CLIP Model: Contrastive Language-Image Pretraining

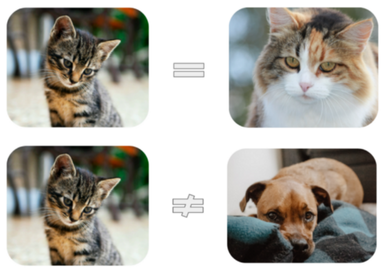
The clip model is serves as a multi-modal approach to convert and match a domain problem of language to the domain of vision. The model serves as a similarity engine to find the corresponding image vector of a 'phrase' or a 'word'.

The clip model is trained on using techniques from three different approaches: self-supervision, contrastive learning and zero-shot learning.

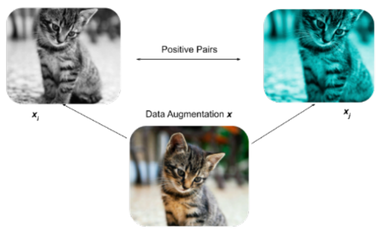
The model also uses a significant development in the space of NLP (Transformer), which has been carried over to vision space through (Vision Transformer).

Contrastive learning:

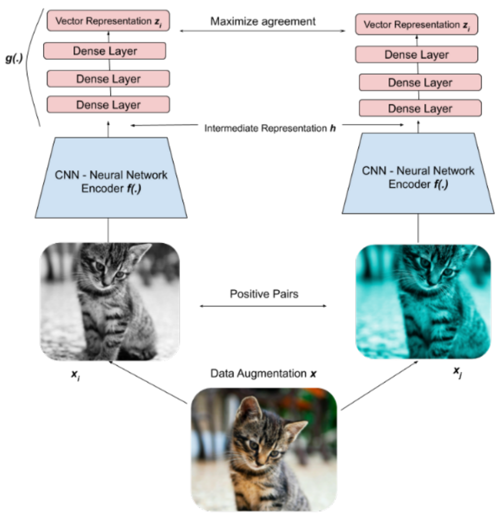
It is a technique of machine learning that trains models to learn which items are similar and which are distant without the use of the labelled data. It is self supervised task independent learning technique.



The way the technique works is that we create different transformations & augmentations of a picture and create positive pairs and negative pairs for the model to train on. The model learns general features of the dataset to see which pairs are similar and which are not.

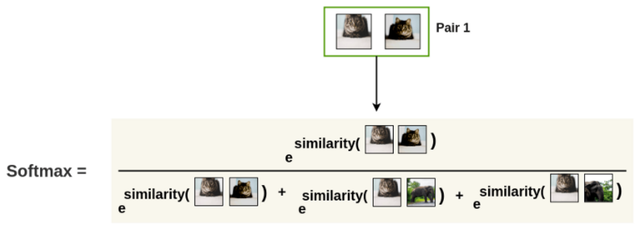


We use Big CNN networks to convert these images to their vector representations and use a contrastive objective to maximize the similarity between the two input streams (i.e. minimize the contrastive loss function).

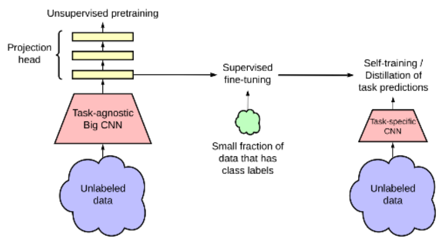

Vector Representation z_i
Maximize agreement
Vector Representation z_j
$g(.)$
Dense Layer
Dense Layer
Dense Layer
Intermediate Representation h
CNN - Neural Network Encoder $f(.)$
CNN - Neural Network Encoder $f(.)$
Positive Pairs
x_i
Data Augmentation x
x_j

The steps of the process are then:

1. Data augmentation & creative positive pairs
2. Encode the images using standard CNN networks "h = f(x)", where 'x' is augmented image, to encode both the images as vector representations.
3. The output is feeded into a Dense Layer projection head "z=g(h)", which transforms data into another space (it is empirically shown to improve performance)
4. We minimize the cosine similarity on the representations. The loss used is: Normalized Temperature-Scaled Cross-Entropy Loss



5. We can improve the performance of the model by using a semi-supervised methodology: unsupervised pre-train, supervised fine-tune, knowledge distillation paradigm

