Vision Transformers (ViT) Explained

Jump to section

Pinecone lets you implement semantic, audio, or visual search into your applications using vector search. But first you need to convert your data into vector embeddings, and vision transformers do that for images. This article introduces vision transformers, how they work, and how to use them.

Vision and language are the two big domains in machine learning. Two distinct disciplines with their own problems, best practices, and model architectures. At least, that *was* the case.

The **Vi**sion **T**ransformer (ViT)[1] marks the first step towards the merger of these two fields into a single unified discipline. For the first time in the history of ML, a single model architecture has come to dominate both language *and vision*.

Before ViT, transformers were *"those language models"* and nothing more. Since then, ViT and further work has solidified them as a likely contender for the architecture that merges the two disciplines.

This article will dive into ViT, explaining and visualizing the intuition behind how and why it works. Later, we'll look at how to implement it ourselves.

Transformers

Transformers were introduced in 2017 by Vaswani et al. in the now-famous paper *"Attention is All You Need"* [2]. The primary function powering these models is the *attention mechanism*.

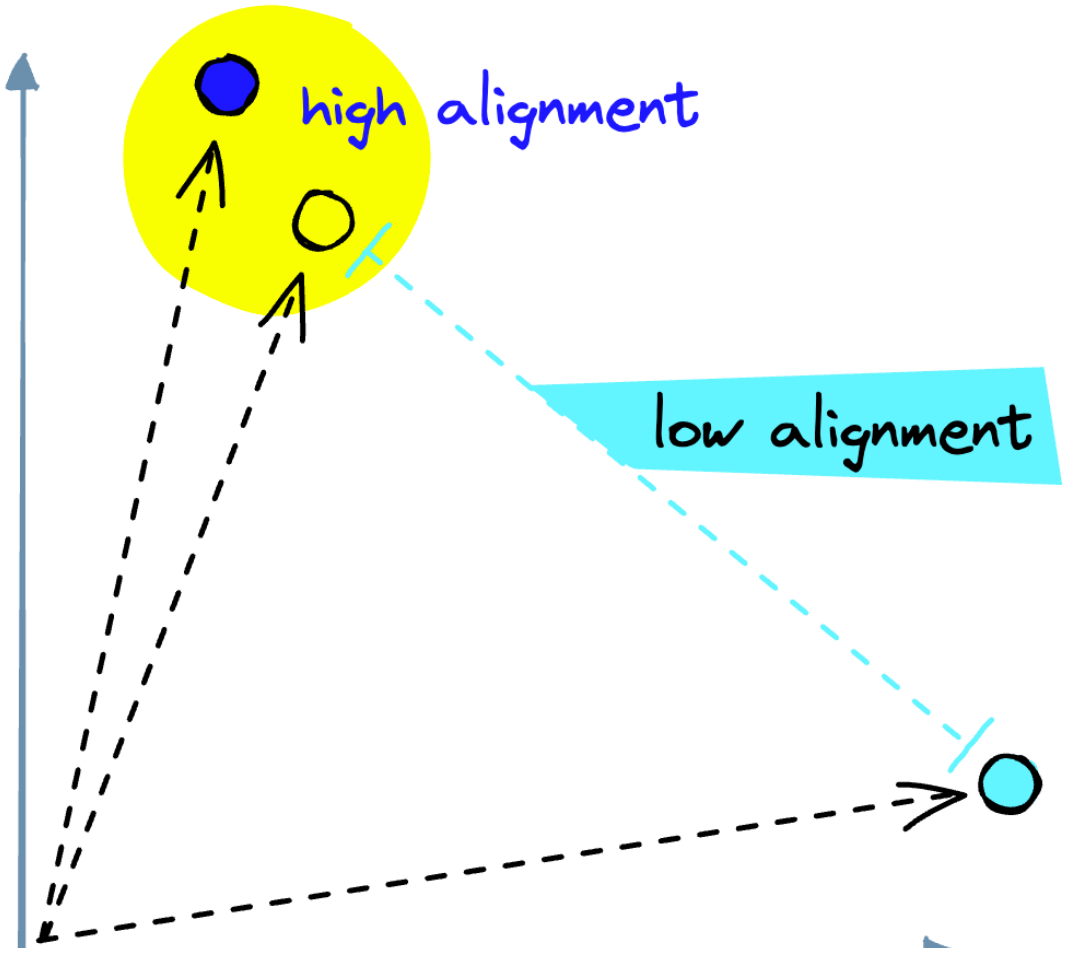
Attention 101

In NLP, attention allows us to consider the context of words and *focus attention* on the key relationships between different tokens (represented as word or sub-word tokens).

It works by comparing "token embeddings" and calculating an *"alignment"* score that describes how similar two tokens are based on their semantic and contextual meaning.

In the layers preceding the attention layer, each word embedding is encoded into a *"vector space"*.

In this vector space, similar tokens share a similar location. Therefore, when we calculate the dot product between token embeddings (inside the attention mechanism), we return a high *alignment* score when embeddings are aligned in vector space. When embeddings are *not aligned*, we produce a low alignment score.



Alignment between vectors is higher where vectors share similar direction and magnitude.

