ARTIFICIAL INTELLIGENCE

PROJECT ON VISION TRANSFORMER BY USING PYTHON

By:

Perugu Sisira Reddy.

ABSTRACT

Vision Transformers, or ViTs, have become an exciting new way to look at computer vision. They are starting to shake up the old stronghold of convolutional neural networks, known as CNNs. While CNNs focus on images with local operations, ViTs take a different route. They use the transformer design directly on image patches. This leads to a more adaptable & scalable method for understanding visuals.

In this abstract, we dive into the basic ideas behind ViTs. We shine a light on their setup, how they get trained, & what they are used for. Some important points include how they break down images into tokens (or patches), use positional encoding, employ multi-head self-attention methods, and rely on feedforward neural networks.

The advantages of ViTs are noteworthy. They can capture global dependencies quite well! Plus, they deal with different input sizes without much trouble. We also touch upon the recent progress in this area, face some challenges, and consider possible future paths for ViTs in computer vision.

All in all, Vision Transformers show a significant shift towards using transformer models for recognizing images. This opens up many new paths for research & application in artificial intelligence.

OBJECTIVE

The main goal of the Vision Transformer (ViT) is to use the transformer architecture well for computer vision tasks, especially image classification. For a long time, convolutional neural networks (CNNs) have been the leaders in this area. They are great at understanding details in images because they work with local features through convolutions.

But ViTs want to shake things up! Their method is. They cut images into smaller patches. Each patch is treated like a sequence of tokens, kind of like words in text processing tasks. By doing this, ViTs take advantage of the transformer's ability to see relationships over longer distances and understand how different parts of the image connect with each other globally, instead of just looking at them piece by piece.

Here are the key points about what ViTs aim to do:

1. Understanding the Whole Image: ViTs focus on understanding the entire image together. They use self-attention mechanisms, which help them to look at all patches at once. This can create stronger and more aware representations.

2. Being Scalable & Flexible: Unlike CNNs that need fixed input sizes because of their layers, ViTs can work on images of all kinds of sizes. They treat images as sequences of patches no matter the resolution, which is super useful when high-resolution or varied-size inputs are needed.

3. Using Transformers Effectively: The ViTs take the transformer model, meant for data that comes in order like text, and show how self-attention can apply to spatial data such as images, This opens new doors for using transformer models across different areas and kinds of data.

4. Boosting Performance: ViTs aim to perform as well or even better than CNNs on popular tests like ImageNet. With smart training techniques and careful attention mechanisms, ViTs want to show they can be very effective at classifying images and maybe even help with tasks like object detection & segmentation.

In conclusion, the Vision Transformer wants to dig into what transformers can do in computer vision. It aims to create a fresh way to understand images that works alongside traditional convolutional methods—and maybe even goes beyond them.

INTRODUCTION

In recent times, Vision Transformers (ViTs) have really changed the game in computer vision. They're shaking things up and challenging the old favourites—those are the convolutional neural networks (CNNs). CNNs did super well with tasks like image classification, but ViTs are offering a fresh approach by using transformer design straight on visual data.

Now, let's talk about how we got from CNN to ViTs. CNNs have been the backbone of progress in computer vision ever since AlexNet came along in 2012. They are great with breaking down features through lots of layers, using convolutions & pooling to efficiently grab hold of local patterns. But here's the catch: as images got bigger and datasets exploded, CNNs started struggling a bit. They had a hard time scaling up for larger images and understanding those bigger picture connections.

So, how does that transformer architecture fit into vision? Well, it was originally made for natural language processing (NLP), you know? It brought a whole new level to handling sequential data with its neat self-attention mechanism. ViTs took this cool idea & adapted it for vision tasks by looking at images like they're sequences made of patches. Each patch gets turned into tokens and goes through several layers of self-attention plus feedforward neural networks. It's quite fascinating.

Key Features of Vision Transformers:

Patch Embedding : Images are divided into fixed-size patches, which are then flattened and linearly embedded into a sequence of tokens.

