Annotation of Vision Transformer

Aravind Balaji Srinivasan
Department of Computer Science
Illinois Institue of Technology
asrinivasan@hawk.iit.edu

Vignesh Ram Ramesh Kutti
Department of Computer Science
Illinois Institue of Technology
vrameshkutti@hawk.iit.edu

Abstract

Vision transformers have shown remarkable effectiveness for computer vision tasks. In this report, we examine our method of analyzing and annotating a vision transformer model to better understand its inner workings and decision-making strategies. By using a range of techniques, we investigate how the model processes visual information progressively through its layers. Our annotations show intriguing patterns in hierarchical feature learning, progressing from basic to complex concepts. Additionally, we examine the self-attention mechanisms, observing how distinct components specialize in capturing diverse visual relationships and contextual cues. This work seeks to deepen comprehension of vision transformers and set a basis for further enhancements. The results of this paper provide meaningful insights into the strengths and possible challenges of these models in computer vision.

1 Introduction

Vision transformers have quickly become a leading architecture in computer vision, excelling in areas like image classification, object detection, and segmentation. First introduced by Dosovitskiy et al. [2021], these models modify the transformer design used in language processing by dividing images into sequences of smaller patches.

2 Significance of the paper

The Key breakthrough of this paper is to demonstrate a far superior alternative to CNN models for computer vision tasks such as image classification. Vision Transformer (ViT) is able to get excellent results compared to state-of-the-art convolutional networks while requiring substantially fewer computational resources to train. This is particularly achieved through the attention mechanism. Best model version of VIT reaches the accuracy of 88.55% on ImageNet, 90.72% on ImageNet-ReaL, 94.55% on CIFAR-100, and 77.63 % on the VTAB suite of 19 tasks.

3 Our Objective

This report presents the analysis and annotation of a vision transformer model taking inspiration from Rush [2018], aiming to clarify its internal structure and decision-making strategies. Despite their notable achievements, many aspects of vision transformers' inner processes remain unclear. Through the annotations, we seek to reveal how these models interpret and handle visual data.

CS 577 Class Project, Illinois Institute of Technology.

Several techniques like attention visualization and feature map analysis are used to see how the model processes visual features, moving from simple patterns to more detailed ones. Studying self-attention also shows how different parts capture specific visual relationships. Our goal is to make vision transformers easier to understand, highlighting their strengths and areas to improve for better, clearer AI in visual tasks. In this paper, the authors have specifically employed the CIFAR10 dataset and created a smaller version of the base model.

4 Basic Transformer Architecture

To Understand Transformers, we need to delve into the concept of Attention. The paper 'Attention is all you need' Vaswani et al. [2017] discusses the concept of Self Attention. Below is a brief on the basic Transformer architecture that will help us understand the VIT Model.

What is **Attention** exactly? Attention in transformer focuses on specific parts of an input be it a sequence of words or a sequence of patches of an image. This mechanism assigns importance to various different parts of the input which enables the model to capture relationships between the input sequences and their context effectively. Transformers are models which use the self-attention techniques, to improve the ability to grasp long-range relationships, enable parallel computation, and increase the efficiency of processing sequences. They are made of two primary components: **Encoder** and **Decoder**.

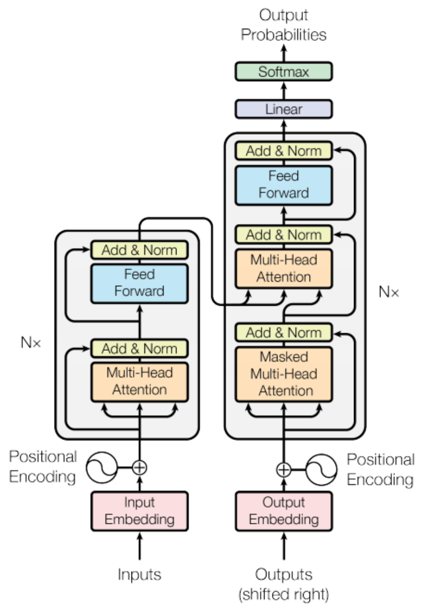

Figure 1: Basic Transformer Architecture showing encoder-decoder structure with multi-head attention mechanisms and Feed Forward Network