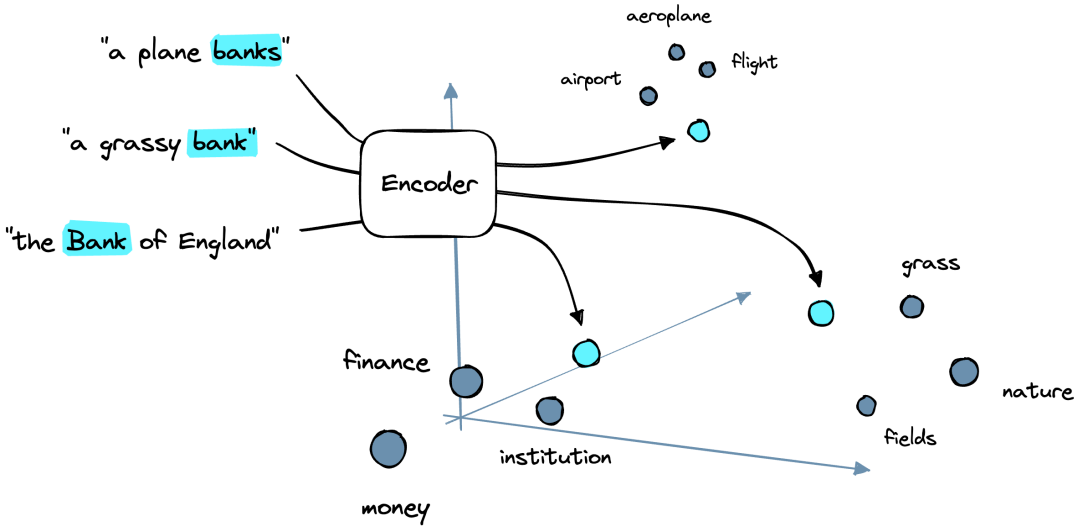
Before applying attention, our tokens' initial positions are based purely on a “general meaning” of a particular word or sub-word token.

As we go through several encoder blocks (these include the attention mechanism), the position of these embeddings is updated to better reflect the meaning of a token *with respect* to its context. The context being all of the other words within that specific sentence.

So, given three phrases:

- A plane **banks**
- The grassy **bank**
- The **Bank** of England

The initial embedding for the token *bank* is equal. Yet, the token is pushed towards its context-based meaning through many attention encoder blocks. These blocks might push *bank* towards tokens like [plane, airport, flight], [nature, fields, outdoors], or [finance, England, money].



An encoder with attention layers can add contextual meaning to embeddings.

Attention has been used in **C**onvolutional **N**eural **N**etworks (CNNs) over the years. Generally speaking, this has been shown to produce *some benefit* but is often computationally limited.

Attention is a heavy operation and does not scale to large sequences. Therefore, attention can *only* be used in later CNN layers — where the number of pixels has been reduced. This limits the potential benefit of attention as it cannot be applied across the complete set of network layers [1] [3].

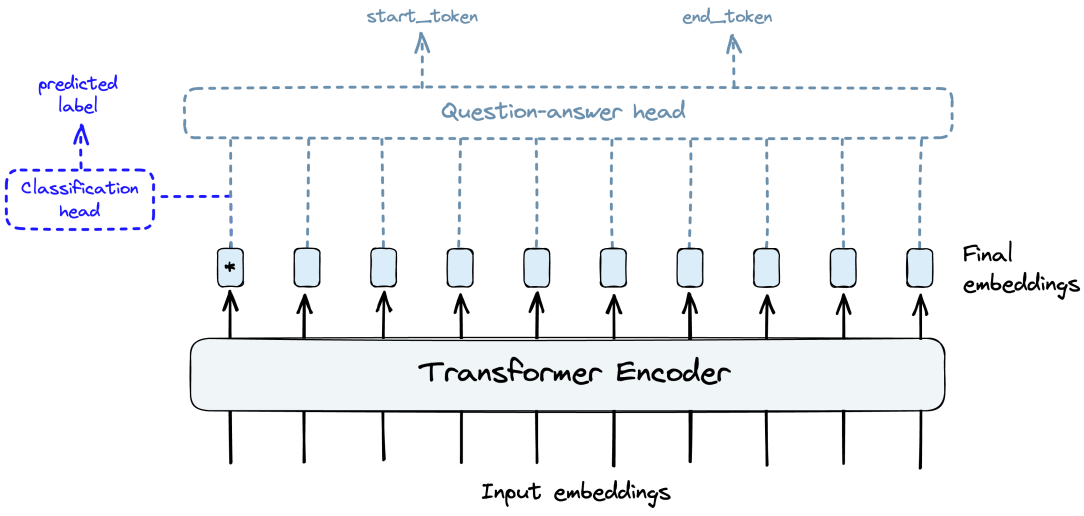
Transformer models do not have this limitation and instead apply attention over many layers.

BERT, a well-known transformer architecture, uses several "encoder" blocks. Each of these blocks consists of normalization layers, *multi-head* attention (i.e., several parallel attention operations) layers, and a multilayer perceptron (MLP) component.

