EE583 Term Project
Implementation of Video Vision Transformer

Eda Özkaynar
Department of Electrical and Electronics Engineering, METU, Ankara, Turkey
eda.ozkaynar@metu.edu.tr

***Abstract*—Transformer models are powerful tools that can capture long-range relationships and changes over time. They are useful in different fields, including natural language processing, computer vision, and video analysis. In this project, I implemented the Vision Video Transformer (ViViT), a state-of-the-art approach for video classification tasks, and evaluated its performance on the OrganMNIST-3D dataset. The goal is to assess the model's capabilities in extracting spatio-temporal features and compare it with conventional methods. I conduct a series of experiments to analyze the model's generalization capabilities. Results demonstrate the effectiveness of ViViT in handling 3D data while highlighting potential areas for improvement, with the best Top-1 accuracy of 88.36% and Top-5 accuracy of 99.84%. The source code is available on https://github.com/eozkaynar/EE-583-Term-Project**

***Index Terms*—Vision Video Transformer (ViViT), Video classification, OrganMNIST-3D dataset, Spatio-temporal features, Model robustness, Scalability, Generalization, Top-1 accuracy, Top-5 accuracy, 3D data analysis**

I. Introduction

This paper investigates the use of the Vision Video Transformer the Vision Video Transformer (ViViT) for video classification, focusing on the OrganMNIST-3D dataset. This dataset is part of the MedMNIST benchmark for classifying 3D medical images. I implemented ViViT Model-1 to see how well it captures features from 3D medical data over time and space.

The paper starts with an overview of ViViT's structure. It explains how ViViT breaks down video inputs into manageable parts using tubelet embedding, adds positional encodings, and processes them with transformer layers.

In the methodology section, I describe the OrganMNIST-3D data and detail the implementation, including choices about patch size, projection dimension, and the number of attention heads.

In the experiments section, we compare ViViT to a 3D CNN. We highlight the differences in training, performance, and efficiency. The results show that ViViT achieves high accuracy but needs a lot of computing power and careful adjustments to work well with small datasets like OrganMNIST-3D.

Finally, the conclusion discusses ViViT's strengths, like its ability to generalize and scale for video classification. It also mentions limitations, such as slower training and higher computing needs compared to 3D CNNs, and offers suggestions for future research to improve efficiency and adapt the model for larger datasets.

II. Model

In this project, I used ViViT Model-1 [1]. This model breaks down video samples into small segments called tubelets, which capture both space and time information. These tubelets are turned into features using a patch embedding layer, combined with positional information, and then processed by a standard transformer encoder.

A. Attention is All You Need: An Overview

1) Attention

Attention is a mechanism that enables each part of an input sequence, such as the words in a sentence, to focus on other parts of the sequence when making predictions or generating output [2]. It allows the model to "look around" at other words to better understand the meaning of each word within its context.

Query (Q): The Query represents what a specific word or token in a sentence is seeking to focus on. It acts like a "question" asking, "What other words are relevant to me?" For instance, when translating "apple," it might look for related concepts like "fruit" or "eating."

Keys (K): The Key signifies potential matches for the Query. Each word or token has a Key that describes its content. When calculating attention, each Query is compared to Keys to determine relevance. For example, the Query about "apple" might match closely with Keys for "fruit" or "healthy."

Values (V): The Value contains the actual information used to interpret the Query. After matching Queries to Keys, the Values are weighted and combined based on their relevance.

Match Query and Key: Each Query is compared to each Key by calculating a score (usually a dot product) that measures how closely they match.

Calculate Attention Weights: These scores are then converted into attention weights using a softmax function, so the scores add up to 1. This tells the model which Values to focus on more.

Weighted Sum of Values: Finally, the Values are combined according to these weights to produce a final output that considers the most relevant information in the sequence.

2) Scaled Dot Product Attention

In the Transformer model, the attention function used is called Scaled Dot-Product Attention [2]. It takes a set of Queries (Q),

Keys (K), and Values (V) as input. The process is described as follows:

1. Similarity Calculation: The model calculates the dot product between each Query (Q) and each Key (K) to determine their similarity:

$$\text{Score}(Q, K) = QK^T$$

2. Scaling: To prevent the dot product values from becoming excessively large due to high dimensions, the results are scaled by dividing by the square root of the dimension of the Keys ($\sqrt{d_k}$):

$$\text{Scaled Scores} = \frac{QK^T}{\sqrt{d_k}}$$

3. SoftMax Application: A SoftMax function is applied to convert these scaled values into attention weights, which indicate the level of focus that should be given to each Value:

$$\text{Attention Weights} = \text{SoftMax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$$

4. Weighted Sum: The model produces an output by taking a weighted sum of the Values (V), using the calculated attention weights as weights:

$$\text{Output} = \text{Attention Weights} \cdot V$$

This process allows the model to concentrate on the most relevant parts of the input sequence when generating an output.

