When we train a neural network, we make it **embed** inputs that are semantically similar, close to each other



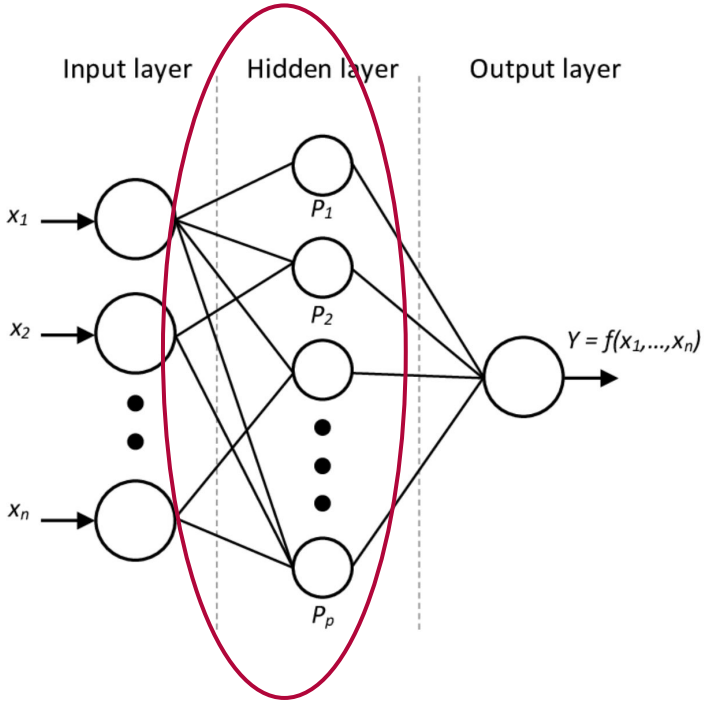


Dense vector embeddings can be produced by flattening, pooling, or feeding (learned) features to linear layers

Source of image (try exploring interactively)

Approaches to produce embeddings with neural networks

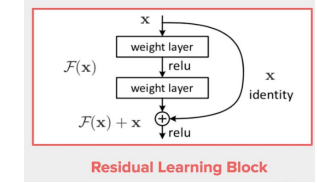
```python
import torch.nn as nn

# Approach 1: Direct embedding
linear_embedder = nn.Sequential(
    nn.Flatten(),  # 784
    nn.Linear(784, 10)
)

# Approach 2: Convolution and then fully connected layer
conv_linear_embedder = nn.Sequential(
    nn.Conv2d(1, 32, kernel_size=3),  # 26x26x32
    nn.ReLU(),
    nn.Conv2d(32, 64, kernel_size=3),  # 24x24x64
    nn.ReLU(),
    nn.Flatten(),  # 24*24*64
    nn.Linear(24*24*64, 10)
)

# Approach 3: Global pooling after convolutions
conv_pool_embedder = nn.Sequential( # MNIST input is size 28x28
    nn.Conv2d(1, 32, kernel_size=3),  # 26x26x32
    nn.ReLU(),
    nn.Conv2d(32, 10, kernel_size=3),  # 24x24x10
    nn.ReLU(),
    nn.AdaptiveAvgPool2d((1, 1)),  # 1x1x10
    nn.Flatten()  # 10
)
```

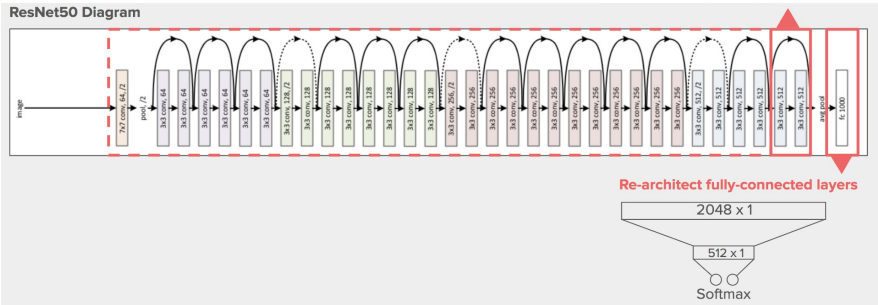


$$L_{L1} = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|$$

Extract embeddings from a pretrained Resnet50

```python
import torch
from torchvision import models

# Load pre-trained ResNet50 weights and transforms
weights = models.ResNet50_Weights.IMAGENET1K_V2
model = models.resnet50(weights=weights)
# Preprocess the image using the transformations from the imagenet dataset
preprocess = weights.transforms()

# Remove final Imagenet classification layer, keep 2048 dense vector
# Take time to unpack this syntax in the practical notebook
embedder = torch.nn.Sequential(*list(model.children())[:-1])

# Disable BatchNormalization
embedder.eval()
# Disable gradient computation
with torch.inference_mode():
    # unsqueeze(0) adds the batch size for inference
    # squeeze() removes the singleton dimensions from the output vector
    embedding = embedder(preprocess(image).unsqueeze(0)).squeeze()
```





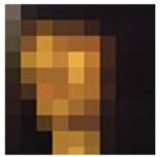
README.md

Art Recommendation system

This is the repository of a portfolio project at DSR. This project aims to identify similar images using pre-trained computer vision networks. For an explanation of the technology see the technology section.