4.1 Encoder Stack

The Encoder stack is usually made of N identical layer with two major components :

- **Multi-Head Attention**: This component allows the model to look at each position in the input sequence and attend to all other positions. By doing this, the model can capture complex relationships between different parts of the input. The "multi-head" aspect means that the model processes different parts of the sequence in parallel, attending to multiple representation subspaces at once, which helps it learn different kinds of patterns simultaneously.

```
class MultiHeadAttention(nn.Module):
    def __init__(self, emb_size, num_heads=8, dropout=0.1):
        ''' Multi-Head Attention module'''
        super().__init__()
        self.num_heads = num_heads
        self.emb_size = emb_size
        self.qkv = nn.Linear(emb_size, emb_size * 3)
        self.attn_dropout = nn.Dropout(dropout)
        self.projection = nn.Linear(emb_size, emb_size)
        self.attention_weights = None

    def forward(self, x):
        batch_size, tokens, emb_size = x.shape
        qkv = self.qkv(x).reshape(batch_size,
              tokens, 3, self.num_heads, emb_size // self.num_heads)
        q, k, v = qkv.permute(2, 0, 3, 1, 4)
        attn = (q @ k.transpose(-2, -1)) * (emb_size ** -0.5)
        attn = attn.softmax(dim=-1)
        self.attention_weights = attn
        attn = self.attn_dropout(attn)
        x = (attn @ v).transpose(1, 2).reshape(batch_size, tokens, emb_size)
        return self.projection(x)
```

- **Feed Forward Network**: After the attention mechanism, each position's output is passed through a fully connected feed-forward network. This network processes the output independently for each position in the sequence, helping the model to refine the representations further. The FFN is defined within the TransformerEncoderLayer Class which initializes all the sub-components of the Encoder Stack.

```
class TransformerEncoderLayer(nn.Module):
    def __init__(self, emb_size, num_heads, forward_expansion, dropout):
        super().__init__()
        self.layernorm1 = nn.LayerNorm(emb_size)
        self.mha = MultiHeadAttention(emb_size, num_heads)
        self.dropout1 = nn.Dropout(dropout) # Dropout
        self.layernorm2 = nn.LayerNorm(emb_size)
        # Feed Forward Network for classification part
        self.feed_forward = nn.Sequential(
            nn.Linear(emb_size, forward_expansion * emb_size),
            nn.GELU(),
            nn.Linear(forward_expansion * emb_size, emb_size),
            nn.Dropout(dropout)
        )

    def forward(self, x):
        x = x + self.dropout1(self.mha(self.layernorm1(x)))
        x = x + self.feed_forward(self.layernorm2(x))
        return x
```

4.2 Decoder Stack

- **Masked Multi-Head Attention**: This layer takes the previous inputs into account while processing attention but leaves out future positions this is called masking. Before applying softmax function we will initialize all future input positions with $-\infty$ so that after applying softmax function they become 0, without disturbing the probability distribution.
- The Encoder and Decoder layers are connected to each other. This enables the decoder in learning to attend to specific positions in the input sequence that are most relevant for generating the current output token.

However, for the Vision Transformer architecture, This componenet is conveniently dropped as we will see in section 1.4.

4.3 How to Calculate Attention?

Attention is calculated using formula:

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

Keys (K), Queries(Q), Values(V): Let X be the input matrix containing the input values. $X \in \mathbb{R}^{T\times d}$. Where T is the length of the sequence be it tokens or patches, d is Dimensionality of embeddings. $Q = XW_Q, \quad K = XW_K, \quad V = XW_V$, where W_Q, W_K, W_V are learnable weight matrices, they get updated after each iteration. $Q, K, V \in \mathbb{R}^{T\times d_k}$ where $d_k = \frac{d}{h}$ where h is the number of attention heads. Let us look at a sample calculation:

The input sequence X has shape (4×512) (4 tokens, 512-dimensional embeddings).