Each of these encoder *"blocks"* encodes *more information* into the token (or patch) embeddings using their context. This operation produces a *deeper* semantic representation of each token.

At the end of this process, we get super information-rich embeddings. These embeddings are the ultimate output of the *core* of a transformer, including ViT.

Another set of layers is used to transform these rich embeddings into useful predictions. These final few layers are called the *"head"* and a different *head* is used for each task, such as for classification, **N**amed **E**ntity **R**ecognition (NER), question-answering, etc.



Example of a transformer encoder building information-rich final embeddings before passing these on to a task-specific "head".

ViT works similarly, but rather than consuming *word tokens*, ViT consumes *image patches*. The remainder of the transformer functions in the same way.

Images to Patch Embeddings

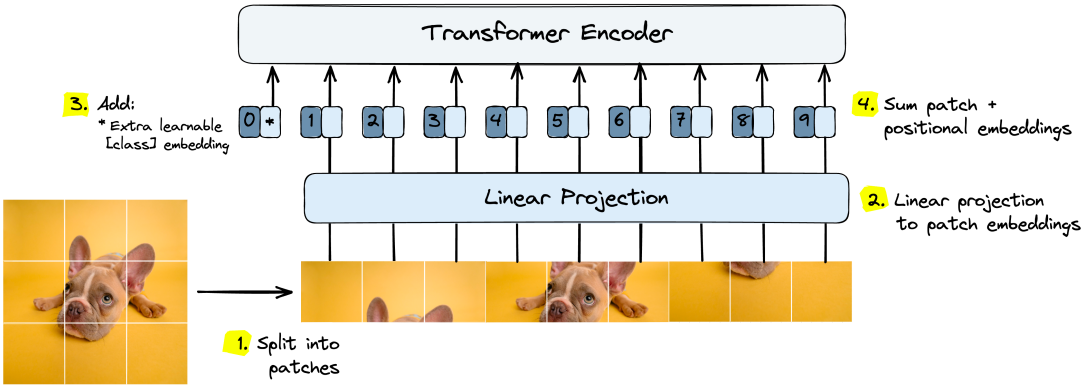
The new procedure introduced by ViT is limited to the first few processing steps. These first steps take us from images to a set of *patch embeddings*.

If we didn't split the images into patches, we could alternatively feed in pixel values of the image directly. However, this causes problems with the attention mechanism.

Attention requires the comparison of every input to all other inputs. If we perform that on a *224x224* pixel image, we must perform 224^4 ($2.5E^9$) comparisons. That's for a single attention layer, of which transformers contain several.

Doing this would be a computational nightmare far beyond the capabilities of even the latest GPUs and TPUs within a reasonable timeframe.

Therefore, we create image patches and embed those as patch embeddings. Our high-level process for doing this is as follows:

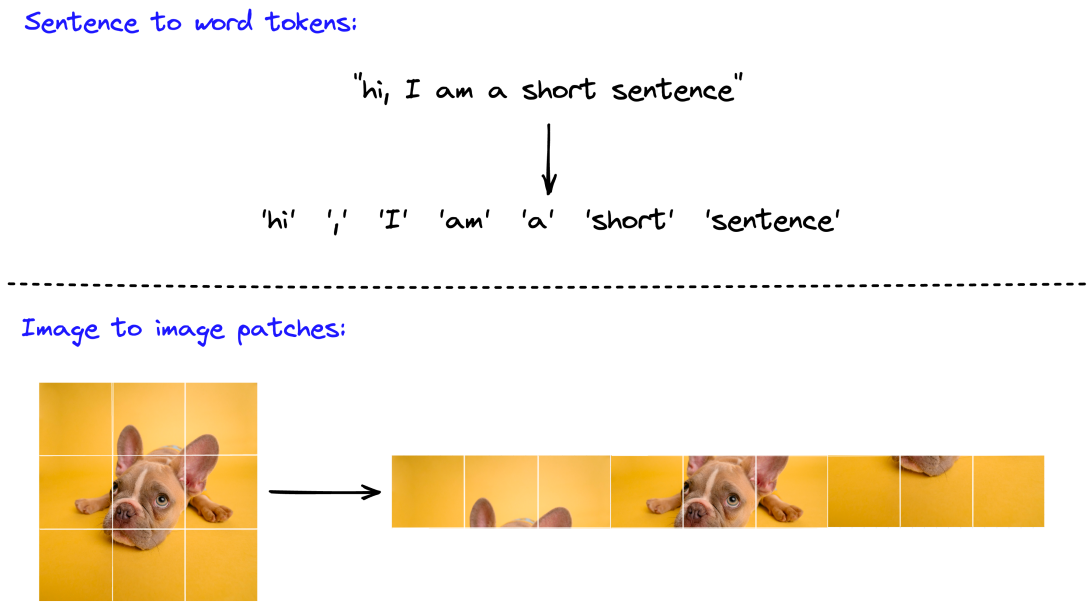


1. Split the image into image patches.
2. Process patches through the linear projection layer to get initial patch embeddings.
3. Preappend trainable *"class"* embedding to patch embeddings.
4. Sum patch embeddings and *learned positional embeddings*.