Self-Attention Mechanism : This allows the model to weigh the importance of different patches relative to each other, enabling it to capture long-range dependencies.

Positional Encoding : Since transformers do not have a built-in notion of order, positional encodings are added to the patch embeddings to retain spatial information.

Advantages of Vision Transformers:

Global Contextual Understanding: Vision Transformers, or ViTs for short, are really good at catching those long-distance connections in images. They help with understanding complicated relationships & the important features that matter.

Scalability: Unlike CNNs, which often have a tough time with different input sizes, ViTs manage various resolutions & aspect ratios without breaking a sweat. They use a neat patch-based approach.

Transfer Learning: These transformers get a boost from being pre-trained on huge datasets, much like what's done in NLP. This means they can easily pass on what they've learned to other tasks with just a little fine-tuning, making them super versatile!

METHODOLOGY

Understanding Vision Transformers with PyTorch : A Beginner's Guide

In the rapidly evolving world of artificial intelligence, the introduction of Vision Transformers (ViTs) has marked a significant shift in how we approach image processing tasks. Unlike traditional convolutional neural networks (CNNs), ViTs leverage the power of transformer architectures, originally designed for natural language processing, to analyze visual data. This article will guide you through the fundamentals Vision Transformers, how they work, and how to implement them using PyTorch.

What Are Vision Transformers?

Vision Transformers are a novel approach to image classification and other vision tasks that utilize the transformer architecture. They were introduced in the paper "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale" by Dosovitskiy et al. in 2020. The key innovation lies in treating image patches as sequences, similar to how words are treated in sentences.

Why Use Vision Transformers?

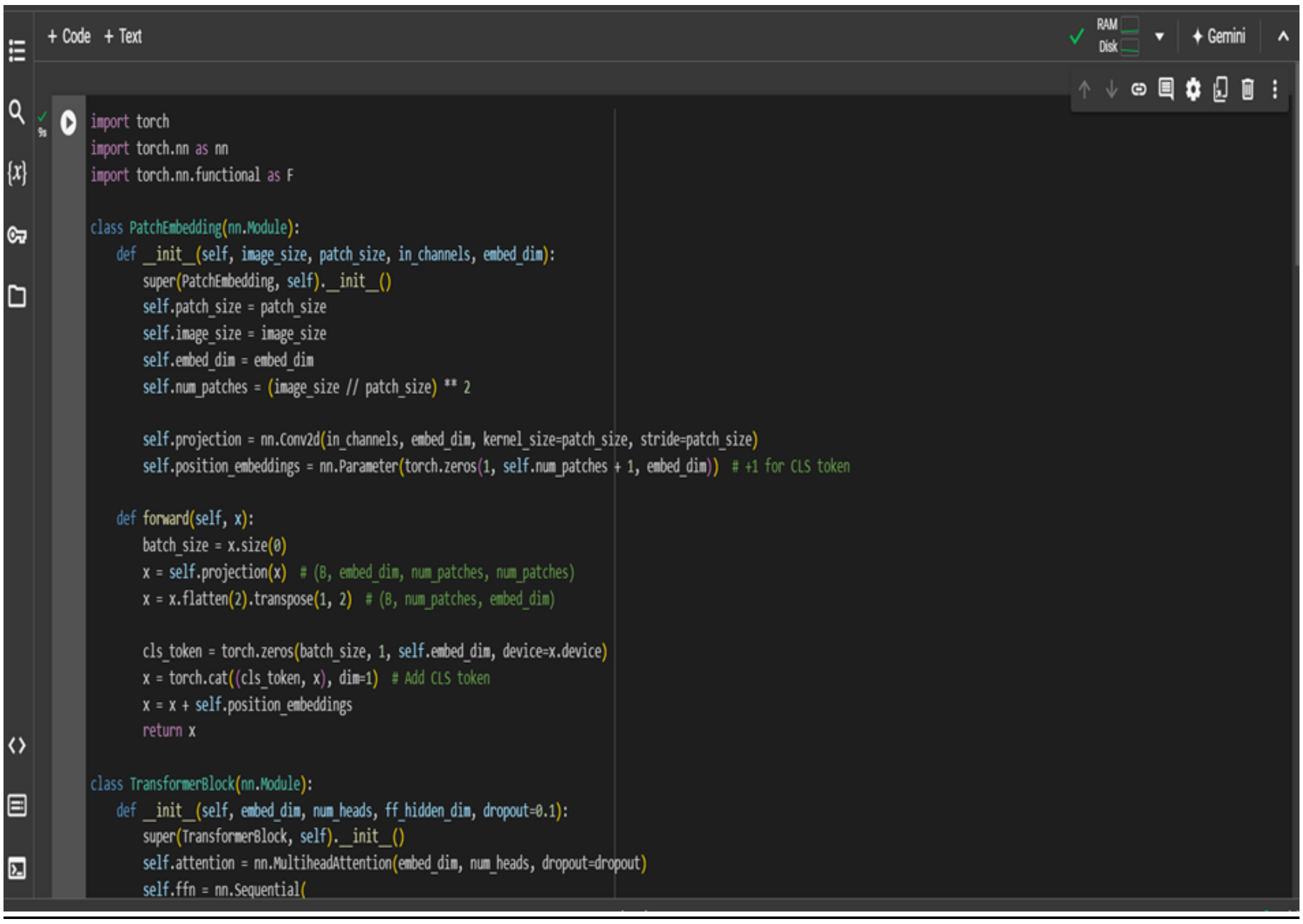
The shift from CNNs to Vision Transformers has several advantages:

1. Scalability : ViTs can be scaled up easily, allowing for larger models that can capture more complex patterns.

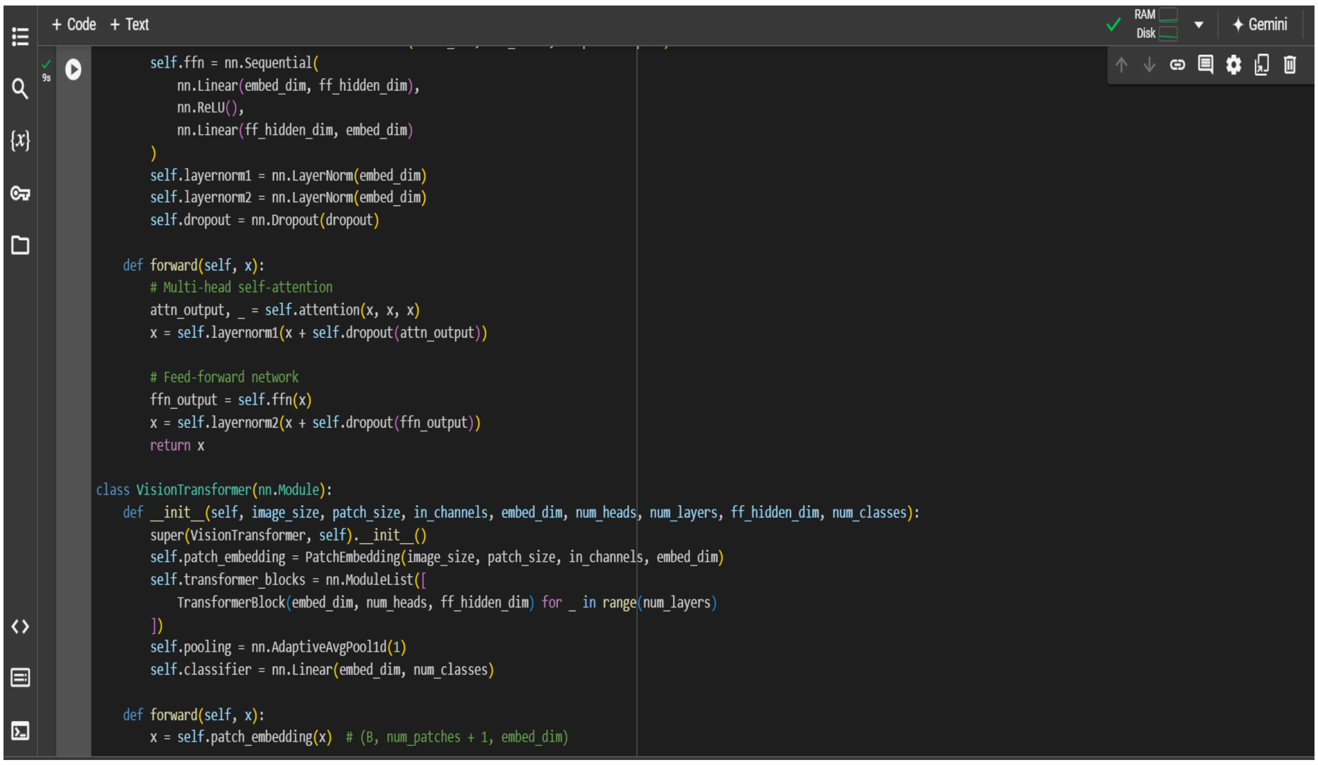
2. Transfer Learning : They have shown remarkable performance on various datasets, making them suitable for transfer learning.