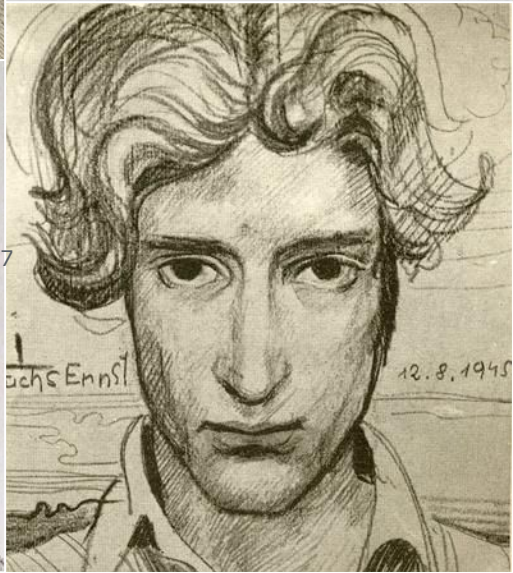
Contributors

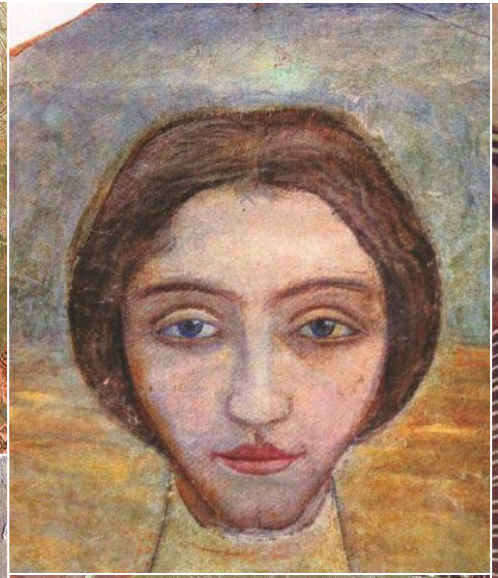
- Catarina Ferreira
- Gargi Maheshwari

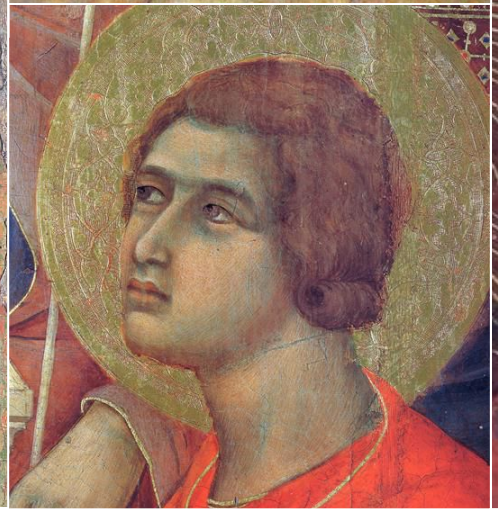


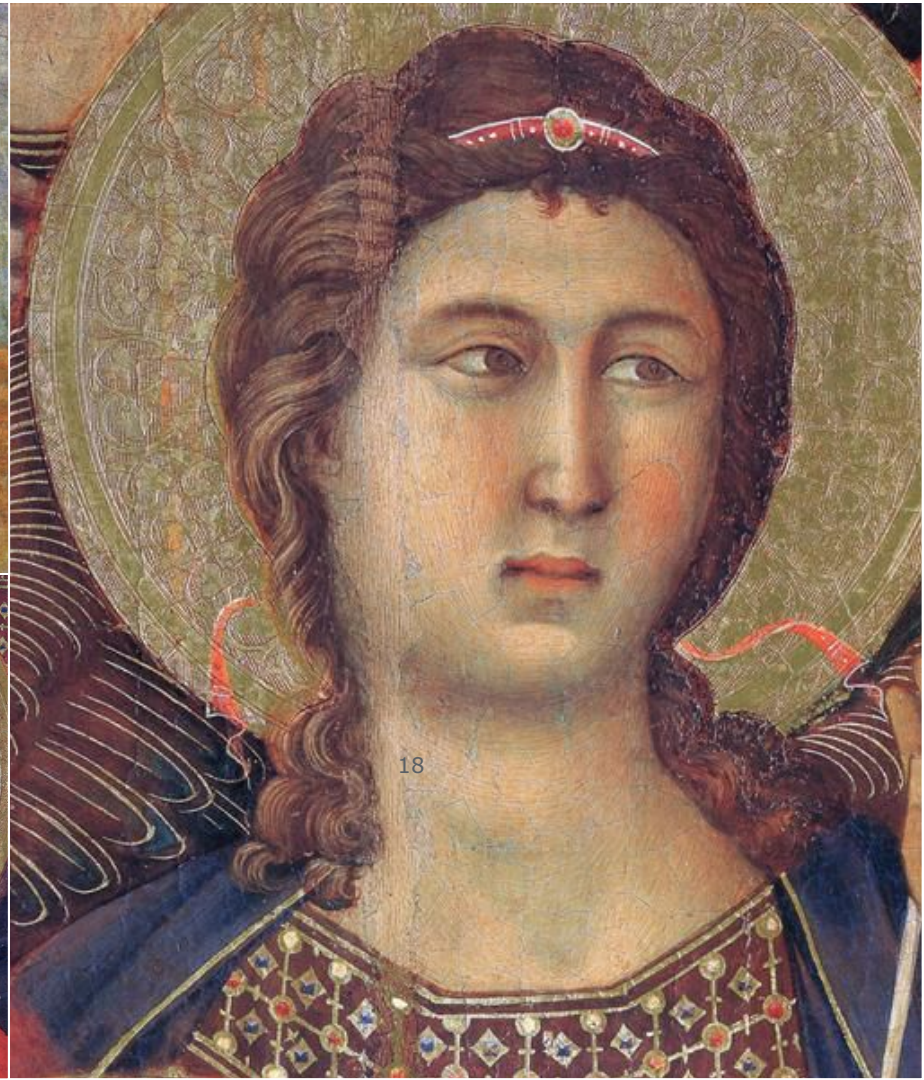
WIKIART
VISUAL ART ENCYCLOPEDIA

https://github.com/gargimaheshwari/Wikiart-similar-art


12.8.1945







Fine-grained embeddings with triplets

```python
import torch import torch.nn.functional as F
def triplet_loss(query, positive,
                 negative, margin=1.0):
    pos_dist = F.pairwise_distance(query, positive)
    neg_dist = F.pairwise_distance(query, negative)
    loss = F.relu(pos_dist - neg_dist + margin)
    return loss.mean()
```

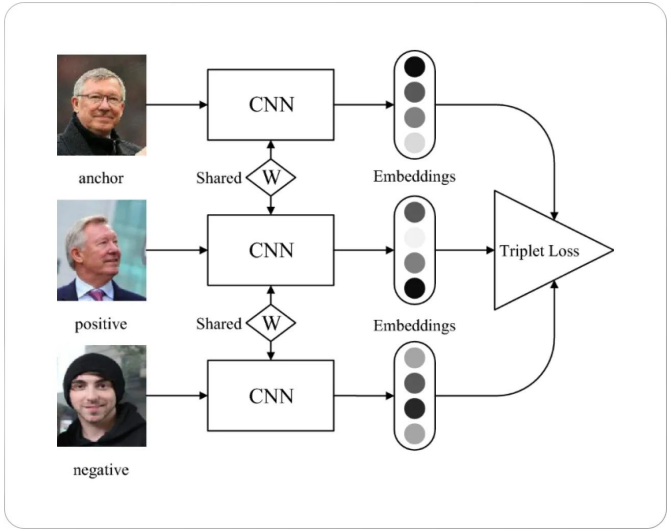


Image from [Triplet Loss: Intro, Implementation, Use Cases]

$$L = \max\left(\|f(a) - f(p)\|_2^2 - \|f(a) - f(n)\|_2^2 + \alpha, 0\right)$$

Application: differential diagnosis

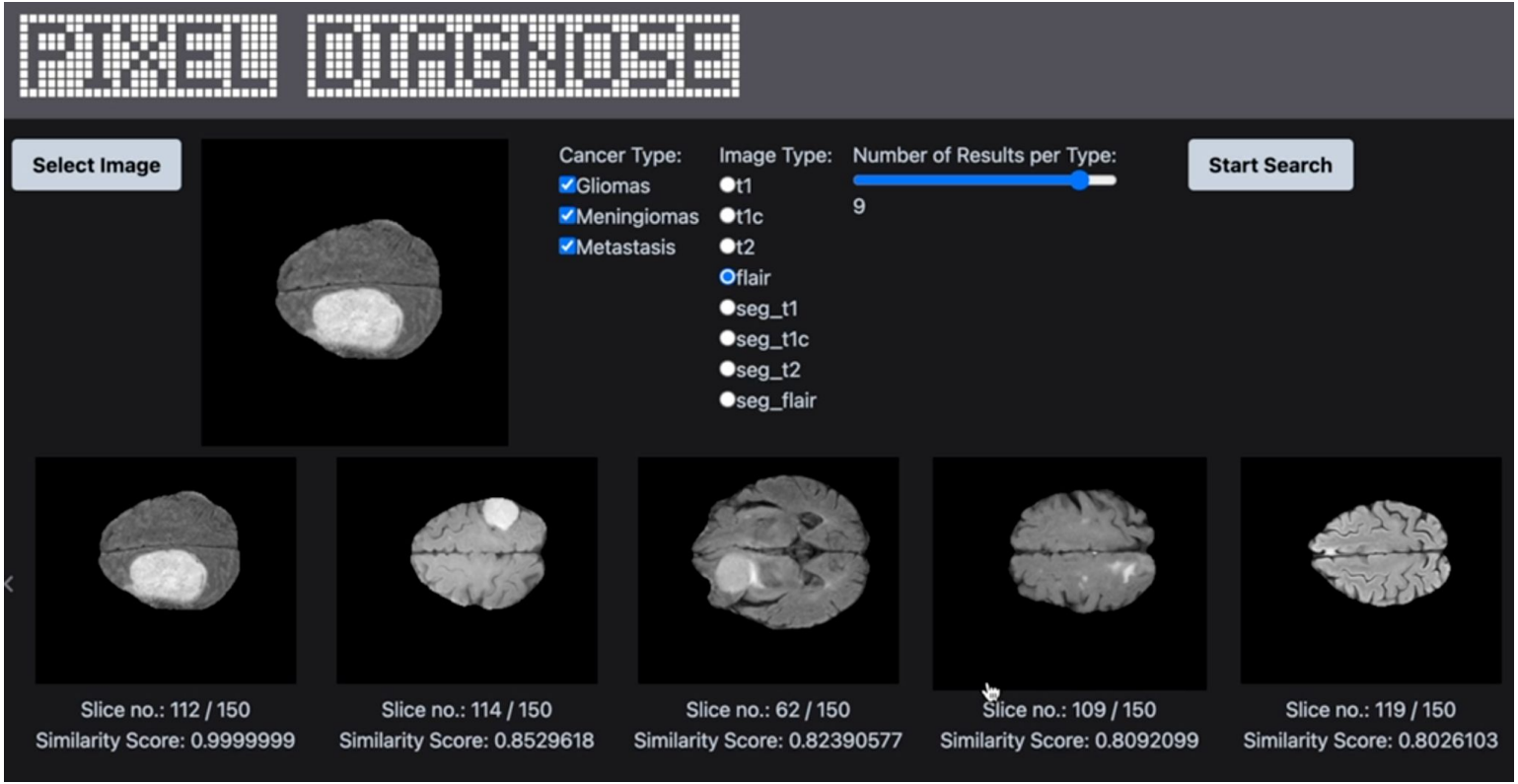
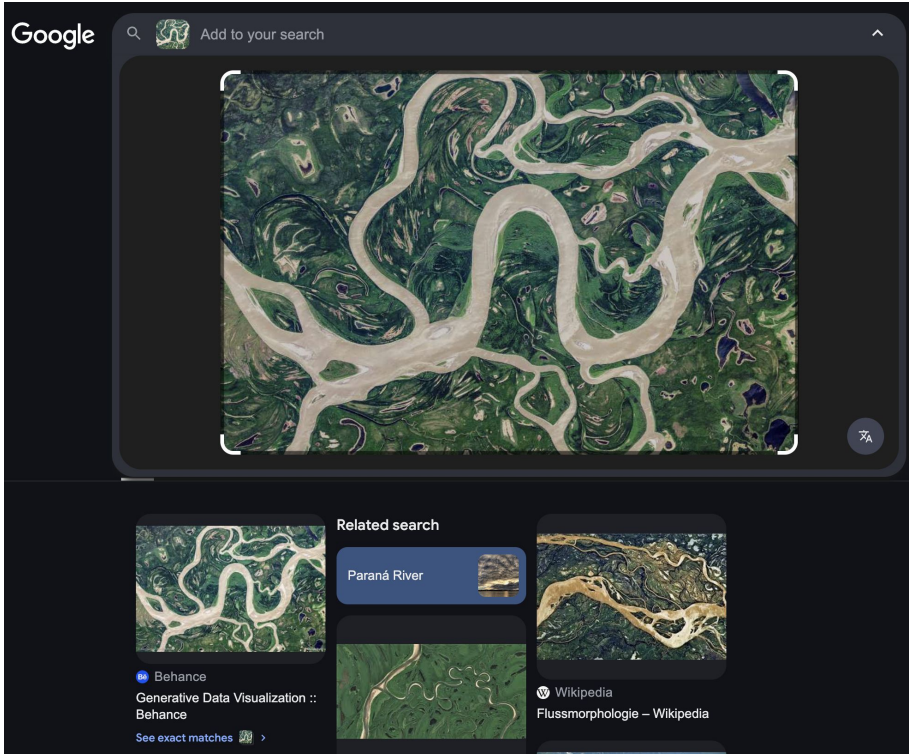