The weight matrices W_Q, W_K, W_V have shape (512×64), where $d_k = 64$.

1. Compute Q: $Q = XW_Q$ Resulting Q has shape (4×64).
2. Similarly, compute K and V: $K = XW_K, \quad V = XW_V$
   Both K and V will also have shape (4×64).
3. Use Q, K, V to calculate attention: $\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$

5 Vision Transformer

Now let us look at Vision Transformers (ViT). ViTs apply the same concept of transformers but for images. The image is divided into patches and these patches are flattened and fed as input to the Encoder only transformer. The transformer output is passed to a MLP head that performs the classification part.

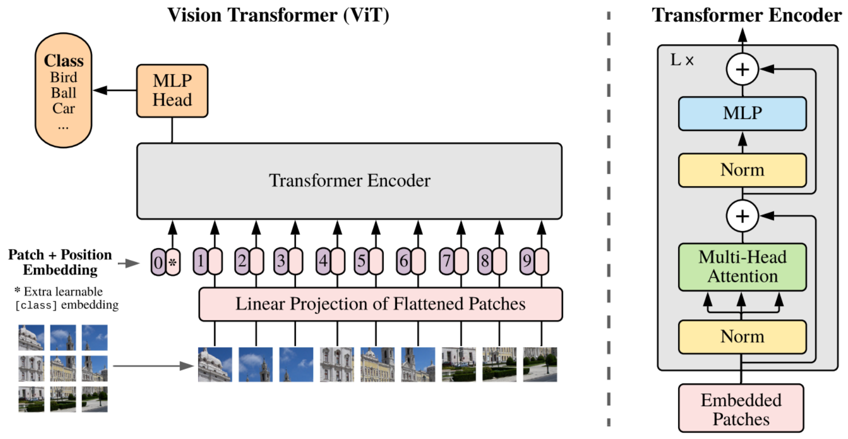

Figure 2: Architecture of Vision Transformer. Overview of the model where it has Patch and Positional Embeddings. There is an extra "classification token" which is essential to perfom classification.

5.0.1 Vision Transformer Architecture

The input to the model is patches of an image. The patches must not overlap. The patches are flattened before feeding to the transformer layer forming a vector. Here are the key components

- **Patch Embeddings:** The vector which is a result of flattening is then linearly projected into a feature space of dimension d. If the input has size $H \times W$ and patch size $P \times P$, number of patches is given by: $N = \frac{H \times W}{P^2}$

```
class PatchEmbedding(nn.Module):

    def __init__(self, in_channels=3, patch_size=4, emb_size=64, img_size=32):
        super().__init__()
        self.patch_size = patch_size
        self.projection = nn.Conv2d(in_channels, emb_size,
           kernel_size=patch_size, stride=patch_size)
        self.cls_token = nn.Parameter(torch.randn(1, 1, emb_size))
        self.positional_embedding = nn.Parameter(torch.randn(1,
          (img_size // patch_size) ** 2 + 1, emb_size))

    def forward(self, x):
        x = self.projection(x)
        x = x.flatten(2).transpose(1, 2)
        batch_size = x.shape[0]
        cls_tokens = self.cls_token.expand(batch_size, -1, -1)
        x = torch.cat((cls_tokens, x), dim=1)
        x += self.positional_embedding
        return x
```