3. Flexibility : Vision Transformers can be adapted for various tasks beyond classification, such as object detection and segmentation.

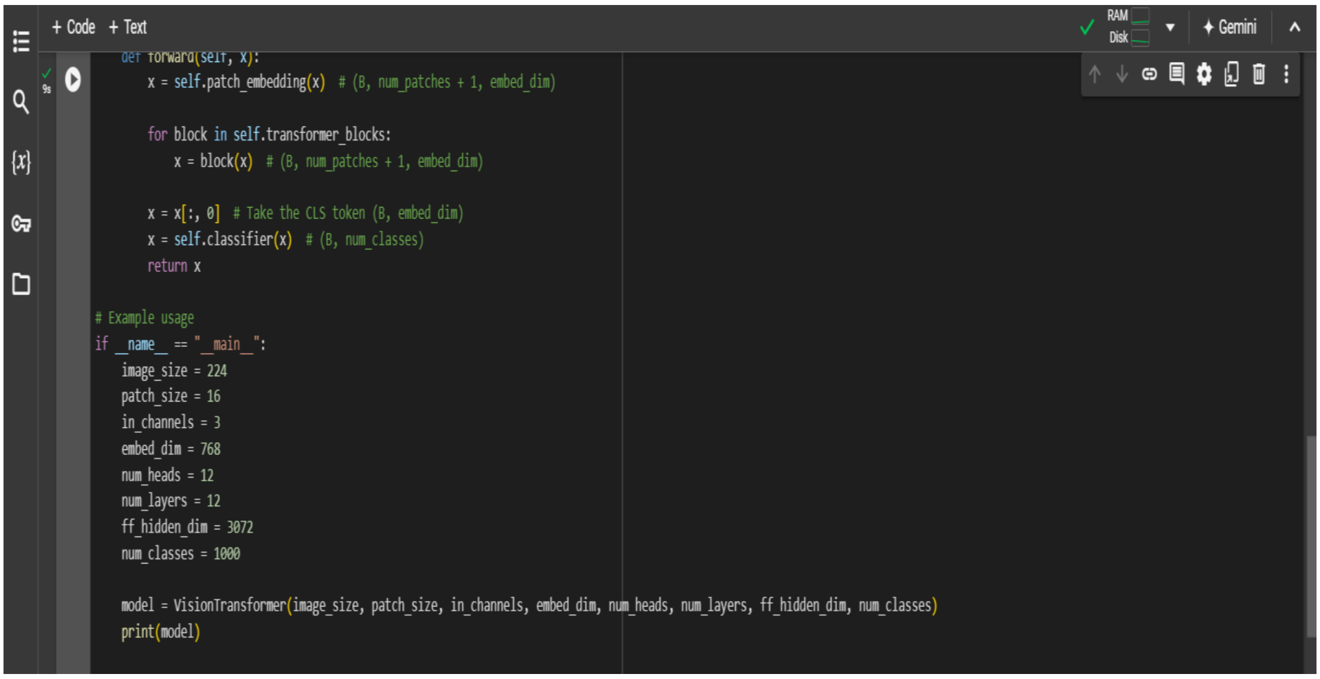
CODE



```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class PatchEmbedding(nn.Module):
    def __init__(self, image_size, patch_size, in_channels, embed_dim):
        super(PatchEmbedding, self).__init__()
        self.patch_size = patch_size
        self.image_size = image_size
        self.embed_dim = embed_dim
        self.num_patches = (image_size // patch_size) ** 2

        self.projection = nn.Conv2d(in_channels, embed_dim, kernel_size=patch_size, stride=patch_size)
        self.position_embeddings = nn.Parameter(torch.zeros(1, self.num_patches + 1, embed_dim))  # +1 for CLS token

    def forward(self, x):
        batch_size = x.size(0)
        x = self.projection(x)  # (B, embed_dim, num_patches, num_patches)
        x = x.flatten(2).transpose(1, 2)  # (B, num_patches, embed_dim)

        cls_token = torch.zeros(batch_size, 1, self.embed_dim, device=x.device)
        x = torch.cat((cls_token, x), dim=1)  # Add CLS token
        x = x + self.position_embeddings
        return x

class TransformerBlock(nn.Module):
    def __init__(self, embed_dim, num_heads, ff_hidden_dim, dropout=0.1):
        super(TransformerBlock, self).__init__()
        self.attention = nn.MultiheadAttention(embed_dim, num_heads, dropout=dropout)
        self.ffn = nn.Sequential(
```