Image from https://pixel-diagnose.github.io/

Application: search by image



Search for similar images in images.google.com

Summary

Embeddings are learned vector representations

- Neural networks learn to map similar images to similar vectors through training
- We measure the similarity of embeddings using metrics like cosine similarity and Euclidean distance

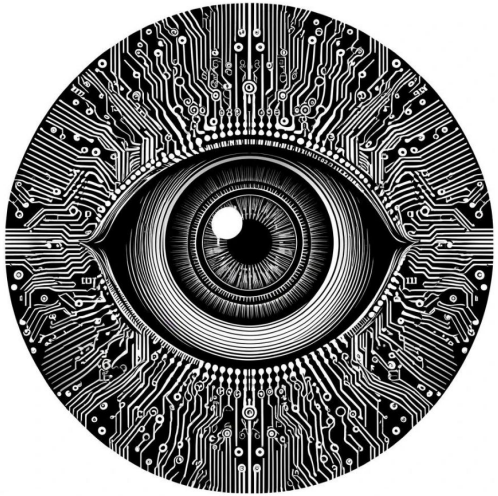
There are different architectural approaches to create embeddings

- We can use linear layers, or flattened and pooled convolutions to produce embeddings
- Imagenet-pretrained models produce rich representations in their embeddings
- Embeddings can also be fine tuned through the triplet loss

Embeddings have multiple applications in semantic search and dataset curation

- They can be used for reverse image search, recommendation systems, and deduplication

Vision Transformers



Antonio Rueda-Toicen

Learning goals

- Gain an overview of the Vision Transformer (ViT) architecture and the self-attention mechanism
- Understand tradeoffs between vision transformers and convolutional networks

The Vision Transformer (ViT) architecture

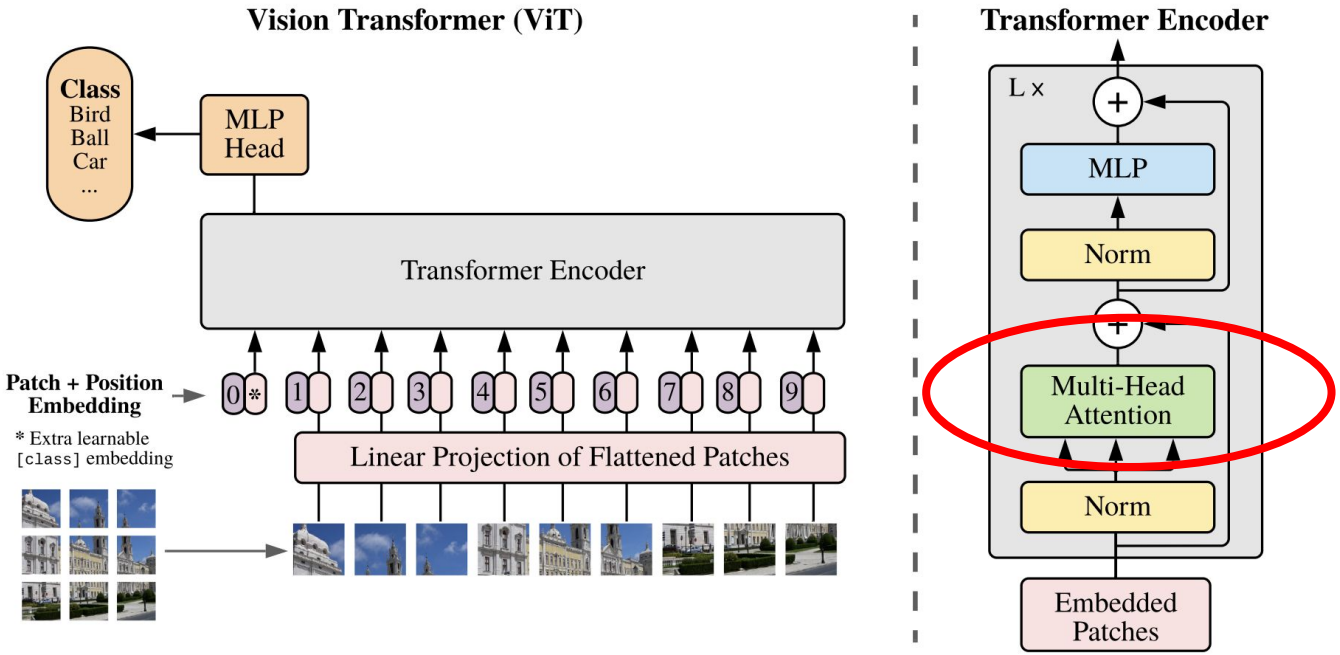


Image from [An Image is Worth 16x16 Words - Transformers for Image Recognition at Scale]

The Vision Transformer (ViT) architecture

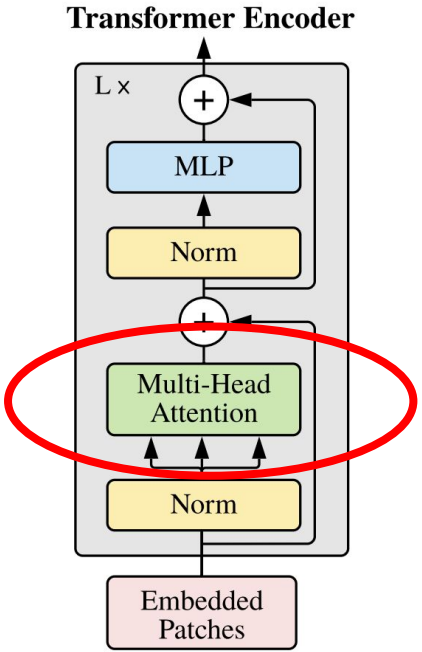
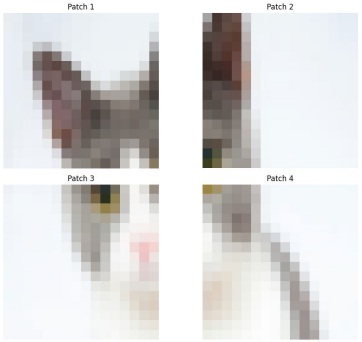


Image from [Transformers for Image Recognition at Scale]

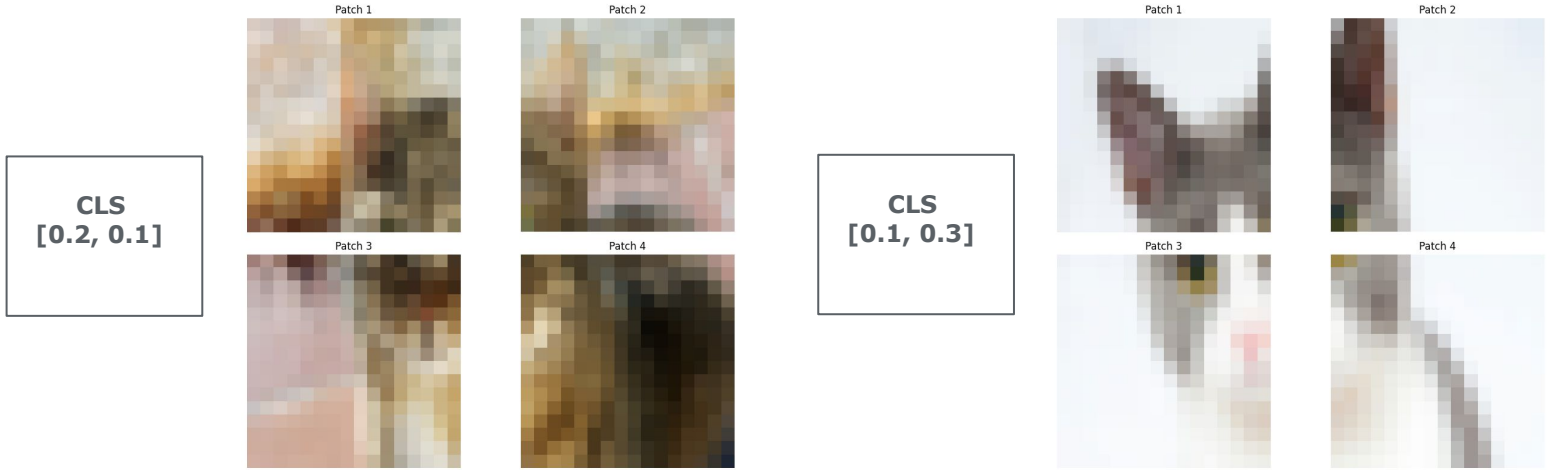
Projection of flattened patches and adding positional embeddings

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

# Example CIFAR-10-like data: (batch_size, 3, 32, 32)
data = torch.randn(4, 3, 32, 32)

# Hyperparameters
patch_size = 16
channels = 3
embed_dim = 768  # Standard ViT-Base embedding dimension

# Calculate number of patches
num_patches = (32 // patch_size) * (32 // patch_size)  # 4 = 2*2
patch_dim = patch_size * patch_size * channels  # 768 = 16*16*3

# Linear projection and positional embedding
patch_embed = nn.Linear(patch_dim, embed_dim)
pos_embed = nn.Parameter(torch.zeros(1, num_patches, embed_dim))