After these steps, we process the patch embeddings like token embeddings in a typical transformer. Let's dive into each of these components in more detail.

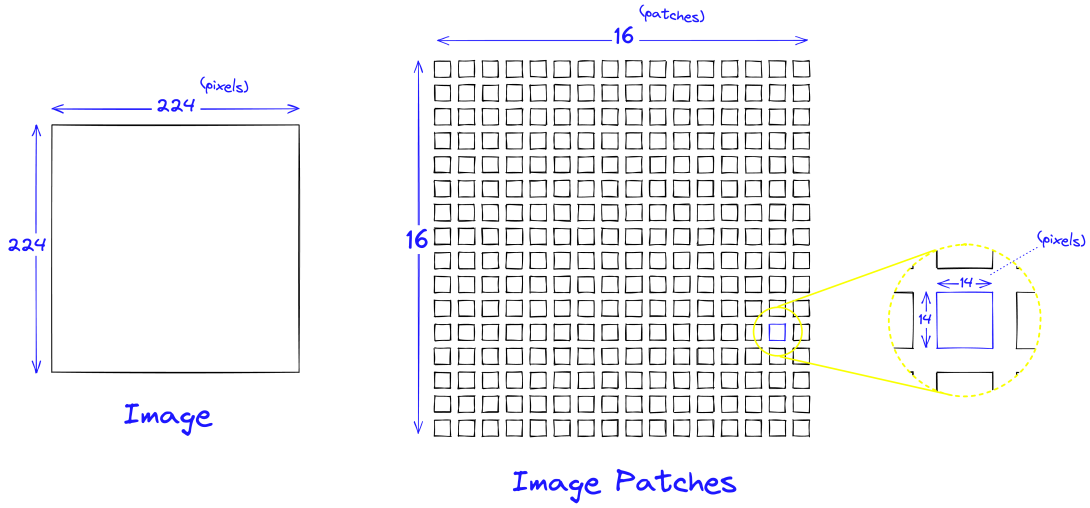
Image Patches

Our first step is the transformation of images into image patches. In NLP, we do the same thing. Images are sentences and patches are word or sub-word tokens.



NLP transformers and ViT both split larger sequences (sentences or images) into tokens or patches.

Recall that a *224x224* pixel image requires $2.5E^9$ comparisons. If, instead, we split a *224x224* pixel image into 256 *14x14* pixel image patches, a single attention layer requires a more manageable $256 * 14^4$ ($9.8e^6$) comparisons.

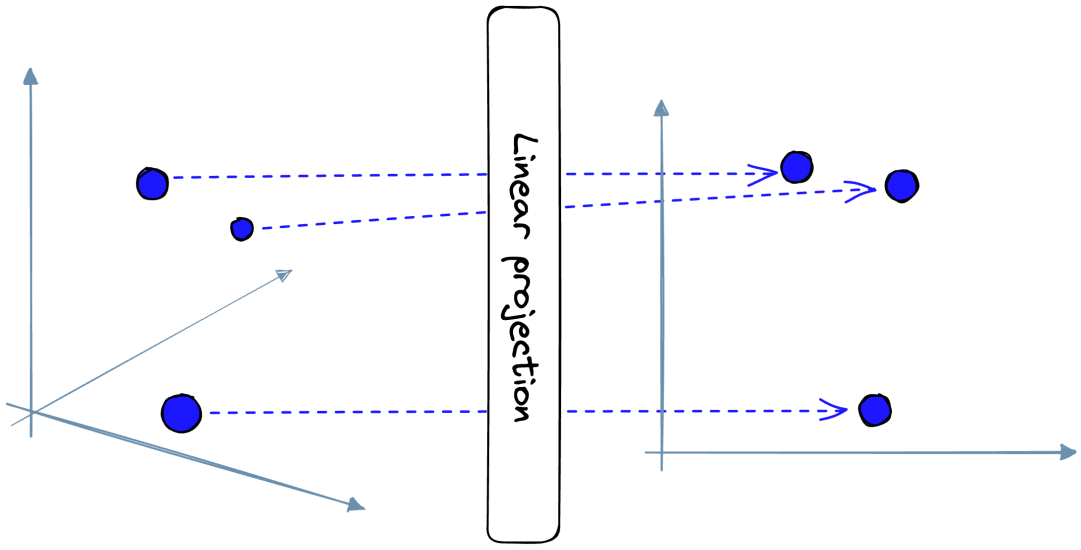


Conversion of 224x224 pixel image into 256 14x14 pixel image patches.

Through this, these image patches act as a form of *much needed* quantization required for effective use of attention.

Linear Projection

After building the image patches, a *linear projection* layer is used to map the image patch *"arrays"* to *patch embedding "vectors"*.