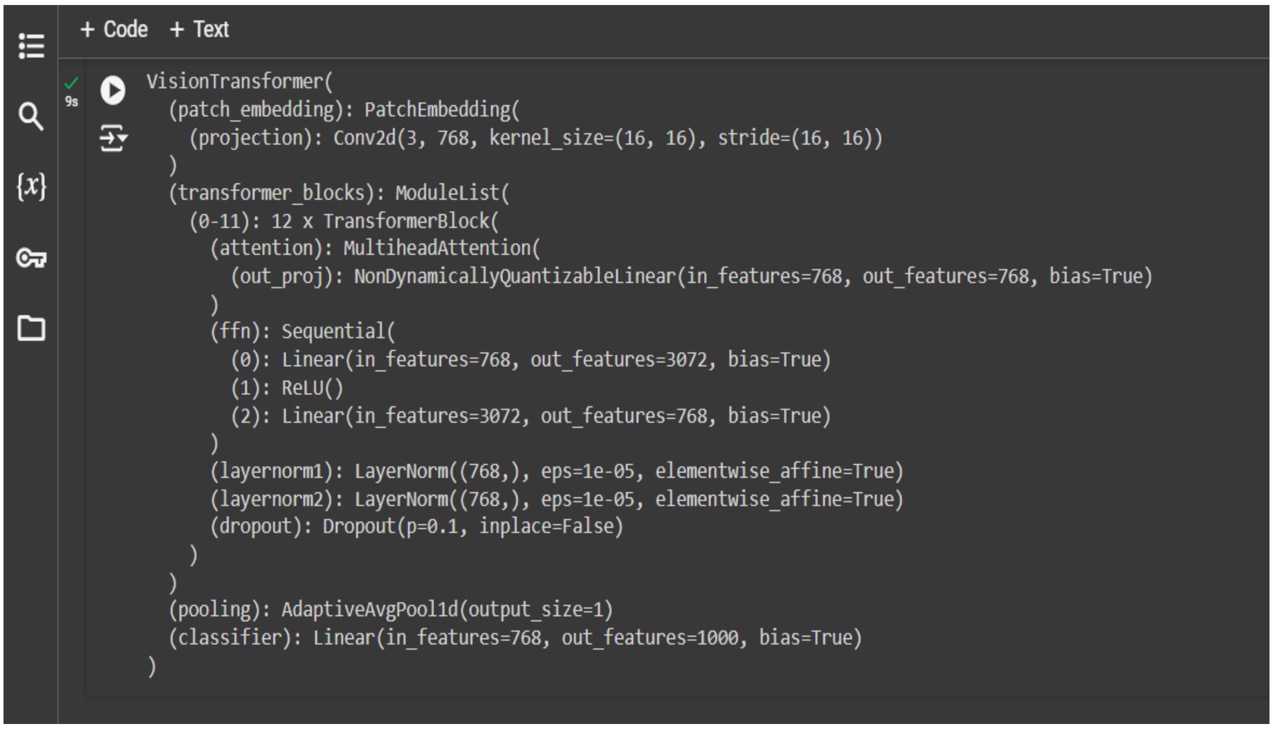


```python
        self.ffn = nn.Sequential(
            nn.Linear(embed_dim, ff_hidden_dim),
            nn.ReLU(),
            nn.Linear(ff_hidden_dim, embed_dim)
        )
        self.layernorm1 = nn.LayerNorm(embed_dim)
        self.layernorm2 = nn.LayerNorm(embed_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        # Multi-head self-attention
        attn_output, _ = self.attention(x, x, x)
        x = self.layernorm1(x + self.dropout(attn_output))

        # Feed-forward network
        ffn_output = self.ffn(x)
        x = self.layernorm2(x + self.dropout(ffn_output))
        return x

class VisionTransformer(nn.Module):
    def __init__(self, image_size, patch_size, in_channels, embed_dim, num_heads, num_layers, ff_hidden_dim, num_classes):
        super(VisionTransformer, self).__init__()
        self.patch_embedding = PatchEmbedding(image_size, patch_size, in_channels, embed_dim)
        self.transformer_blocks = nn.ModuleList([
            TransformerBlock(embed_dim, num_heads, ff_hidden_dim) for _ in range(num_layers)
        ])
        self.pooling = nn.AdaptiveAvgPool1d(1)
        self.classifier = nn.Linear(embed_dim, num_classes)

    def forward(self, x):
        x = self.patch_embedding(x)  # (B, num_patches + 1, embed_dim)
```



```python
    def forward(self, x):
        x = self.patch_embedding(x)  # (B, num_patches + 1, embed_dim)

        for block in self.transformer_blocks:
            x = block(x)  # (B, num_patches + 1, embed_dim)

        x = x[:, 0]  # Take the CLS token (B, embed_dim)
        x = self.classifier(x)  # (B, num_classes)
        return x