# Reshape data into patches
# From (batch_size, channels, height, width) to (batch_size, channels, h_patches, patch_size, w_patches, patch_size)
patches = data.unfold(2, patch_size, patch_size).unfold(3, patch_size, patch_size)

# Reshape to (batch_size, num_patches, patch_dim)
patches = patches.permute(0, 2, 4, 1, 3, 5).reshape(data.size(0), num_patches, -1)

# Linear projection of patches
x = patch_embed(patches)  # Shape: (batch_size, num_patches, embed_dim)

# Add positional embeddings
x = x + pos_embed

print(f"Input shape: {data.shape}")  # [4, 3, 32, 32]
print(f"Patches shape: {patches.shape}")  # [4, 4, 768]
print(f"Output shape: {x.shape}")  # [4, 4, 768]
```



Notice that every pixel of the original image has its own positional embedding value

The CLS patch / token



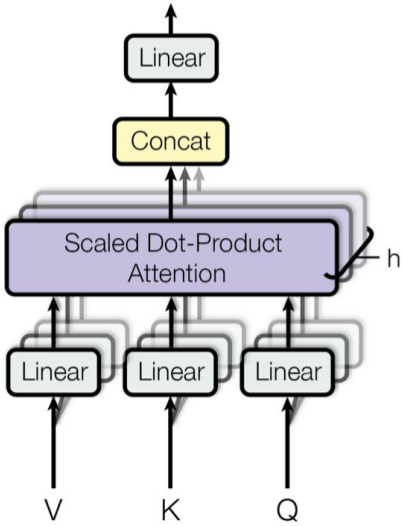

```
cls_token = nn.Parameter(torch.randn(1, 1, embed_dim) * 0.02)
```

Similar class images end up with similar CLS embeddings only after training

The Vision Transformer (ViT) architecture

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = [\text{head}_1, \ldots, \text{head}_h]\mathbf{W}_0$$

$$\text{where head}_i = \text{Attention}\left(\mathbf{Q}\mathbf{W}_i^Q, \mathbf{K}\mathbf{W}_i^K, \mathbf{V}\mathbf{W}_i^V\right)$$



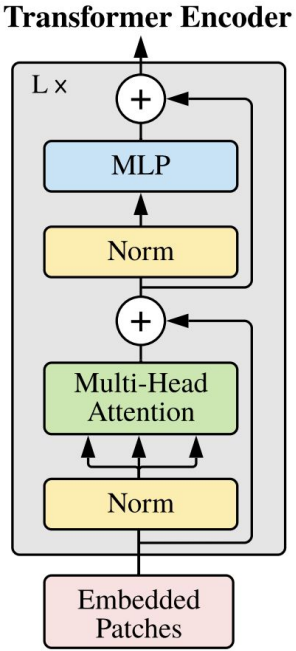
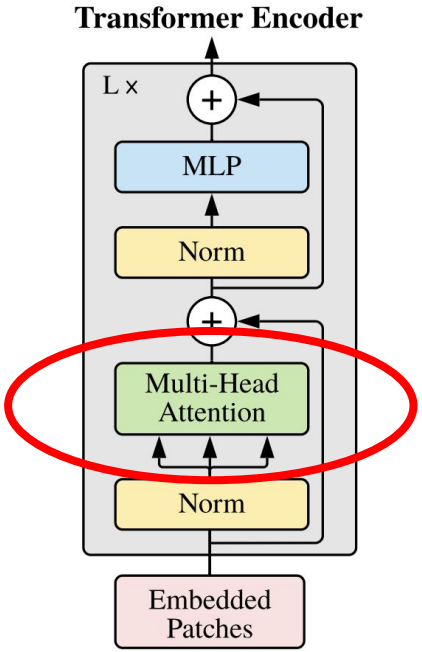


Image from [Multi-Head Attention Explained | Papers With Code]

The attention component

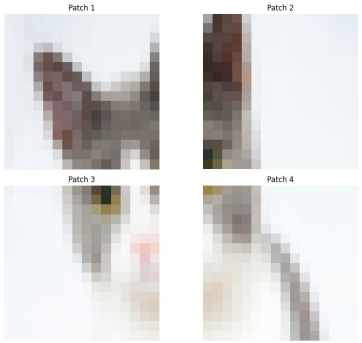
$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$



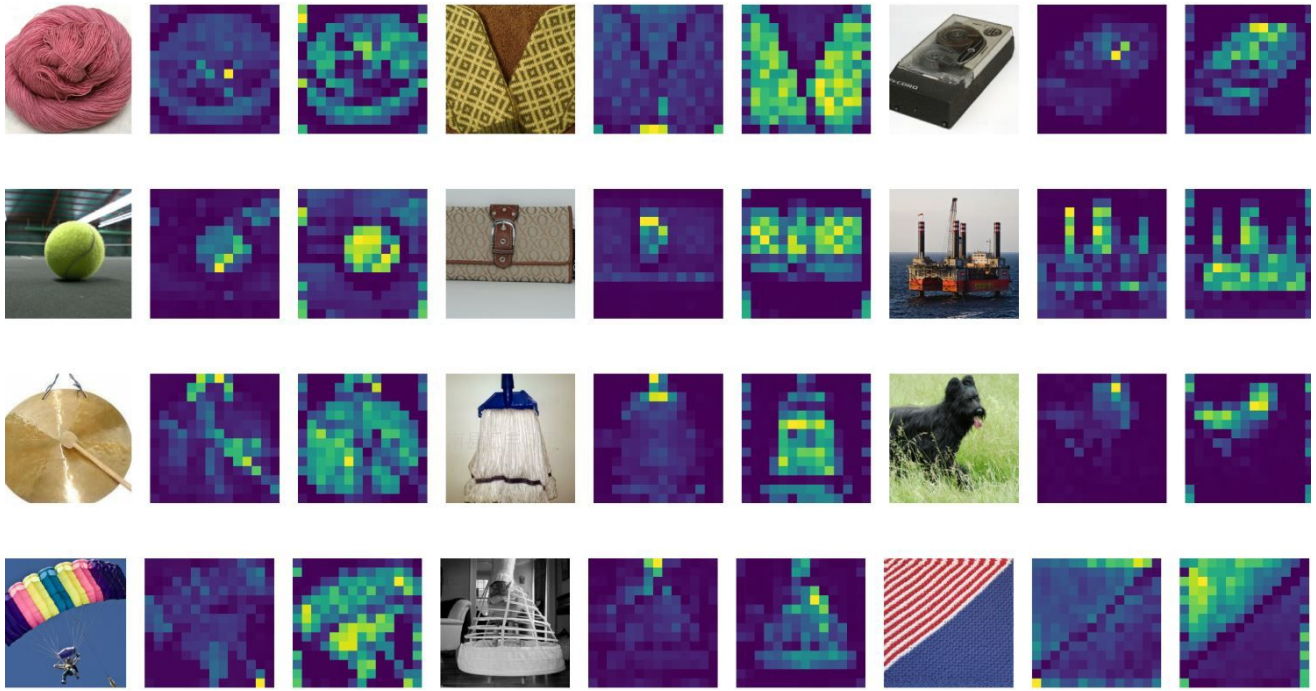
Attention mechanism simplified

```
def attention_mechanism(patch):
    # Transform patch into query, key, value representations
    query = linear_layer(patch)  # "What am I looking for?"
    key = linear_layer(patch)    # "What do I contain?"
    value = linear_layer(patch)  # "Actual information"