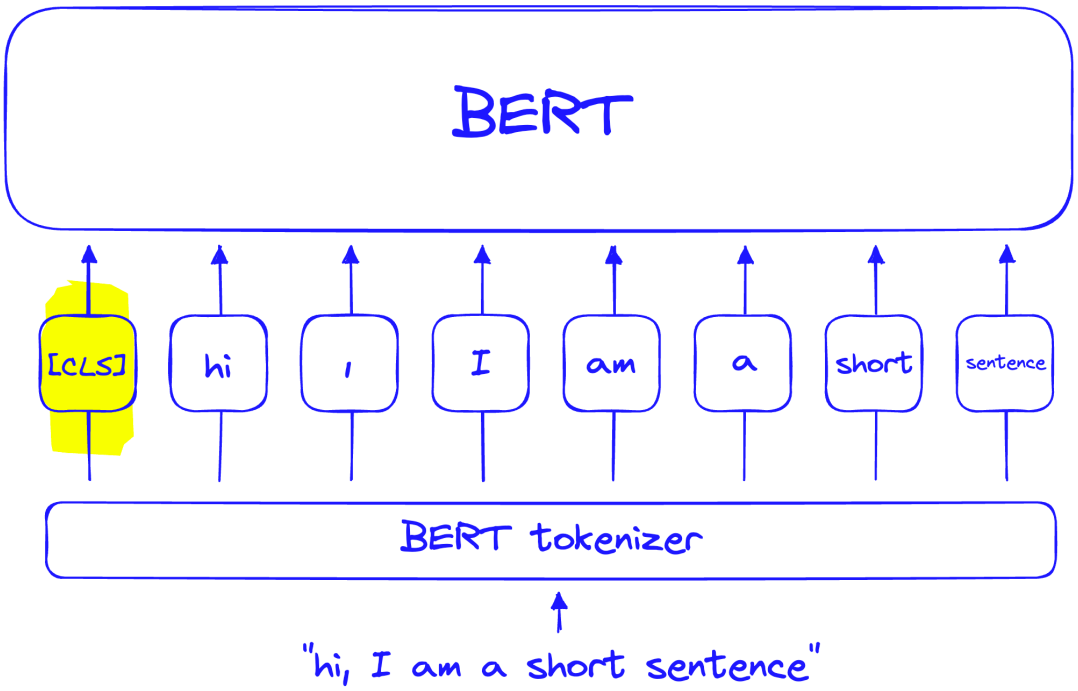
The linear projection layer attempts to transform arrays into vectors while maintaining

their “physical dimensions”. Meaning similar image patches should be mapped to similar patch embeddings.

By mapping the patches to embeddings, we now have the correct dimensionality for input into the transformer. However, two more steps remain before the embeddings are fully prepared.

Learnable Embeddings

One *feature* introduced to transformers with the popular BERT models was the use of a `[CLS]` (or *“classification”*) token. The `[CLS]` token was a *“special token”* prepended to every sentence fed into BERT[4].



The BERT [CLS] token is preappended to every sequence.

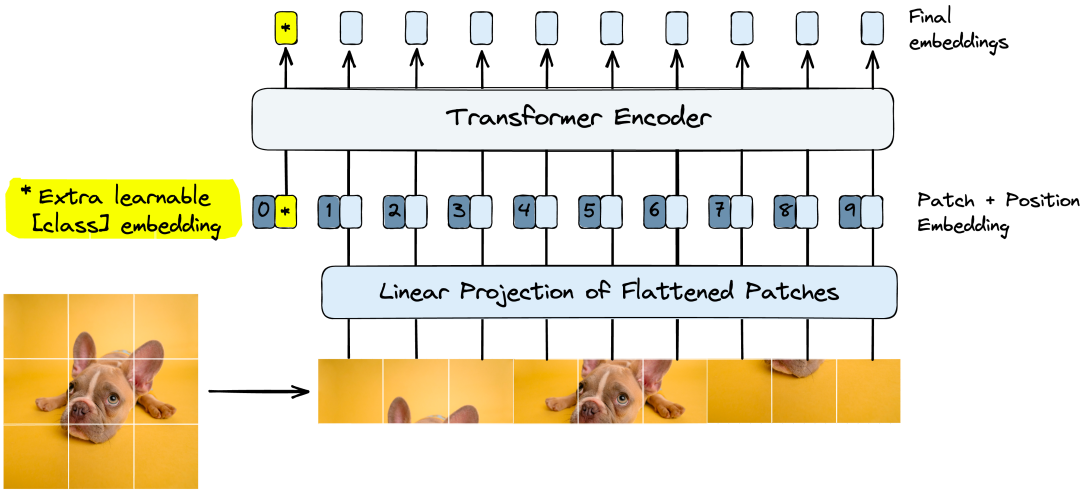
This `[CLS]` token is converted into a token embedding and passed through several encoding layers.

Two things make `[CLS]` embeddings special. First, it does *not* represent an actual token,

meaning it begins as a "blank slate" for each sentence. Second, the final output from the `[CLS]` embedding is used as the input into a classification head during pretraining.

Using a "blank slate" token as the sole input to a classification head pushes the transformer to learn to encode a *"general representation"* of the entire sentence into that embedding. The model *must* do this to enable accurate classifier predictions.

ViT applies the same logic by adding a *"learnable embedding"*. This learnable embedding is the same as the `[CLS]` token used by BERT.



ViT process with the learnable class embedding highlight (left).

The preferred pretraining function of ViT is based solely on classification, unlike BERT, which uses masked language modeling. Based on that, this learning embedding is *even more* important to the successful pretraining of ViT.