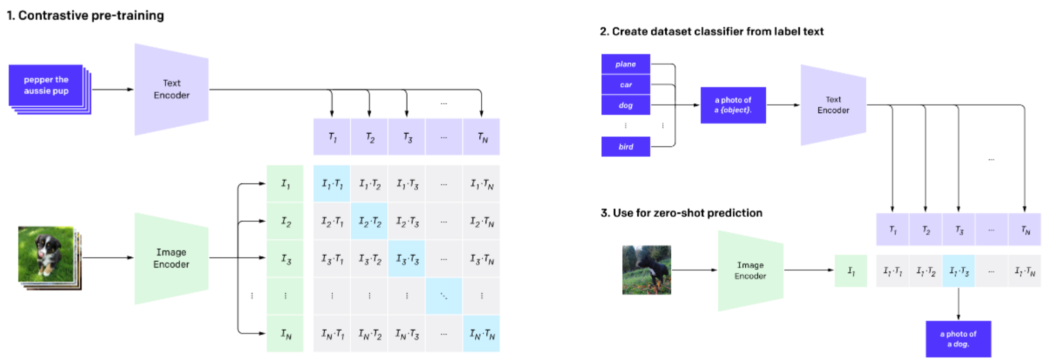


CLIP Model:

The overall process of the model is to take a text, use a standard tokenizer to tokenize it and feed it into a transformer.

The image input passes through a Vision transformer, whose initial layers differ to intake an image.

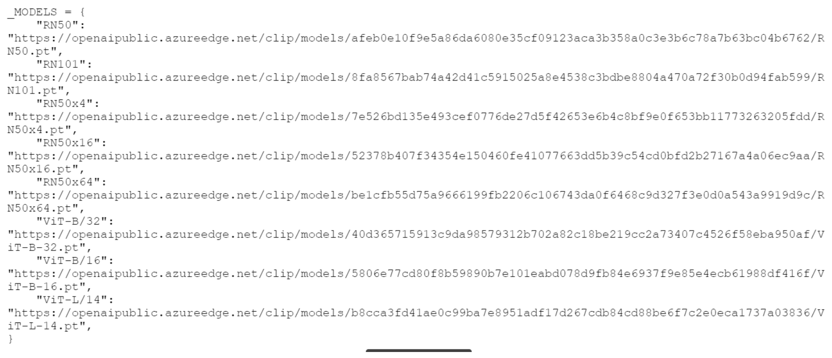
The architecture after the image processing layer is the same as of (with multiple attention heads) the text transformer and both the transformers are trained in a joint manner. The purpose of the joint training is to create a multi-modal encoding at the end.


1. Contrastive pre-training
pepper the aussie pup
Text Encoder
Image Encoder
T_1 T_2 T_3 ... T_N
I_1 I_2 I_3 ... I_N
$I_1 \cdot T_1$ $I_1 \cdot T_2$ $I_1 \cdot T_3$ ... $I_1 \cdot T_N$
$I_2 \cdot T_1$ $I_2 \cdot T_2$ $I_2 \cdot T_3$ ... $I_2 \cdot T_N$
$I_3 \cdot T_1$ $I_3 \cdot T_2$ $I_3 \cdot T_3$ ... $I_3 \cdot T_N$
$I_N \cdot T_1$ $I_N \cdot T_2$ $I_N \cdot T_3$ ... $I_N \cdot T_N$
2. Create dataset classifier from label text
plane
car
dog
bird
a photo of a {object}.
Text Encoder
3. Use for zero-shot prediction
Image Encoder
I_1
$I_1 \cdot T_1$ $I_1 \cdot T_2$ $I_1 \cdot T_3$ ... $I_1 \cdot T_N$
a photo of a dog.

This encoding generated lies in a common space of vision & text, and is used to identify the similarity between text and images

The model can be further improved as well by fine-tuning on labelled datasets by swapping the final layers of the model to train on a respective loss function.