Multi-Head Attention

The Transformer splits the attention mechanism into multiple heads [2]. This process is described as follows:

1. Projection: The Queries (Q), Keys (K), and Values (V) are each linearly projected multiple times (one for each head) into smaller dimensions:

$$Q_i = QW_i^Q, \quad K_i = KW_i^K, \quad V_i = VW_i^V$$

Each of these projections is done using a learned linear transformation, producing d_k dimensions for Keys and Queries, and d_v dimensions for Values.

2. Parallel Attention Heads: Each of these projections (called "heads") then performs the attention function independently and in parallel on its own set of Queries, Keys, and Values:

$$\text{Attention Head}_i = \text{SoftMax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) V_i$$

This allows each head to focus on different parts of the sequence or capture different types of relationships.

3. Concatenation: The outputs from each of these attention heads (each with d_v dimensions) are concatenated back together into a single vector:

$$\text{Concatenated Output} = \text{Concat}(\text{Head}_1, \text{Head}_2, \ldots, \text{Head}_h)$$

where h is the number of heads.

4. Final Projection: This combined output is then projected again using another learned linear transformation to match the model's full dimension size:

$$\text{Output} = \text{Concatenated Output}W^O$$

This multi-head mechanism enables the model to capture diverse relationships and attend to multiple aspects of the input sequence simultaneously.

3) Positional Encoding

Positional Encoding is designed to help the Transformer model understand the order of words in a sequence [2]. Since the Transformer model doesn't use recurrent (RNN) or convolutional (CNN) structures, it cannot naturally learn the sequential order of tokens. Therefore, we need to add positional information so the model can process data according to the sequence order. This position information is added to each word's embedding, so the model gains a sense of order.

1) **Matching Dimensions**: The positional encoding vectors are set to the same size as the word embedding vectors (d_{model}), allowing them to be easily summed.
2) **Sine and Cosine Functions**: The positional encoding vectors are created using sine and cosine functions at different frequencies.
   Position information is calculated with different sine or cosine values for each dimension:
   - For even dimensions:

$$PE(\text{pos}, 2i) = \sin\left(\frac{\text{pos}}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$

   - For odd dimensions:

$$PE(\text{pos}, 2i+1) = \cos\left(\frac{\text{pos}}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$

This approach assigns different wavelengths to each position, helping the model understand the relative distance between positions.

4) Encoder

Input Embedding

Each word or token is represented as a vector (e.g., 512 dimensions) using a learned embedding matrix in the model. These vectors capture the meanings and contexts of the words. After this step, each word or token becomes a fixed-size vector.

Positional Encoding

In the Transformer architecture, the order of words isn't inherently understood. To add information about the order, positional encoding is added to each word's embedding vector. These combined vectors are then given as input to the encoder block.

Multi-Head Attention

Multi-head attention enables the model to focus on different parts of a sentence simultaneously. Multi-head attention captures multiple relationships within a sentence by allowing different heads to focus on various aspects. This enriches each

word's representation with both immediate and distant connections. It also enables direct relationships between words, making it easier to learn long-distance dependencies without extensive processing.

Residual Connections

In the Transformer model, each sub-layer (like multi-head attention and feed-forward networks) uses residual connections and layer normalization to stabilize the network and improve gradient flow, enabling more effective training of deeper models. Residual connections allow gradients to flow directly through the network by bypassing the non-linear transformations, which mitigates the vanishing gradient problem and helps prevent overfitting.

$$\text{Output} = x + \text{Sublayer}(x)$$

Layer Normalization

After the residual connection, layer normalization is applied. Layer normalization normalizes the output across the feature dimensions (each dimension of the vector) for each token separately, rather than across the batch. This helps stabilize and accelerate training by ensuring that the scale of each layer's output remains consistent.

$$\text{LayerNorm}(x) = \frac{x - \mu}{\sigma} \cdot \gamma + \beta$$

Position-Wise Feed-Forward Network

After the multi-head attention layer in the encoder and decoder, each token's representation is processed by a Position-Wise Feed-Forward Network (FFN). This network operates independently on each token without sharing weights, meaning that each token is processed individually, regardless of its position among others.