    # Calculate attention scores
    attention_scores = softmax((query @ key.transpose()) / sqrt(d_k))

    # Get weighted sum of values
    output = attention_scores @ value
```



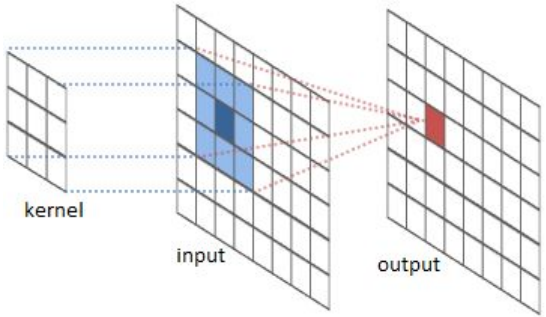
$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

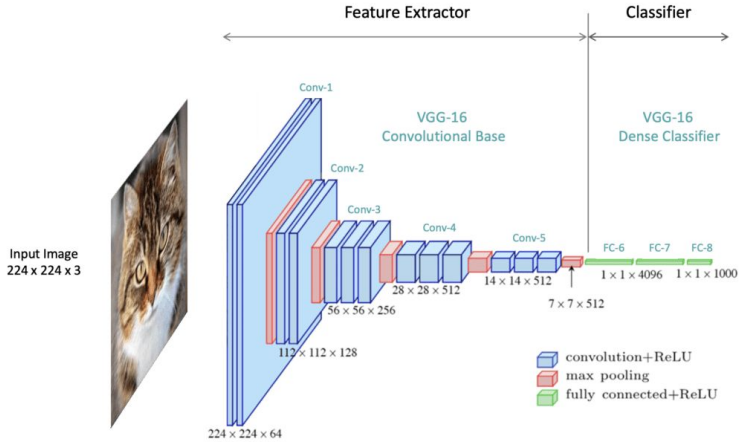
Interpretability of attention maps



Raw images (left) with attention maps of the ViT-S/16 model (right) Source

The locality bias of convolutional networks

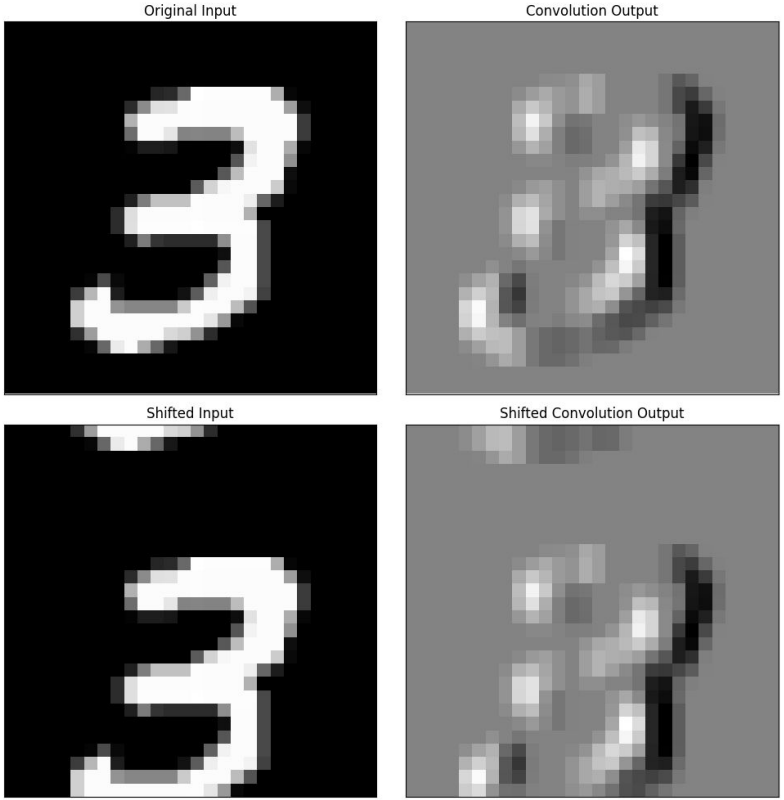




Convolutional networks have the inductive biases of the convolution: pixels in a neighborhood activate together (**locality bias**). The whole image receives the same set of weights in a convolution which creates

We get a “global view” of the image the deeper we go into a convolutional network (aggregating on aggregations) after pooling or strided convolutions.

The ‘translational equivariance’ bias of convolutional networks



$$W = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

If the inputs gets shifted **n** pixels, so does the output.

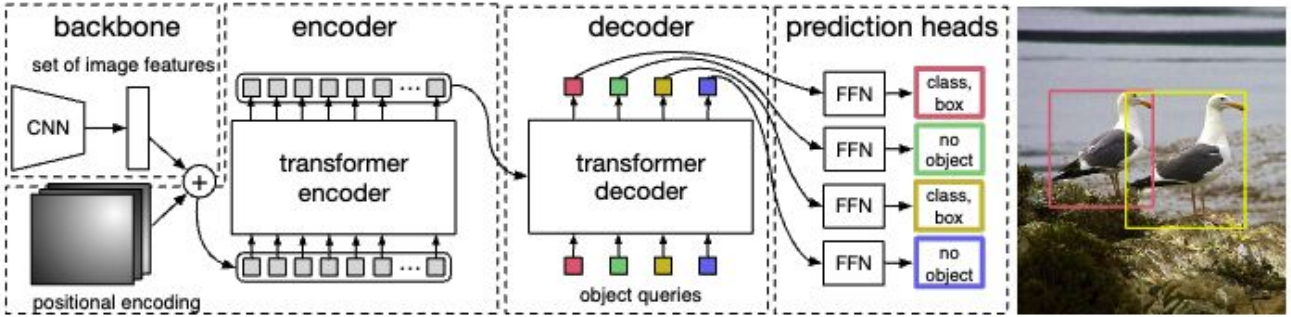
Weights remain the same while capturing this desirable effect

Tradeoffs of ViT with convolutional networks (CNNs)

In general transformers use more parameters than CNNs and require:

- More computing power to train and run inference (quadratic time per patch due to self-attention)
- More data to train (lack the translational equivariance of convolution), ViT was trained on a Google-internal dataset containing **300 million labeled images**
- DeIT is a vision transformer variant that relies on distillation and data augmentation and compute to reduce the size of the training data corpus

In practice, many architectures (like DETR) **combine convolutions with transformers**



Overview of the DETR architecture (source)

Summary

The Vision Transformer (ViT) architecture

- ViT splits images into patches and process them through self-attention
- The attention mechanism computes relationships between all image regions
- Position embeddings maintain spatial information of the patches
- The attention mechanism uses queries, keys, and values to compute weighted relationships between patches
- Multiple attention heads capture different features

Tradeoffs with convolutional neural networks (CNNs)

- ViTs excel at capturing global relationships