- **Positional Encoding:** To keep a record of the order of sequence of patches we include positional encoding are added to patch embeddings to encode spatial information.
- **Classification Token:** A classification token is added to the patch embeddings. This token is learnable as model trains. After the model finishes training, this token represents the class of the image.
- **Classification Head** is the output of class token when it is processed through a fully-connected layer for classification.
- **Transformer Encoder:** Similar to basic transformer, the sequence of patches and class tokens are passed through a Transformer encoder. This encoder also contains the Multi-head self attention, which captures the relation between patches globally and the Feed-Forward Network. The normalization layer and residual (skip) connections are applied as well.

```
class VisionTransformer(nn.Module):
    def __init__(self, in_channels=3, patch_size=4, emb_size=64, img_size=32,
    num_classes=10, depth=7, num_heads=8, forward_expansion=4, dropout=0.0):
        super().__init__()
        self.patch_embedding = PatchEmbedding(in_channels,
        patch_size, emb_size, img_size)
        # Transformer Encoder Layers
        self.transformer_encoders = nn.Sequential(
        *[TransformerEncoderLayer(emb_size, num_heads,
             forward_expansion, dropout) for _ in range(depth)])
        self.layernorm = nn.LayerNorm(emb_size)
        self.mlp_head = nn.Linear(emb_size, num_classes)

    def forward(self, x):
        x = self.patch_embedding(x)
        x = self.transformer_encoders(x)
        x = self.layernorm(x[:, 0])
        return self.mlp_head(x)
```

5.1 Comparison of basic Transformer and Vision Transformer

Aspect	ViT (Vision Transformer)	Original Transformer (NLP)
Input Representation	Splits images into patches and treats them as tokens.	Processes text tokens (words or subwords).
Token Length	Number of patches depends on image size and patch size.	Sequence length depends on text length.
Positional Embedding	Encodes spatial layout of image patches.	Encodes sequential order of text tokens.
Classification Token	Used for image classification tasks.	Used for tasks like text classification.
Inductive Biases	Minimal (e.g., no convolutional filters).	Exploits sequential nature of text.
Data Requirements	Requires large-scale datasets for effective training (e.g., JFT-300M).	Can perform well on smaller datasets.
Attention Scope	Global self-attention for patches.	Global self-attention for text tokens.

Table 1: Comparison of Vision Transformer (ViT) and Original Transformer.

5.2 Parameters for ViT

Parameters required for implementing ViTs are

- **Input Parameters:**
  - `image_size` = size of input image $(H \times W)$.
  - `patch_size` = number of Patches calculated using formula for$N$
- **Embedding Parameters:**
  - `embedding_dim` dimension of each patch vector represented by d.
  - `positional_embedding`: this parameter assists performing positional encoding. This has the same dimension as `embedding_dim`.
- **Transformer Encoder Parameters:**
  - `num_layers`: number of layers of the Transformer Encoder. This is the depth of the model and depth $\propto$ performance.
  - `num_heads`: number of attention heads given by $\frac{\texttt{embedding\_dim}}{head\_dim}$.
  - `mlp_dim`: dimensions of feed-forward network in each encoder layer. Increasing this more than `embedding_dim` can capture better representations.
- **Classification Parameters:**
  - `num_classes`: number of output classes for classification. this determines the size of output layer in the classification head.
  - `cls_token`: this is a learnable parameter which is appended in front of the sequence of patches.
- **Dropout and Regularization:**
  - `dropout_rate`: This rate determines the amount of neurons to randomly exclude while processing. This will help in reducing overfitting the model.
- **Optimization Parameters:**
  - `learning_rate`: This rate determines the step size for the optimizer during training. This is set by trial and error method.
  - `batch_size`: number of images processed together during training. higher batch size requires more computational power.
  - `epochs`: number of times the entire dataset is passed through the model.

The parameters we used for the Minimal CPU Working model we developed is:

```
"image_size": batch_size x 3 x 32 x 32,
"patch_size": 64 ,
"embedding_dim": 256,
"num_layers": 12,  %correct value we need here.
"num_heads": 12,
"mlp_dim": 3072,
"num_classes": 10,
"dropout_rate": 0.1,
```

6 Training and Evaluation

Section 1.5 discusses in length, how the training and evaluation setup was implemented to closely mimic the original paper's version and making it simple enough to run on a single CPU.