Positional Embeddings

Transformers do *not* have any default mechanism that considers the "order" of token or patch embeddings. Yet, *order* is essential. In language, the order of words can completely change their meaning.

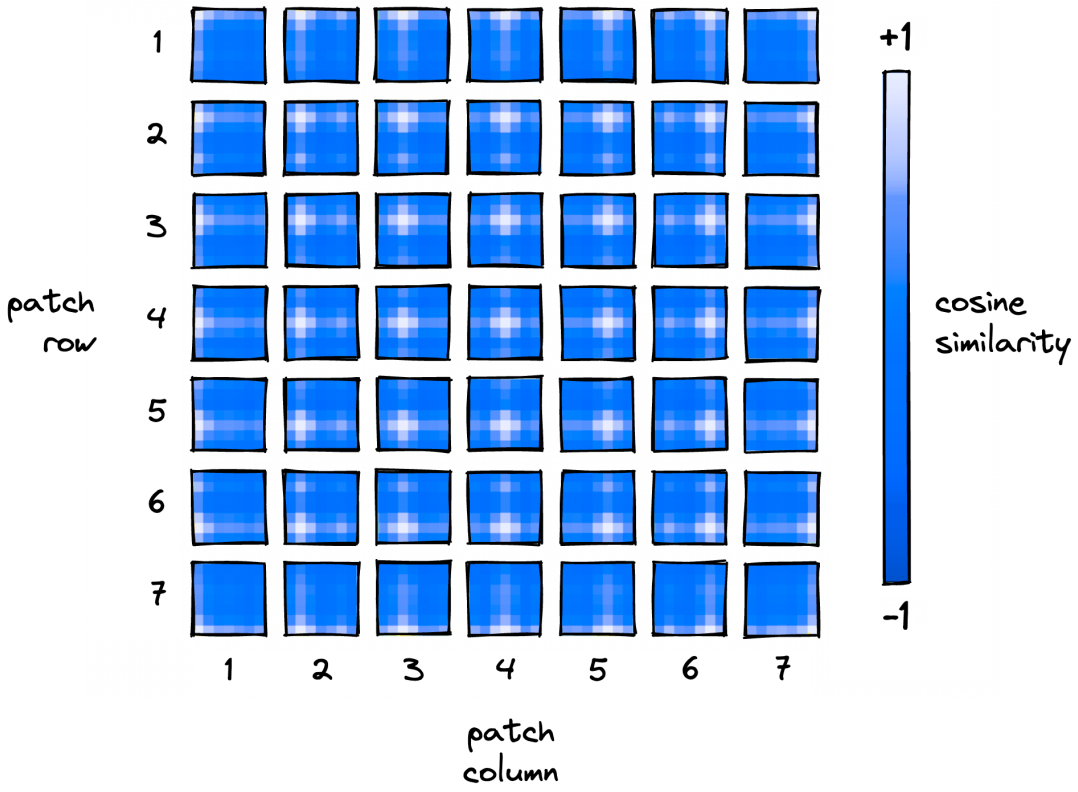
The same is true for images. If given a jumbled jigsaw set, it's hard-to-impossible for a person to accurately predict what the complete puzzle represents. This applies to transformers too. We need a way of enabling the model to infer the *order* or *position* of the

puzzle pieces.

We enable order with *positional embeddings*. For ViT, these positional embeddings are learned vectors with the same dimensionality as our patch embeddings.

After creating the patch embeddings and prepending the “class” embedding, we sum them all with positional embeddings.

These positional embeddings are learned during pretraining and (sometimes) during fine-tuning. During training, these embeddings converge into vector spaces where they show *high similarity* to their neighboring position embeddings — particularly those sharing the same column and row:



Cosine similarity between trained positional embeddings. Adapted from [1].

After adding the positional embeddings, our *patch embeddings* are complete. From here, we pass the embeddings to the ViT model, which processes them as a typical transformer model.

Implementation

Open Code Walkthrough

We've worked through the logic and innovations introduced by ViT. Let's now work through an example of implementing the model. We start by installing all of the libraries that we'll be using:

```
!pip install datasets transformers torch
```

We will fine-tune with a well-known image classification dataset called CIFAR-10. It can be downloaded via Hugging Face's *Datasets* library, and we'll download both the training *and* validation/test datasets.

In[2]:

```
# import CIFAR-10 dataset from HuggingFace
from datasets import load_dataset

dataset_train = load_dataset(
    'cifar10',
    split='train', # training dataset
    ignore_verifications=False  # set to True if seeing spli
)

dataset_train
```

Out[2]:

```
Dataset({
    features: ['img', 'label'],
    num_rows: 50000
})
```

In[3]:

```
dataset_test = load_dataset(
    'cifar10',
    split='test', # training dataset
    ignore_verifications=True  # set to True if seeing split
)

dataset_test
```

Out[3]:

```
Dataset({
    features: ['img', 'label'],
    num_rows: 10000
})
```

The training dataset contains 50K images across 10 classes. To find the human-readable class labels, we can do the following:

In[4]:

```
# check how many labels/number of classes
num_classes = len(set(dataset_train['label']))
labels = dataset_train.features['label']
num_classes, labels
```

Out[4]:

```
(10,
 ClassLabel(num_classes=10, names=['airplane', 'automobile',
```

Every record in the dataset contains an `img` and `label` feature. The `img` values are all Python PIL objects with *32x32* pixel resolution and three color channels, **r**ed, **g**reen, and **b**lue (RGB).