Models:



```
_MODELS = {
    "RN50":
"https://openaipublic.azureedge.net/clip/models/afeb0e10f9e5a86da6080e35cf09123aca3b358a0c3e3b6c78a7b63bc04b6762/R
N50.pt",
    "RN101":
"https://openaipublic.azureedge.net/clip/models/8fa8567bab74a42d41c5915025a8e4538c3bdbe8804a470a72f30b0d94fab599/R
N101.pt",
    "RN50x4":
"https://openaipublic.azureedge.net/clip/models/7e526bd135e493cef0776de27d5f42653e6b4c8bf9e0f653bb11773263205fdd/R
N50x4.pt",
    "RN50x16":
"https://openaipublic.azureedge.net/clip/models/52378b407f34354e150460fe41077663dd5b39c54cd0bfd2b27167a4a06ec9aa/R
N50x16.pt",
    "RN50x64":
"https://openaipublic.azureedge.net/clip/models/be1cfb55d75a9666199fb2206c106743da0f6468c9d327f3e0d0a543a9919d9c/R
N50x64.pt",
    "ViT-B/32":
"https://openaipublic.azureedge.net/clip/models/40d365715913c9da98579312b702a82c18be219cc2a73407c4526f58eba950af/V
iT-B-32.pt",
    "ViT-B/16":
"https://openaipublic.azureedge.net/clip/models/5806e77cd80f8b59890b7e101eabd078d9fb84e6937f9e85e4ecb61988df416f/V
iT-B-16.pt",
    "ViT-L/14":
"https://openaipublic.azureedge.net/clip/models/b8cca3fd41ae0c99ba7e8951adf17d267cdb84cd88be6f7c2e0eca1737a03836/V
iT-L-14.pt",
}
```

The clip model uses a number of variation of architecture as an image encoder and uses a masked self-attention Transformer as a text encoder. These encoders are trained to maximize the similarity of (image, text) pairs via a contrastive loss.

RN50, RN101, RN 50x4, etc. are modified ResNet versions which are used for the Vision encoding and

Model	**Layers**	**Hidden size D**	**MLP size**	**Heads**	**Params**
ViT-Base	12	768	3072	12	86M
ViT-Large	24	1024	4096	16	307M
ViT-Huge	32	1280	5120	16	632M

The “Base” and “Large” models are directly adopted from BERT and the larger “Huge” models. For instance, ViT-L/16 means the “Large” variant with 16×16 input patch size. The transformer’s sequence length is inversely proportional to the square of the patch size, thus models with smaller patch size are computationally more expensive.

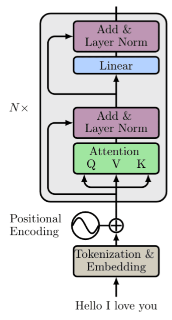
The model has a fixed dimensionality of 512, which stays constant through out the layers so that a joint training is possible with the vision transformer, and there is a possibility to add short residual skip connections between different layers of the block.

The model is also capped at 49,152 vocab size for now. The max sequence length is capped at 76, which means we cannot add a string longer than that. The text sequence is bracketed with [SOS] and [EOS] tokens and the activations of the highest layer of the transformer at the [EOS] token are treated as the feature representation of the text which is layer normalized and then linearly projected into the multi-modal embedding space

Layers: Please see appendix

Transformer:

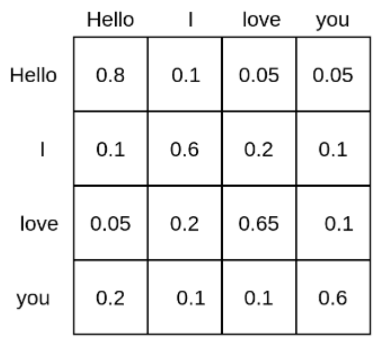
The basic building block of a transformer model is placed here



The steps of the process are as follows:

1. Covert the text sequence into tokenization sets (order irrelevant)
2. Create word embeddings in a continued space of low dimensional vectors simultaneously.
3. Add positional encodings to the embeddings in the form of small constants to create word vectors
   - sinusoidal function for the positional encoding. The sine function tells the model to pay attention to a particular wavelength λ. Given a signal y(x) = sin (kx), the wavelength will be k = (2π) / λ. In our case the λ will be dependent on the position in the sentence. 'I' is used to distinguish between odd and even positions.
4. These are passed to the encoder block: each block has below layers & can be repeated
5. Add layer of self-attention which uses dot product attention values as a measure for the similarity of two words
   - "Self-attention, sometimes called intra-attention, is an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence." ~ Ashish Vaswani et al. from Google Brain.

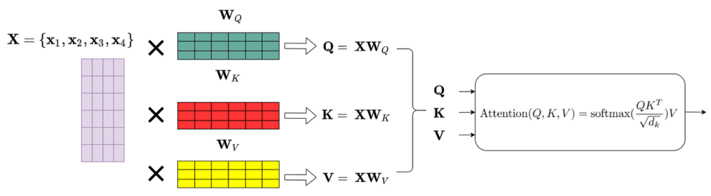
Self-attention
Probability score matrix



	Hello	I	love	you
Hello	0.8	0.1	0.05	0.05
I	0.1	0.6	0.2	0.1
love	0.05	0.2	0.65	0.1
you	0.2	0.1	0.1	0.6

Softmax(Attention)
equation

- Each attention layer has three attention weight matrices for Query, Key & Value (an idea copied from information retrieval theory) and they run in parallel.