# Example usage
if __name__ == "__main__":
    image_size = 224
    patch_size = 16
    in_channels = 3
    embed_dim = 768
    num_heads = 12
    num_layers = 12
    ff_hidden_dim = 3072
    num_classes = 1000

    model = VisionTransformer(image_size, patch_size, in_channels, embed_dim, num_heads, num_layers, ff_hidden_dim, num_classes)
    print(model)
```

OUTPUT



```
VisionTransformer(
  (patch_embedding): PatchEmbedding(
    (projection): Conv2d(3, 768, kernel_size=(16, 16), stride=(16, 16))
  )
  (transformer_blocks): ModuleList(
    (0-11): 12 x TransformerBlock(
      (attention): MultiheadAttention(
        (out_proj): NonDynamicallyQuantizableLinear(in_features=768, out_features=768, bias=True)
      )
      (ffn): Sequential(
        (0): Linear(in_features=768, out_features=3072, bias=True)
        (1): ReLU()
        (2): Linear(in_features=3072, out_features=768, bias=True)
      )
      (layernorm1): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
      (layernorm2): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
  )
  (pooling): AdaptiveAvgPool1d(output_size=1)
  (classifier): Linear(in_features=768, out_features=1000, bias=True)
)
```

Conclusion

Vision Transformers represent a groundbreaking approach to image processing, offering a fresh perspective on how we can analyse visual data. By leveraging the transformer architecture, they provide scalability, flexibility, and impressive performance across various tasks.

As you explore the world of Vision Transformers, consider experimenting with different datasets and model configurations to see how they perform. The implementation in PyTorch is just the beginning; the possibilities are vast.