- ViTs require more compute and data than CNNs
- CNNs offer built-in translation equivariance
- Hybrid architectures like DETR combine the benefits of both approaches

Further reading and references

An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale

- https://arxiv.org/abs/2010.11929

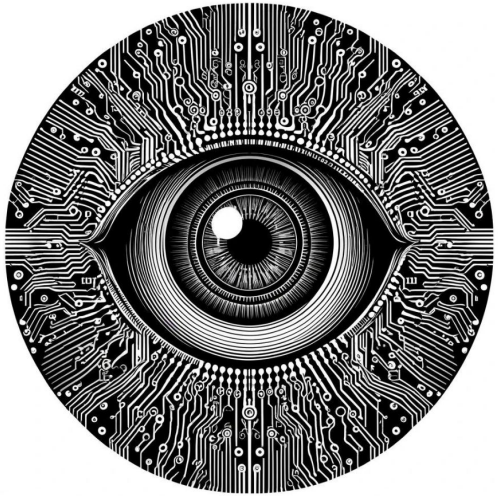
End-to-end object detection with transformers

- https://arxiv.org/abs/2005.12872

Training data-efficient image transformers & distillation through attention

- https://arxiv.org/abs/2012.12877

Contrastive Language-Image Pretraining (CLIP)



Antonio Rueda-Toicen

Learning goals

- Understand how CLIP models are trained
- Classify images with CLIP using a zero-shot approach
- Create image and text embeddings with a pretrained CLIP model
- Discuss CLIP’s uses and limitations

CLIP: ‘Contrastive Language Image Pretraining’

Learning Transferable Visual Models From Natural Language Supervision

Alec Radford [*1] **Jong Wook Kim** [*1] **Chris Hallacy** [1] **Aditya Ramesh** [1] **Gabriel Goh** [1] **Sandhini Agarwal** [1]
Girish Sastry [1] **Amanda Askell** [1] **Pamela Mishkin** [1] **Jack Clark** [1] **Gretchen Krueger** [1] **Ilya Sutskever** [1]

- Connects text and images in a shared embedding space
- Created by OpenAI in 2021
- Trained on **400M image-textual description pairs** scraped from the internet
- Predicts the most relevant text snippet given an image, enabling “zero-shot classification”

Aligning text and image embeddings

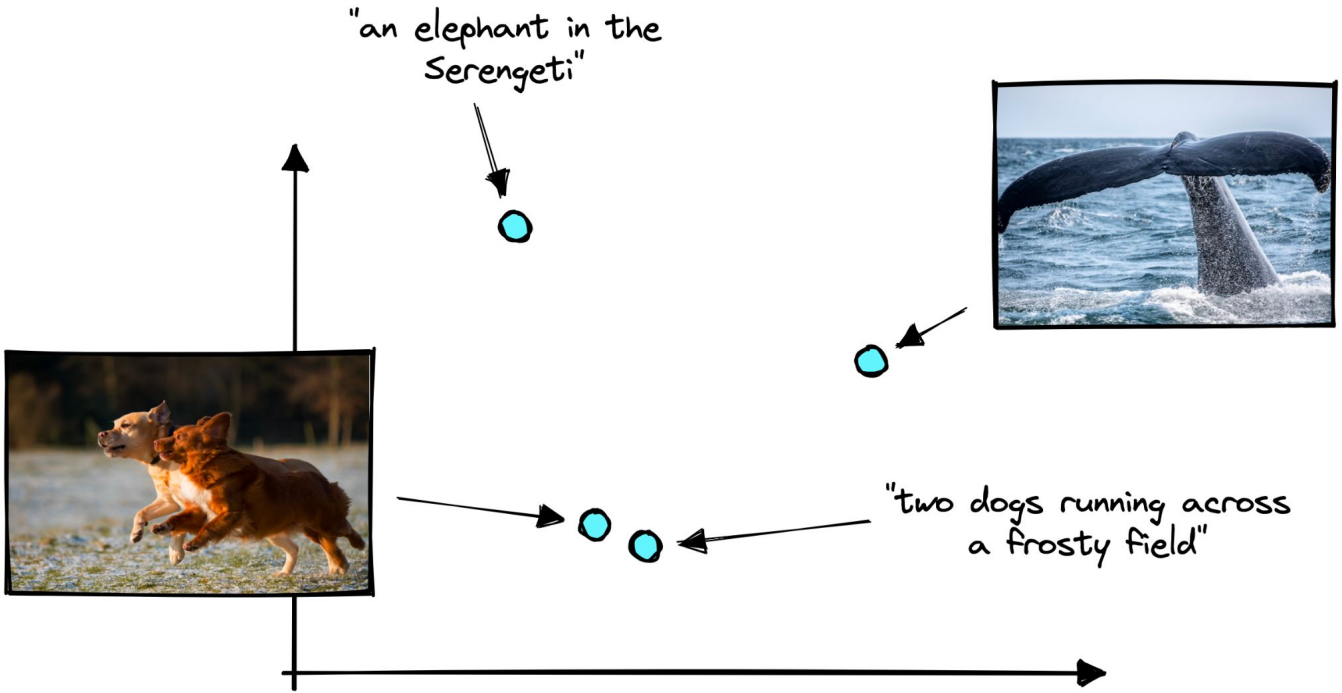
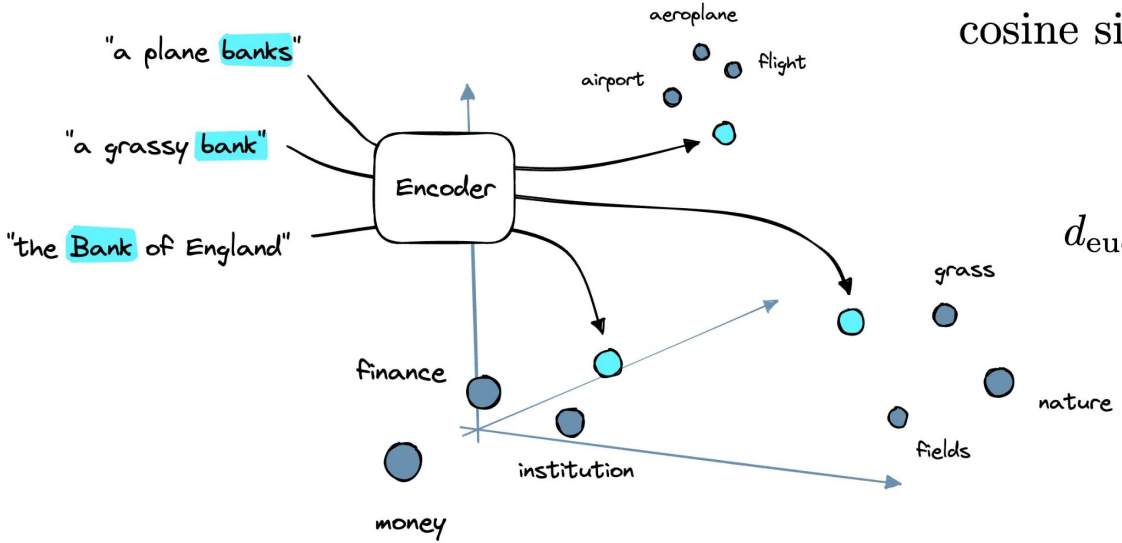


Image from https://www.pinecone.io/learn/series/image-search/clip/

Text encoders

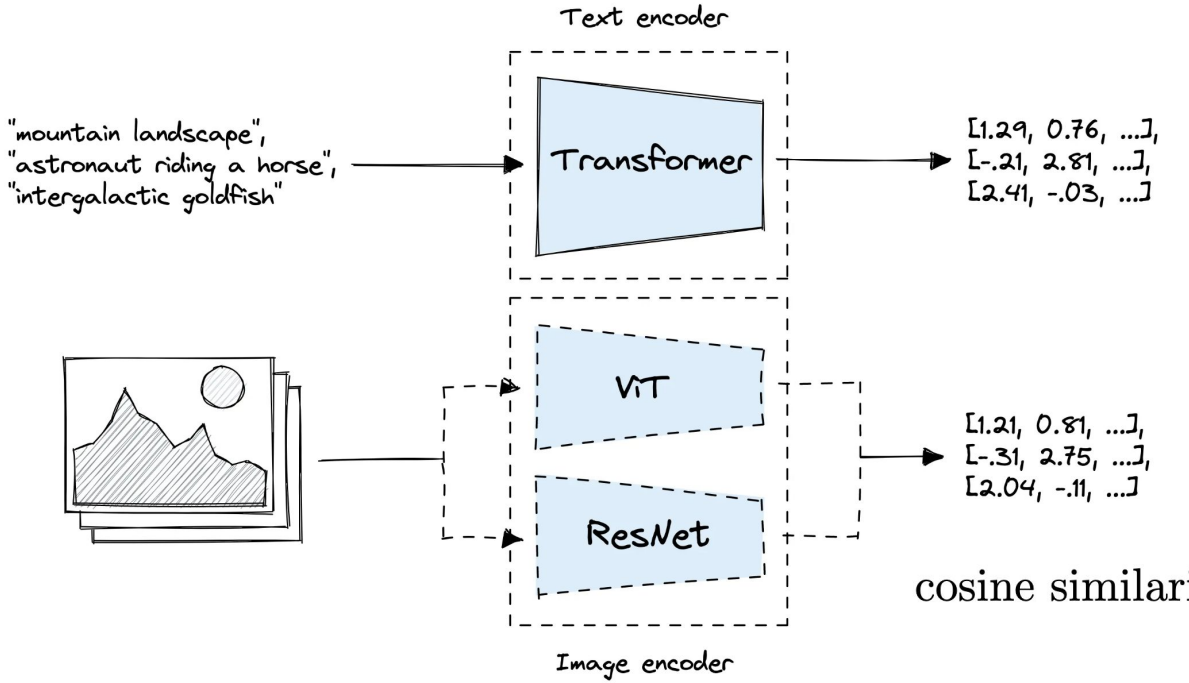


$$\text{cosine similarity}(x_1, x_2) = \frac{x_1 \cdot x_2}{\|x_1\| \, \|x_2\|}$$

$$d_{\text{euclid}}(x_1, x_2) = \sqrt{\sum_{i=1}^{n} (x_{1i} - x_{2i})^2}$$

Image from https://www.pinecone.io/learn/series/image-search/vision-transformers/

CLIP's architecture



Both encoders map inputs to 512 dimensional embeddings

$$\text{cosine similarity}(x_1, x_2) = \frac{x_1 \cdot x_2}{\|x_1\| \ \|x_2\|}$$

Image from https://www.pinecone.io/learn/series/image-search/clip/

Maximizing cosine similarity of matching text and image embeddings

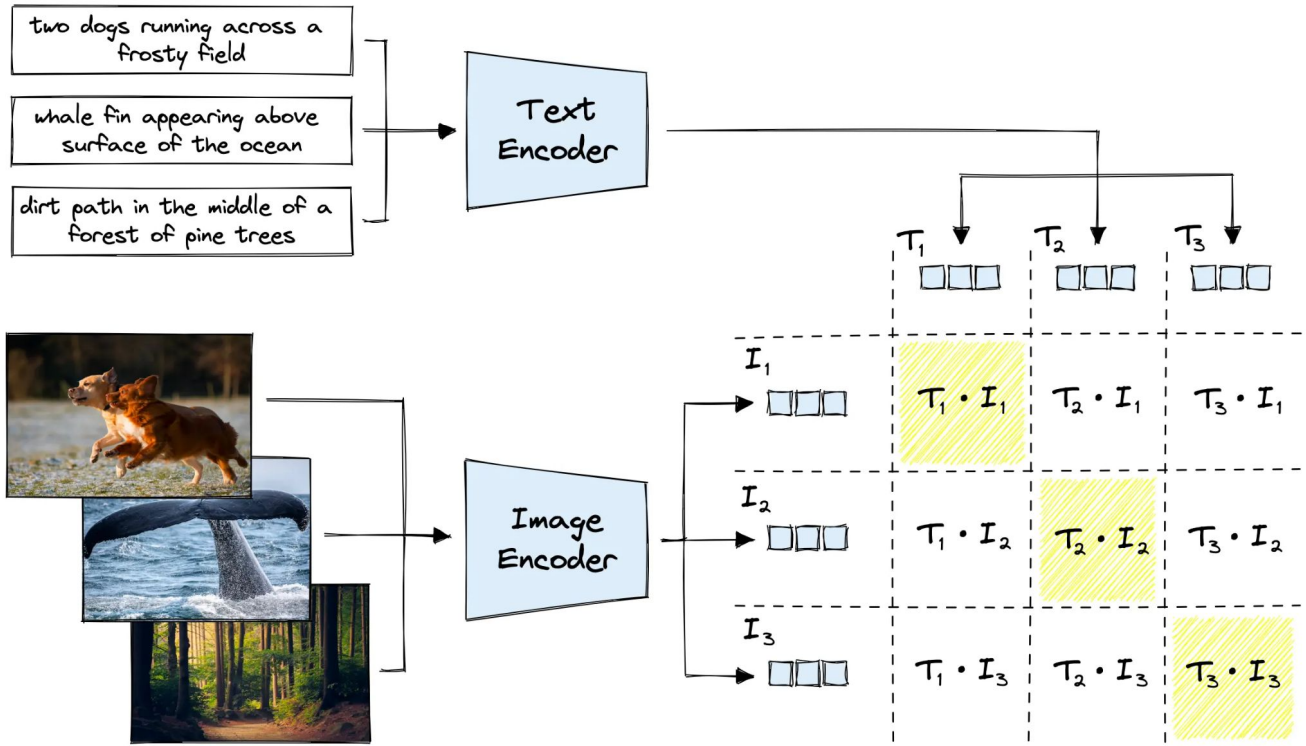
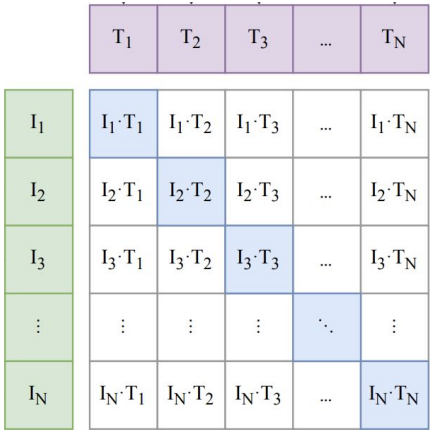


Image from https://www.pinecone.io/learn/series/image-search/clip/

Training algorithm