6.1 Dataset

The dataset used for the project is CIFAR-10Krizhevsky [2009]. It is relatively small consisting of of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images. However, for the purpose of this project the authors have used a train subset of 600 images and a test subset of 200 images only. This ensures the training time is significantly lower and runnable in a CPU. The code snippet below handles the data-loaders that essentially download the dataset, takes a subset, performs normalization and loads it into Pytorch data-loaders. This is then directly passed to the training loop.

```
def get_data_loaders(batch_size=128, train_subset_size=600,
 test_subset_size=200):
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])
    trainset = torchvision.datasets.CIFAR10(root='./data',
     train=True,download=True, transform=transform)
    train_subset = Subset(trainset, torch.arange(train_subset_size))
    trainloader = DataLoader(train_subset, batch_size=batch_size,
    shuffle=True, num_workers=2)

    testset = torchvision.datasets.CIFAR10(root='./data', train=False,
    download=True,
    transform=transform)
    test_subset = Subset(testset, torch.arange(test_subset_size))
    testloader = DataLoader(test_subset, batch_size=batch_size,
    shuffle=False, num_workers=2)

    return trainloader, testloader
train_loader, test_loader = get_data_loaders()
```

6.2 Training Loop

The following train function incorporates much of the mechanism behind training this large transformer architecture. Due to hardware limitations, the model is trained for **6 epochs** only. The train loop iterates over images and corresponding labels from the train-set and passes it to the model. It finally performs back-propagation with the pre defined loss and optimizer in place. The metrics are promptly calculated and returned as a list.

```
def train(model, dataloader, criterion, optimizer):
    model.train()
    total_loss, correct = 0, 0
    # Reset metric values
    accuracy_metric.reset()
    auc_metric.reset()
    f1_metric.reset()
    precision_metric.reset()
    recall_metric.reset()
    top1_accuracy_metric.reset()
    top5_accuracy_metric.reset()

    for images, labels in dataloader: # Loop through the batches
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        loss = criterion(outputs, labels)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_loss += loss.item()
        preds = outputs.argmax(1)
        probs = torch.softmax(outputs, dim=1)

        # Update metrics
        accuracy_metric.update(preds, labels)
        auc_metric.update(probs, labels)
        f1_metric.update(preds, labels)
        precision_metric.update(preds, labels)
        recall_metric.update(preds, labels)
        top1_accuracy_metric.update(outputs, labels)
        top5_accuracy_metric.update(outputs, labels)

    avg_loss = total_loss / len(dataloader)
    train_accuracy = accuracy_metric.compute()
    train_auc = auc_metric.compute()
    train_f1 = f1_metric.compute()
    train_precision = precision_metric.compute()
    train_recall = recall_metric.compute()
    train_top1_accuracy = top1_accuracy_metric.compute().item()
    train_top5_accuracy = top5_accuracy_metric.compute().item()

    return avg_loss, train_accuracy.item(), train_auc.item(),
    train_f1.item(), train_precision.item(), train_recall.item(),
    train_top1_accuracy, train_top5_accuracy
```

When the model is set to training mode, it initializes the required metrics such as accuracy, F1- score, AUC etc. At the beginning of each epoch, these values are reset to calculate values for that epoch. After loading the data, they are converted into 5 mini-batches of size 128 to optimize the memory usage and computation efficiency. Then the batches containing the images and labels are designated with either CPU or GPU, in our case GPU. The images are passed through the model to generate predictions, and the **Cross-Entropy Loss** is calculated to measure how far the predictions are from the actual labels.

The optimization process is handled by the **Adam optimizer**, a method that adjusts learning rates for each parameter and incorporates momentum to speed up convergence. Before gradients are calculated, the optimizer clears any stored values from previous batches. The model's gradients are then computed by back-propagating the loss through the network, and the optimizer updates the model's parameters based on these gradients. A weight decay of 0.1 is introduced as L2 regularization to minimize the overfitting as the discourage the large weights. Dropout is also used to deactivate certain neuron over the course of training.