$\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4\}$
$\mathbf{W}_Q$
$\mathbf{W}_K$
$\mathbf{W}_V$
$\mathbf{Q} = \mathbf{X}\mathbf{W}_Q$
$\mathbf{K} = \mathbf{X}\mathbf{W}_K$
$\mathbf{V} = \mathbf{X}\mathbf{W}_V$
Q
K
V
$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$

6. Normalization layer is added
7. Short residual skip connections are added around the previous two sublayers
   - These enable a higher-level understanding of last layers to be passed down to the previous layers (a top-down influence)
8. We add Layer Normalization across channels and spatial dimensions
9. We add linear/trainable/dense layers (fully connected layers)
   - This projects the output of the self-attention to high dimensional space
10. Lastly, we add one more normalization layer followed by second residual connection layer

The model can be improved by running through multiple attention heads (multi-head attention), rather than a single self-attention matrix – which are laid out in parallel to learn/focus on different areas of the text (just like receptive fields in CNNs)

- These heads are concatenated and transformed using a weight matrix
- This gives multiple paths for the model to understand the input and capture more positional information
- To compensate for the extra complexity, the output vector size is divided by the number of heads. Specifically, in the vanilla transformer, they use d (model)=512 & h=8 heads, which gives us vector representations of 64.

Vision Transformer:

The vision transformer adds a few layers before a standard transformer to convert the image into a series of flattened linear embeddings. This helps in creating a Transformation Invariant training process.

The vision transformer works in the following steps:

1. Split an image into patches which are similar to sequence tokens (like words)
   - An image is represented as

i. 3D Image (X) ∈ resolution R^{HxWxC}

- reshape the 3D image into flattened 2D patches

ii. Patch Image (X_p) ∈ $R^{Nx(P\text{^}2\ .\ C)}$

- Where sequence length N = H.W / P^2 and (P, P) is the resolution of each image patch. Each patch is a D dimension vector with a trainable linear projection

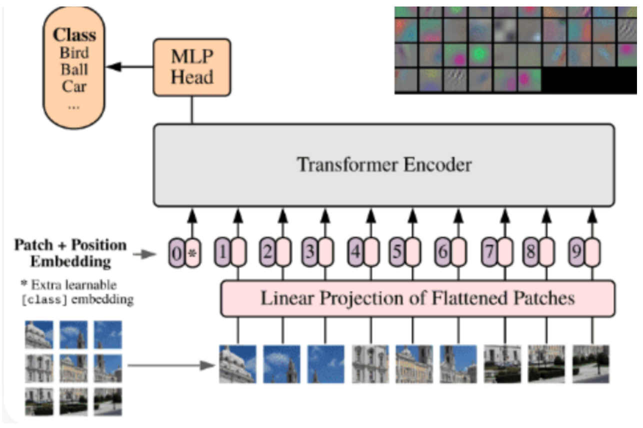
2. Flatten the patches and produce lower-dimensional linear embeddings from the flattened patches
   - E.g., a (16, 16) will be converted into a 16x16 vector
   - Similar to BERT's [class] token, we prepend a learnable embedding to the sequence of embedded patches ($z_0^0 = x_{class}$)

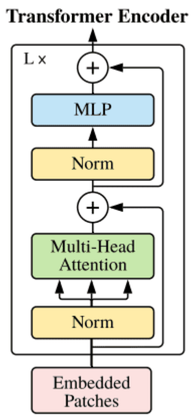
      i. $z^0 = [x_{class}; x^1_pE; x^2_pE; \cdots; x^N_p E] + E_{pos}$, $E \in R^{(P\text{^}2C)\times D}$, $E_{pos} \in R^{(N+1)\times D}$

      ii. X_{class} is a class label and X^N_p is patch images N ∈ 1 to n

      iii. Using the transformer encoder to pre-train we always need a Class label at the 0th position. When we pass the patch images as inputs we always need to prepend one classification token as the first patch as shown in the figure.