```
# image_encoder - ResNet or Vision Transformer
# text_encoder  - CBOW or Text Transformer
# I[n, h, w, c] - minibatch of aligned images
# T[n, l]       - minibatch of aligned texts
# W_i[d_i, d_e] - learned proj of image to embed
# W_t[d_t, d_e] - learned proj of text to embed
# t             - learned temperature parameter

# extract feature representations of each modality
I_f = image_encoder(I) #[n, d_i]
T_f = text_encoder(T)  #[n, d_t]

# joint multimodal embedding [n, d_e]
I_e = l2_normalize(np.dot(I_f, W_i), axis=1)
T_e = l2_normalize(np.dot(T_f, W_t), axis=1)

# scaled pairwise cosine similarities [n, n]
logits = np.dot(I_e, T_e.T) * np.exp(t)

# symmetric loss function
labels = np.arange(n)
loss_i = cross_entropy_loss(logits, labels, axis=0)
loss_t = cross_entropy_loss(logits, labels, axis=1)
loss   = (loss_i + loss_t)/2
```

Figure 3. Numpy-like pseudocode for the core of an implementation of CLIP.

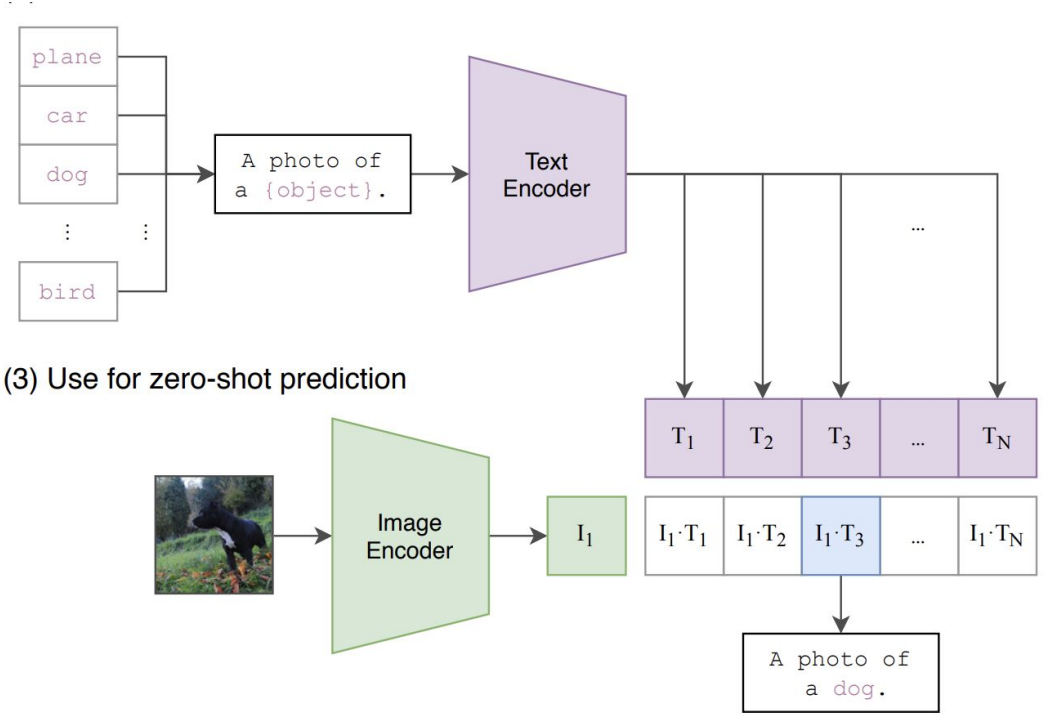


$$H(p,q) = -\sum_i p(i) \log q(i)$$

$$H(q,p) = -\sum_i q(i) \log p(i)$$

$$\text{symmetric CE loss} = \frac{H(p,q) + H(q,p)}{2}$$

Zero-shot classification with CLIP



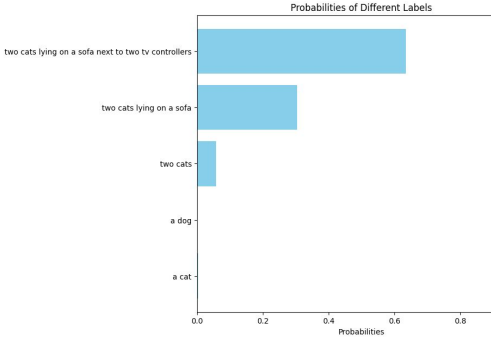
The model is able to create good classifiers without extra training

$$\text{softmax}(x_i, T) = \frac{e^{x_i/T}}{\sum_{j=1}^{n} e^{x_j/T}}$$

Image from Learning Transferable Visual Models From Natural Language Supervision

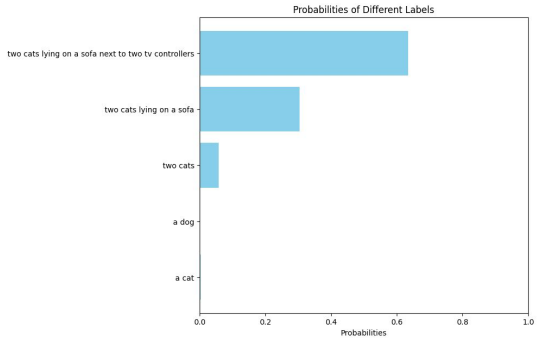
Producing embeddings with CLIP (½)

```
import torch
from transformers import CLIPProcessor, CLIPModel
from PIL import Image

# Load model and inputs
model =
CLIPProcessor.from_pretrained("openai/clip-vit-base-patch32")
processor =
CLIPModel.from_pretrained("openai/clip-vit-base-patch32")
image = Image.open("cats.jpg")
cat_text = ['a cat', 'a dog', 'two cats',
            'two cats lying on a sofa',
             """two cats lying on a sofa
             next to two tv controllers"""]
```



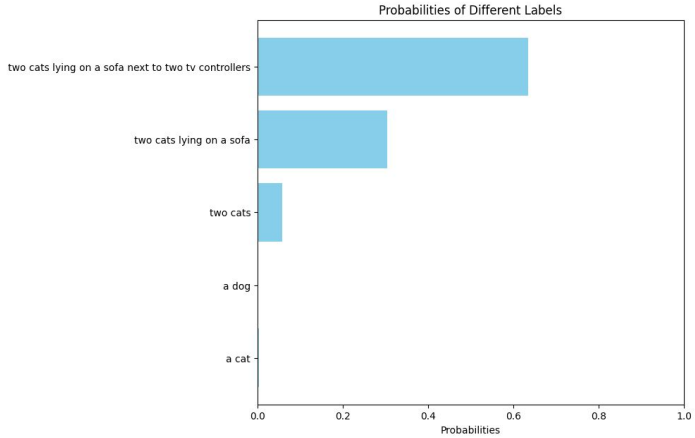
Producing embeddings with CLIP (2/2)

```python
# Get embeddings
inputs = processor(text=cat_text, images=image, return_tensors="pt",
padding=True)
outputs = model(**inputs)

# Calculate similarities
sims = torch.nn.functional.cosine_similarity(
    outputs.image_embeds[:, None],
    outputs.text_embeds[None, :],
    dim=-1
)

# Print results
for text, sim in zip(cat_text, sims[0]):
    print(f"{text}: {sim:.3f}")
```



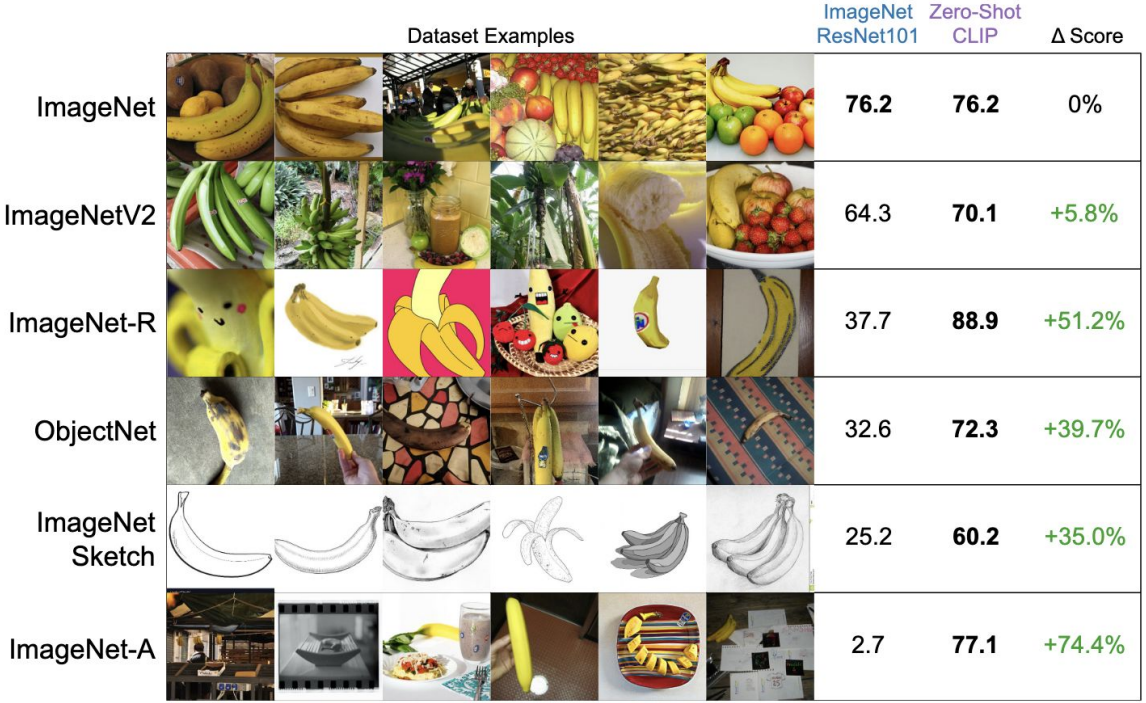
Zero-shot classification with CLIP