During training, the model's performance is assessed by updating metrics as each batch is processed. Performance measures are updated continually during the training process to provide a complete perspective of the progress, and predictions are finally computed by taking the softmax of the logits generated by the final layer and taking the highest probability. This guarantees effective and efficient learning for the model.

6.3 Evaluation

The evaluation component is called after training, with back-propagation turned off. The evaluate function calculates a few metrics namely: accuracy, F1 score, Precision and Recall, Top1 and Top5 accuracy, the training and validation loss.

```
def evaluate(model, dataloader, criterion):
    ''' evaluate the model on the test set '''
    model.eval()
    total_loss, correct = 0, 0

    # Reset metric values
    accuracy_metric.reset()
    auc_metric.reset()
    f1_metric.reset()
    precision_metric.reset()
    recall_metric.reset()
    top1_accuracy_metric.reset()
    top5_accuracy_metric.reset()
    img_size, patch_size = 32, 4
    with torch.no_grad():
        for images, labels in dataloader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)

            total_loss += loss.item()
            preds = outputs.argmax(1)
            probs = torch.softmax(outputs, dim=1)  # Probabilities for AUC

            # Update metrics
            accuracy_metric.update(preds, labels)
            auc_metric.update(probs, labels)
            f1_metric.update(preds, labels)
            precision_metric.update(preds, labels)
            recall_metric.update(preds, labels)
            top1_accuracy_metric.update(outputs, labels)
            top5_accuracy_metric.update(outputs, labels)

    # Get attention weights from the last transformer encoder layer
    last_layer = model.transformer_encoders[-1]
    attention_weights = last_layer.mha.attention_weights[0]

    # Average attention weights across heads
    attention_weights = attention_weights.mean(dim=0)

    # Get attention weights for CLS token
    cls_attention = attention_weights[0, 1:]  # Skip CLS token

    # Reshape attention weights into a square grid
    num_patches = (img_size // patch_size) ** 2
    attention_grid = cls_attention.reshape(img_size // patch_size,
    img_size // patch_size)
```

```
attention_grid = (attention_grid - attention_grid.min()) /
(attention_grid.max() - attention_grid.min())

# Visualize attention weights from last encoder layer
fig = attn_viz(images[0], attention_grid)
plt.suptitle(f'Attention Visualization (Class: {labels[0]})')
plt.show()

avg_loss = total_loss / len(dataloader)
val_accuracy = accuracy_metric.compute()
val_auc = auc_metric.compute()
val_f1 = f1_metric.compute()
val_precision = precision_metric.compute()
val_recall = recall_metric.compute()
val_top1_accuracy = top1_accuracy_metric.compute().item()
val_top5_accuracy = top5_accuracy_metric.compute().item()

return avg_loss, val_accuracy.item(), val_auc.item(),
val_f1.item(), val_precision.item(),
val_recall.item(), val_top1_accuracy, val_top5_accuracy
```

When the evaluation function is called the model turns goes into the evolution mode where it disables the dropout so that it can produce stable predictions. The performance metrics are again put into reset and calculated for the the current batch.The test data comes from the mini-batch as well, but the ones that aren't used are sent to the CPU or GPU (in our instance, the GPU) for processing. After that, the function tracks its progress by updating the metrics according to the actual labels.

As the data is processed, the function computes predictions by applying the softmax function and continuously updates performance metrics such as precision, recall, and F1-score for each batch. The loss is accumulated over the course of the evaluation. Additionally, the function retrieves attention weights from the final transformer encoder layer, which show which areas of the image the model focuses on during prediction. These attention weights are averaged across all attention heads and then reshaped into a grid that matches the image's patch layout.