3. Add positional embeddings (this is a trainable layer)
4. Feed the sequence as an input to a standard transformer encoder (there can be multiple attention heads and parallel blocks in this space to increase the size of the network)
5. Pretrain the model with image labels (fully supervised on a huge dataset)
6. Finetune on the downstream dataset for image classification


Class
Bird
Ball
Car
...
MLP Head
Transformer Encoder
Patch + Position Embedding
* Extra learnable [class] embedding
0 * 1 2 3 4 5 6 7 8 9
Linear Projection of Flattened Patches



Appendix

Text Encoder:

1) ResidualAttentionBlock:
    - MultiheadAttention -> NonDynamicallyQuantizableLinear (both are defined in pytorch)
    - LayerNorm (called from torch)
    - Sequential:
        - Linear (from torch)
        - QuickGELU (activation function created from sigmoid activation)
        - Linear
    - LayerNorm
2) ResidualAttentionBlock:
    - MultiheadAttention -> NonDynamicallyQuantizableLinear (both are defined in pytorch)
    - LayerNorm (called from torch)
    - Sequential:
        - Linear (from torch)
        - QuickGELU (activation function created from sigmoid activation)
        - Linear
    - LayerNorm
3) ResidualAttentionBlock:
    - MultiheadAttention -> NonDynamicallyQuantizableLinear (both are defined in pytorch)
    - LayerNorm (called from torch)
    - Sequential:
        - Linear (from torch)
        - QuickGELU (activation function created from sigmoid activation)
        - Linear
    - LayerNorm

4) ResidualAttentionBlock:

- MultiheadAttention -> NonDynamicallyQuantizableLinear (both are defined in pytorch)
- LayerNorm (called from torch)
- Sequential:
  - Linear (from torch)
  - QuickGELU (activation function created from sigmoid activation)
  - Linear
- LayerNorm

5) ResidualAttentionBlock:

- MultiheadAttention -> NonDynamicallyQuantizableLinear (both are defined in pytorch)
- LayerNorm (called from torch)
- Sequential:
  - Linear (from torch)
  - QuickGELU (activation function created from sigmoid activation)
  - Linear
- LayerNorm

5) ResidualAttentionBlock:

- MultiheadAttention -> NonDynamicallyQuantizableLinear (both are defined in pytorch)
- LayerNorm (called from torch)
- Sequential:
  - Linear (from torch)
  - QuickGELU (activation function created from sigmoid activation)
  - Linear
- LayerNorm

5) ResidualAttentionBlock:

- MultiheadAttention -> NonDynamicallyQuantizableLinear (both are defined in pytorch)
- LayerNorm (called from torch)
- Sequential:
  - Linear (from torch)
  - QuickGELU (activation function created from sigmoid activation)

Linear

LayerNorm

6) ResidualAttentionBlock:

MultiheadAttention -> NonDynamicallyQuantizableLinear (both are defined in pytorch)

LayerNorm (called from torch)

Sequential:

Linear (from torch)

QuickGELU (activation function created from sigmoid activation)

Linear

LayerNorm

7) ResidualAttentionBlock:

MultiheadAttention -> NonDynamicallyQuantizableLinear (both are defined in pytorch)

LayerNorm (called from torch)

Sequential:

Linear (from torch)

QuickGELU (activation function created from sigmoid activation)

Linear

LayerNorm

8) ResidualAttentionBlock:

MultiheadAttention -> NonDynamicallyQuantizableLinear (both are defined in pytorch)

LayerNorm (called from torch)

Sequential:

Linear (from torch)

QuickGELU (activation function created from sigmoid activation)

Linear

LayerNorm

9) ResidualAttentionBlock:

MultiheadAttention -> NonDynamicallyQuantizableLinear (both are defined in pytorch)

LayerNorm (called from torch)

Sequential:

        Linear (from torch)

        QuickGELU (activation function created from sigmoid activation)

        Linear

    LayerNorm

10) ResidualAttentionBlock:

    MultiheadAttention -> NonDynamicallyQuantizableLinear (both are defined in pytorch)

    LayerNorm (called from torch)

    Sequential:

        Linear (from torch)

        QuickGELU (activation function created from sigmoid activation)

        Linear

    LayerNorm

11) ResidualAttentionBlock:

    MultiheadAttention -> NonDynamicallyQuantizableLinear (both are defined in pytorch)

    LayerNorm (called from torch)

    Sequential:

        Linear (from torch)

        QuickGELU (activation function created from sigmoid activation)

        Linear

    LayerNorm

12) ResidualAttentionBlock:

    MultiheadAttention -> NonDynamicallyQuantizableLinear (both are defined in pytorch)

    LayerNorm (called from torch)

    Sequential:

        Linear (from torch)

        QuickGELU (activation function created from sigmoid activation)

        Linear

    LayerNorm

Embedding

LayerNorm