```
# Notice what happens with the output probabilities
# when we make the labels more specific
cat_text = ['a cat',
            'a dog',
            'two cats',
           ' two cats lying on a sofa',
            'two cats lying on a sofa next to two tv controllers'
            ]

inputs = processor(text=cat_text,
                   images=[cats_img],
                   return_tensors="pt", padding=True)
cat_outputs = model(**inputs)
cat_logits_per_image = cat_outputs.logits_per_image
cat_probs = (cat_logits_per_image/temperature).softmax(dim=1)
```



$$\text{softmax}(x_i, T) = \frac{e^{x_i/T}}{\sum_{j=1}^{n} e^{x_j/T}}$$

Transferable representations: CLIP against a ResNet101 pretrained on Imagenet



Training Resnet101: about 90 V100 GPU hours (4 days)
Training CLIP ViT-L/14: about ~16,000 V100 GPU hours (666 days)

Note that CLIP was trained on a dataset of **400 million images** scraped from the Internet

The Imagenet1K dataset has only **1.28 million training images**

CLIP is a much more capable "foundation model"

Resnet101

~44.5 million parameters

CLIP ViT-L/14@336px:

~428 million parameters

Limitations against fully supervised models

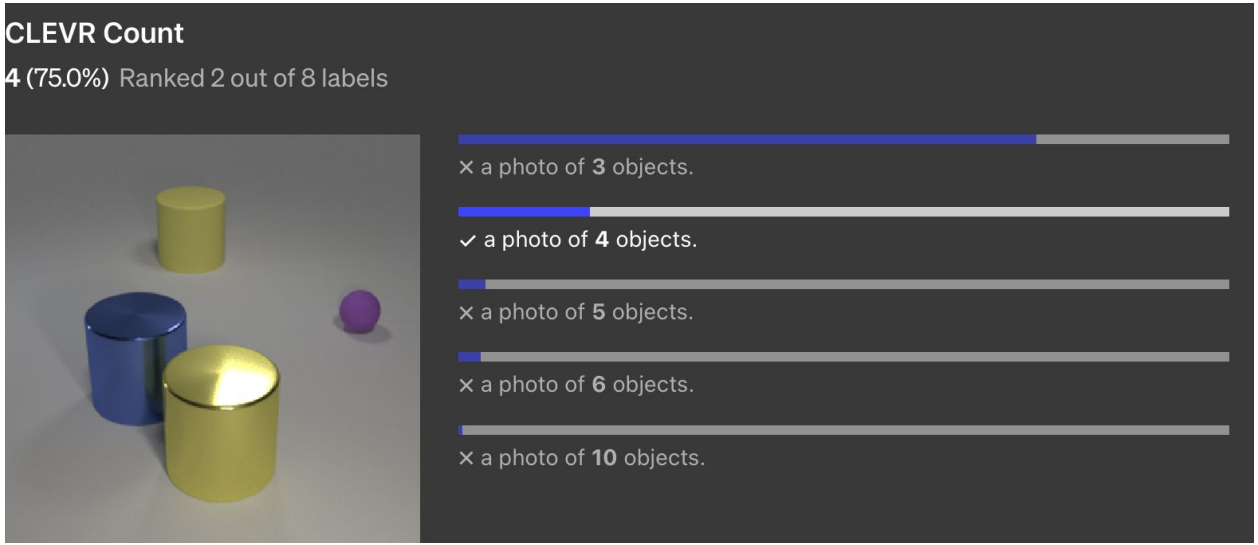
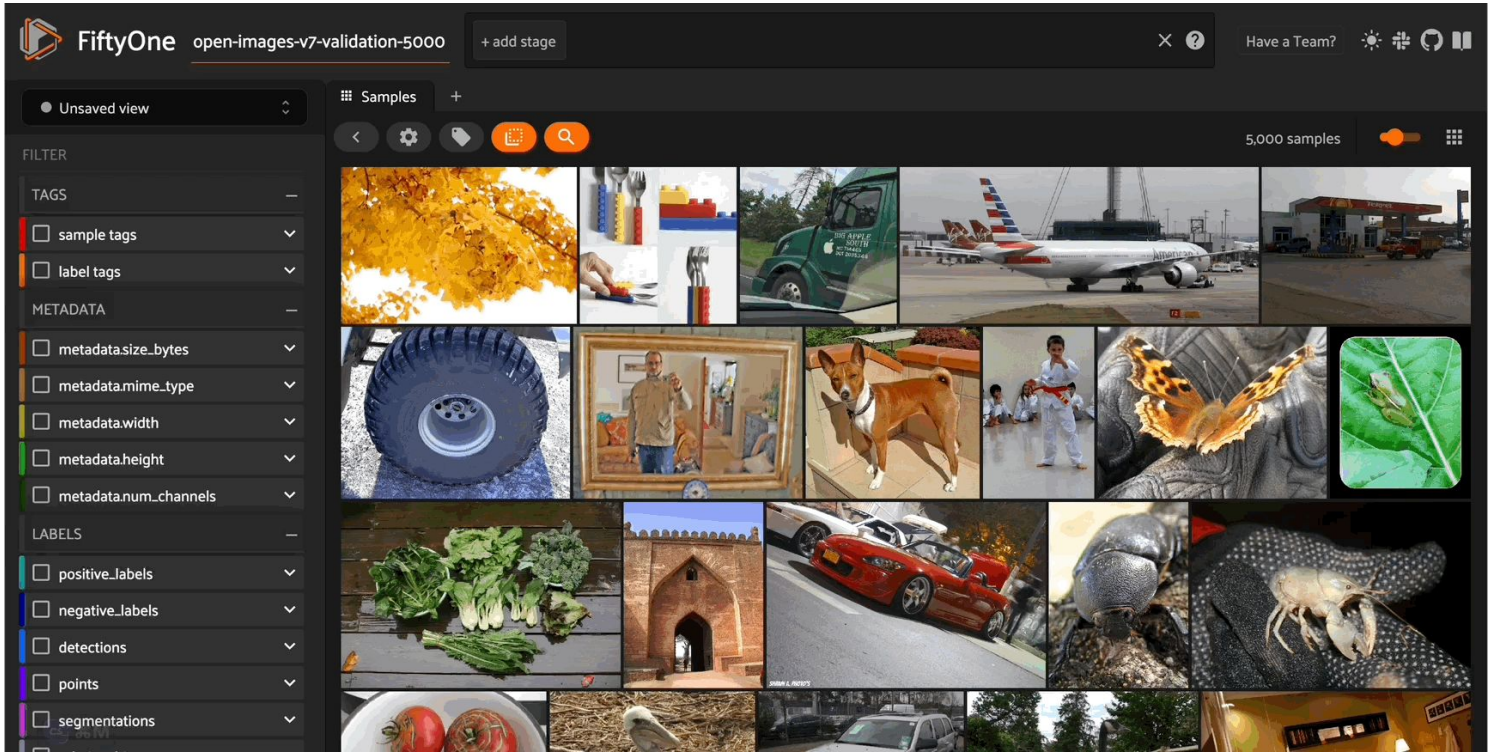


Image from
https://openai.com/index/clip/

The authors comment on the limitations section of the paper
that CLIP often fails to perform as well as a fully supervised model fine-tuned for the task
Notable examples:

- digit recognition (MNIST)
- tumor classification (PatchCamelyon)
- satellite imaging (EuroSAT)
- object counting (CLEVR Count)

Semantic search with CLIP



CLIP guides image generation of diffusion models

Prompt:

"Create an image of an astronaut riding a horse in pencil drawing style"

Image by OpenAI's Dall-E 3

Summary

CLIP learns joint embeddings

- CLIP creates a shared embedding space for images and text
- These multimodal embeddings have applications on semantic search and image generation

CLIP excels at zero-shot classification

- CLIP performs well on unseen tasks without additional training

CLIP has limitations

- CLIP is not always better than models fine-tuned for specific tasks
- Retraining CLIP with a ViT base is expensive both computationally and data-wise

Further reading

Learning Transferable Visual Models From Natural Language Supervision

- https://arxiv.org/abs/2103.00020

A Google Search Experience for Computer Vision Data

- https://voxel51.com/blog/a-google-search-experience-for-computer-vision-data/

Multi-modal ML with OpenAI's CLIP

- https://www.pinecone.io/learn/series/image-search/clip/

Resources

- Github Repository
- YouTube playlist
- Discord channel
  #practical-computer-vision-workshops