Normalizing and overlaying the attention grid highlights the areas of the image that have the most impact on the model's decisions helping us in understanding the model's processes for making a predictions.This visualization could be seen in Figure 3. After all batches are processed, the function calculates the average loss and gathers all the performance metrics, offering a clear picture of how well the model is performing. The evaluation process not only predict but also helps us in visualize the key insights of data interpretation.

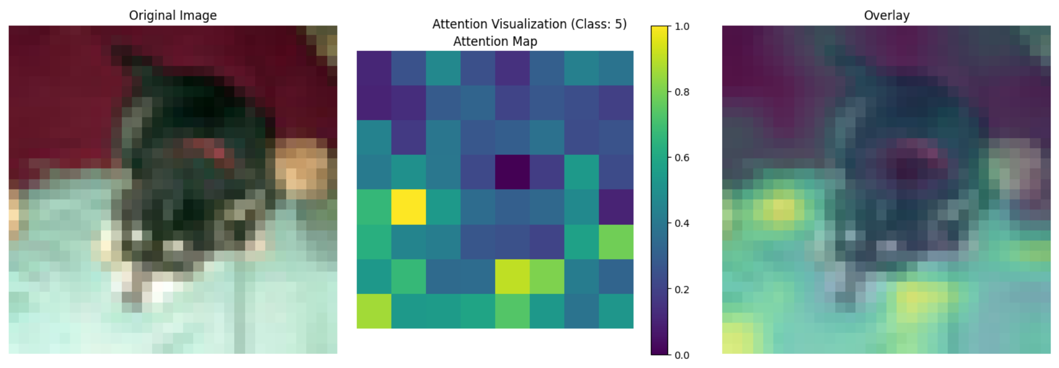

Figure 3: shows the attention matrix for a sample image during the training of the last layer on the last epoch. We can notice how the brighter scores are more concentrated towards the center of the image where the animal's head is present proving the effectiveness of the system.

7 Results and Discussion

Table 2 represents the results after training for 6 epochs. It is evident the accuracy is not up to the mark, especially in comparison to the benchmark of the base model. Key factors behind this performance are the incredibly small subset of the dataset used and the few epochs of training. We can see that despite the performance, the metrics in comparison to each other show consistency and no visible bias. Thus proving the implementation is correct but the model severely underfit which is to be expected.

Metric	Train	Validation
Loss	2.0789	2.1240
Accuracy	0.2027	0.2051
AUC	0.6847	0.6567
F1	0.140	0.1545
Precision	0.1402	0.1337
Recall	0.2027	0.2051
Top-1 Accuracy	0.2027	0.2051
Top-5 Accuracy	0.7253	0.6604

Table 2: ViT Results after 6 epochs

The images in Figure 4 visually demonstrated how the metrics change over time through out the 6 epochs trained.