In[5]:

```
1  dataset_train[0]
```

Out[5]:

```
{'img': <PIL.PngImagePlugin.PngImageFile image mode=RGB size=
 'label': 0}
```

In[6]:

```
1  dataset_train[0]['img']
```

Out[6]:

<PIL.PngImagePlugin.PngImageFile image mode=RGB size=32x32 at

In[7]:

```
1  dataset_train[0]['label'], labels.names[dataset_train[0]['l
```

Out[7]:

```
(0, 'airplane')
```

Feature Extractor

Preceding the ViT model, we use something called a *feature extractor*. The feature extractor is used to *preprocess* images into normalized and resized image *"pixel_values"* tensors. We initialize it from the Hugging Face *Transformers* library like so:

In[8]:

```
from transformers import ViTFeatureExtractor

# import model
model_id = 'google/vit-base-patch16-224-in21k'
feature_extractor = ViTFeatureExtractor.from_pretrained(
    model_id
)
```

In[9]:

```
feature_extractor
```

Out[9]:

```
ViTFeatureExtractor {
  "do_normalize": true,
  "do_resize": true,
  "feature_extractor_type": "ViTFeatureExtractor",
  "image_mean": [
    0.5,
    0.5,
    0.5
  ],
  "image_std": [
```

```
    0.5,
    0.5,
    0.5
  ],
  "resample": 2,
  "size": 224
}
```

The feature extractor configuration shows that normalization and resizing are set to true. Normalization is performed across the three color channels using the mean and standard deviation values stored in `"image_mean"` and `"image_std"` respectively. The output size is set by `"size"` at *224x224* pixels.

To process an image with the feature extractor, we do the following:

```
In[10]:
```

```
example = feature_extractor(
    dataset_train[0]['img'],
    return_tensors='pt'
)
example
```

```
Out[10]:
```

```
{'pixel_values': tensor([[[[ 0.3961,  0.3961,  0.3961,  ...,
          [ 0.3961,  0.3961,  0.3961,  ...,  0.2941,  0.2941,
          [ 0.3961,  0.3961,  0.3961,  ...,  0.2941,  0.2941,
          ...,
          [-0.1922, -0.1922, -0.1922,  ..., -0.2863, -0.2863,
          [-0.1922, -0.1922, -0.1922,  ..., -0.2863, -0.2863,
          [-0.1922, -0.1922, -0.1922,  ..., -0.2863, -0.2863,
         ...
         [[ 0.4824,  0.4824,  0.4824,  ...,  0.3647,  0.3647,
          [ 0.4824,  0.4824,  0.4824,  ...,  0.3647,  0.3647,
```

```
          [ 0.4824,  0.4824,  0.4824,  ...,  0.3647,  0.3647,
          ...,
          [-0.2784, -0.2784, -0.2784,  ..., -0.3961, -0.3961,
          [-0.2784, -0.2784, -0.2784,  ..., -0.3961, -0.3961,
          [-0.2784, -0.2784, -0.2784,  ..., -0.3961, -0.3961,


In[11]:
```

```
example['pixel_values'].shape
```

```
Out[11]:

torch.Size([1, 3, 224, 224])
```

Later we'll be fine-tuning our ViT model with these tensors. Although fine-tuning is *not* as computationally heavy as pretraining, it still takes time. Therefore we *ideally* should be running everything on GPU rather than CPU. So, we move these tensors to a CUDA-enabled GPU *if* it is available.

```
import torch
#  if cuda enabled GPU is available, use it
device = torch.device(
    'cuda' if torch.cuda.is_available() else 'cpu'
)
patches = patches.to(device)
```

Fortunately, the `Trainer` utility we will use for fine-tuning later *does handle* this move for our data by default. Still, we will need to later repeat this step for the model.

To apply this preprocessing step across the entire dataset more efficiently, we will package into a function called `preprocess` and apply the transformations using the

`with_transform` method, like so:

```
def preprocess(batch):
    # take a list of PIL images and turn them to pixel values
    inputs = feature_extractor(
        batch['img'],
        return_tensors='pt'
    )
    # include the labels
    inputs['label'] = batch['label']
    return inputs

# apply to train-test datasets
prepared_train = dataset_train.with_transform(preprocess)
prepared_test = dataset_test.with_transform(preprocess)
```

Loading ViT

The next step is downloading and initializing ViT. Again, we're using Hugging Face *Transformers* with the same `from_pretrained` method used to load the feature extractor.

```
from transformers import ViTForImageClassification

labels = dataset_train.features['label'].names

model = ViTForImageClassification.from_pretrained(
    model_name_or_path,
    num_labels=len(labels)  # classification head
)
# move to GPU (if available)
model.to(device)
```

Because we are fine-tuning ViT for classification, we use the `ViTForImageClassification` class. By default, this will initialize a classification head with just two outputs.