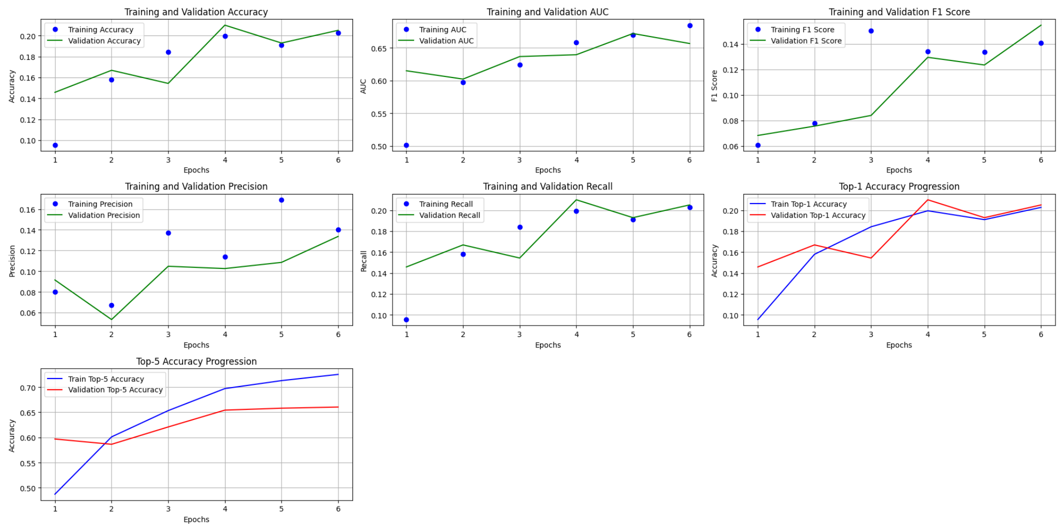

Figure 4: Evaluation Results over 6 epochs

8 Weaknesses and Limitations

- ViTs require massive amounts of data to perform well and to provide competitive accuracy with CNNs. We can mitigate this by pre-training ViT on ImageNet and fine tune it for smaller tasks.
- ViTs are computationally expensive because complexity of self-attention mechanism has $O(n^2)$ (n = num of patches) complexity which means the complexity increases quadratically

as number of patches increases. To mitigate this we can adopt a hybrid architecture with CNNs for feature extraction.

- ViTs don't come with a sense of sequence order of patches(inductive bias). We can add positional embeddings to include spatial information.
- ViTs will overfit small datasets which is probable in our minimal CPU ready working example. So we employed methods like Dropout and weight decay as Regularization methods to battle overfitting.
- Since ViTs require high computational power and lots of memory, their application to low-resource devices are limited.

9 Potential upgrades

1. **Improved Attention Mechanisms**: The attention mechanism could be changed to spares or multi-scale attention so that the model captures more features which helps in improving its efficiency.
2. **Enhanced Patch Embedding**: Some of the drawbacks of ViT's reliance on non-local patch embeddings can be addressed by using convolutional layers or hybrid models that mix CNNs with transformers to improve the model's processing of local visual information.
3. **Regularization Techniques**:The generalization capabilities of the model can be enhanced by avoiding overfitting, particularly when working with smaller datasets, by including techniques like dropout, stochastic depth, or other sophisticated regularization algorithms.
4. **Self-Supervised Learning**:Integrating self-supervised learning techniques, as seen in DeiT, can improve the model's performance by learning useful representations without requiring large amounts of labeled data.
5. **Pretrained Models**:Leveraging pretrained models from large-scale datasets or transfer learning can speed up training, improve model accuracy, and reduce the need for extensive labeled data, following the approach of DeiT's data-efficient training strategy.

10 Conclusion

The analysis and annotation of the Vision Transformer paper provided a deeper understanding of ViTs and their working with code blocks and their explanation. We delved deep into how transformers work, how attention is incorporated in to their working. The minimal CPU ready example and the visualizations on where the model is focusing on provides a clear picture on how ViTs interact with the data. By combining theoretical explanations, practical implementations and visualizations, this annotation study demonstrates the power and capabilities of Vision transformers and their significance in the future of Computer Vision.

During training and evaluation the model is tested on many metrics like F1-score, precision, recall, top-1 accuracy, and top-5 accuracy, which provides a robust analysis of the model's performance. Methods like Dropout, weight_decay (Regularization) have been used to reduce overfitting in our code. This helps us demonstrate a good model with limited data size.

Vision transformers outperform CNNs in tasks like image classification, object detection and segmentation. The ability to capture global attention make it to capture complex relationships within an image. ViTs can be scaled up or down easily to fine tune them for various tasks. Huge amount of research is being done on ViTs which further pushes them to new heights. Their versatility and effectiveness makes them unmatched. Although ViTs cannot fully replace CNNs, they have solidified their role as a cornerstone of modern AI.

References

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2021. Published as a conference paper at ICLR 2021.

Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

Alexander Matt Rush. The annotated transformer. *The Annotated Transformer*, 2018. URL `https://nlp.seas.harvard.edu/annotated-transformer/`. A step-by-step explanation of the Transformer model.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 5998–6008, 2017. URL `https://arxiv.org/abs/1706.03762`.