We have *10* classes in CIFAR-10, so we must specify that we'd like to initialize the head with *10* outputs. We do this via the `num_labels` parameter.

Now we're ready to move on to fine-tuning.

Fine-Tuning

We will implement fine-tuning using Hugging Face's `Trainer` function. `Trainer` is an abstracted training and evaluation loop implemented in PyTorch for transformer models.

There are several variables that we must define beforehand. First, we start with the collate function. Collate helps us handle the collation of our dataset into batches of tensors that we will be fed into the model during training.

```
def collate_fn(batch):
    return {
        'pixel_values': torch.stack([x['pixel_values'] for x in b
        'labels': torch.tensor([x['label'] for x in batch])
    }
```

Another important variable is the *evaluation metric* to measure our model performance over time. We will use a simple *accuracy* metric calculated as:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Where:

TP: True Positives

TN: True Negatives

FP: False Positives

FN: False Negatives

We implement this using *Datasets* metrics, defined in the `compute_metrics` function:

```
import numpy as np
from datasets import load_metric

# accuracy metric
metric = load_metric("accuracy")
def compute_metrics(p):
    return metric.compute(
        predictions=np.argmax(p.predictions, axis=1),
        references=p.label_ids
    )
```

The final variable required by `Trainer` is the `TrainingArguments` configuration. These are simply the training parameters, save settings, and logging settings.

```
from transformers import TrainingArguments

training_args = TrainingArguments(
  output_dir="./cifar",
  per_device_train_batch_size=16,
  evaluation_strategy="steps",
  num_train_epochs=4,
  save_steps=100,
  eval_steps=100,
  logging_steps=10,
  learning_rate=2e-4,
  save_total_limit=2,
  remove_unused_columns=False,
  push_to_hub=False,
  load_best_model_at_end=True,
)
```

With all this, we're ready to initialize `Trainer` and begin the training loop.

```
from transformers import Trainer

trainer = Trainer(
    model=model,
    args=training_args,
    data_collator=collate_fn,
    compute_metrics=compute_metrics,
    train_dataset=prepared_train,
    eval_dataset=prepared_test,
    tokenizer=feature_extractor,
)
# begin training
results = trainer.train()
```

Training will take some time, even on GPU. Once complete, the best version of the model

will be saved in the `output_dir` we set in the `TrainingArguments` config object.

Evaluation and Prediction

The `Trainer` performs evaluation during training but we can also perform a more qualitative check (or make a prediction) by passing a single image through the `feature_extractor` and `model`. We will use this image:

In[50]:

```
# show the first image of the testing dataset
image = dataset_test["img"][0].resize((200,200))
image
```

Out[50]:

```
<PIL.Image.Image image mode=RGB size=200x200 at 0x7FA9D072E0A(
```

In[60]:

```
# extract the actual label for this image
actual_label = dataset_test["label"][0]

labels = dataset_test.features['label']
actual_label, labels.names[actual_label]
```

```
Out[60]:

(3, 'cat')
```

The image isn't very clear, and most people would struggle to correctly classify the image. However, we can see from the label that this is a cat. Let's see what the model predicts.

```
In[6]:

from transformers import ViTForImageClassification

# import fine-tuned version of model from Hugging Face hub (
model_id = 'LaCarnevali/vit-cifar10'
model = ViTForImageClassification.from_pretrained(model_id)


In[30]:

inputs = feature_extractor(image, return_tensors="pt")

with torch.no_grad():
    logits = model(**inputs).logits


In[61]:

predicted_label = logits.argmax(-1).item()
labels = dataset_test.features['label']
labels.names[predicted_label]


Out[61]:

'cat'
```

Looks like the model is correct!

That concludes our introduction to the Vision Transformer and how to use it via Hugging Face *Transformers*. It's worth noting how quickly transformers have come to dominate NLP and, increasingly likely, computer vision in the near future.

Before 2021, transformers being used in anything but NLP was unheard of. Yet, despite being known as *"those language models"*, they have already found use in some of the most advanced computer vision applications. Transformers are a crucial component of diffusion models[5] and even Tesla's Full Self Driving[6].

As time progresses, we will undoubtedly see both fields continue to merge and more real-world applications of transformers in both domains.

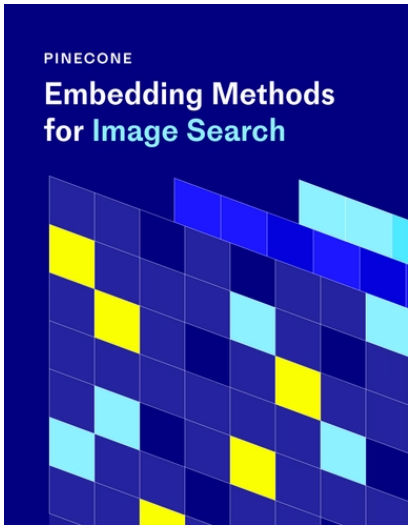
Resources

Previous

Convolutional Neural Nets

Next

CLIP Explained



Embedding Methods for Image Search

Chapters