The FFN layer consists of two linear transformations with a ReLU activation function in between:

$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

where:

- x: input vector, $d_{ff} = 2048$
- $W_1 \in \mathbb{R}^{d_{\text{model}} \times d_{ff}}$ and $b_1 \in \mathbb{R}^{d_{ff}}$ are the weight matrix and bias for the first linear layer.
- $W_2 \in \mathbb{R}^{d_{ff} \times d_{\text{model}}}$ and $b_2 \in \mathbb{R}^{d_{\text{model}}}$ are the weight matrix and bias for the second linear layer.

Expanding the dimensionality from d_{model} to d_{ff} in the feed-forward network (FFN) layer allows the model to capture complex patterns through richer, non-linear transformations. Reducing it back to d_{model} ensures efficiency and maintains consistent output dimensions across layers. This strategy balances expressive power and computational efficiency.

5) Decoder

Definition of Auto-Regressive Model

In an *auto-regressive model*, each token in a sequence is predicted based on the previously generated tokens in the sequence. This means that when generating a sequence, the model generates one token at a time and uses each generated token to help predict the next one. The model doesn't "look ahead" or have access to future tokens during generation.

Linear Layer & Prediction

The *linear layer* is a fully connected layer that takes the final hidden state of the decoder (from the previous layers) as input. It applies a learned transformation to map the decoder's hidden state to a vector with a size equal to the vocabulary size. This means each element in this vector corresponds to a score for a word in the vocabulary. The result is a raw score (or logits) for each possible word.

After applying *softmax*, the result is a probability distribution over the vocabulary, indicating the likelihood of each word being the next in the sequence. The model typically selects the word with the highest probability or uses techniques like beam search to improve sequence quality.

Masked Multi-Head Attention

In the decoder, masking is employed to ensure that the model functions *auto-regressively*. Specifically, at each step, the model can only "see" or attend to the tokens before the current token in the output sequence. *Mask* is applied to set future positions to a large negative value, so that they don't contribute to the *softmax* results.

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

The Encoder-Decoder Multi-Head Attention Layer

It allows the decoder to access the encoder's output. While generating each token in the output, the decoder can examine all tokens in the input sequence to identify the most relevant ones.

Importance of Encoder-Decoder Multi-Head Attention

When generating each word in the output sequence, the model often needs to reference specific words or phrases in the input sequence. For example, in a translation task, the model needs to understand which words in the input sentence are most relevant to the word it is currently translating. Encoder-decoder attention enables this by letting the decoder "focus" on important parts of the input while generating each word in the output.

Mechanism of Encoder-Decoder Multi-Head Attention

- The encoder processes an input sequence through multiple layers, producing hidden states for each token. These states capture the relationships between tokens, providing context for the input.
- In the decoder, the decoder hidden states up to a certain point are used to create queries (Q), indicating what information is needed for the current output.
- The decoder queries (Q) interact with keys (K) and values (V) from the encoder's output. This process allows each query to focus on relevant input tokens.
- Multiple sets of Q, K, and V are used in parallel to capture different aspects of the input-output relationship.
- For each attention head, scores are calculated using the dot product between queries and keys, followed by *softmax* normalization. These scores determine the amount

of attention each input token receives for generating the output.
- The outputs from all attention heads are concatenated and transformed to create the final output of the encoder-decoder attention layer, which is then passed to the next layer in the decoder.

Decoder Input

During training, the decoder uses the target sequence as input. This target sequence is shifted to the right by one position. Shifting right means each token is moved one position later, with a special start token `[START]` or `[BOS]` (beginning of sequence) added at the beginning. This shifted target sequence is fed to the decoder as input, with the goal of predicting the next token in the sequence at each step. Output embedding and positional encoding steps are the same as for the encoder.

B. Vision Transformers (ViT): An Overview

Vision Transformer (ViT) [3] processes 2D images using the transformer architecture. It divides the images into N patches and converts them into tokens. The sequence of tokens is structured as:

$$z = [z_{\text{cls}}; Ex_1; Ex_2; \ldots; Ex_N] + p$$

Here:

- $\mathbf{x}$: Represents individual image patches that are extracted from the original 2D image. Each patch x_i is a small portion of the image (e.g., a square region).
- $\mathbf{E}$: Represents the linear projection applied to each patch. It transforms the 2D patch into a 1D feature vector or token.
- z_{cls} is an optional learned classification token prepended to the sequence.
- p is a learned positional embedding added to spatial information.

The tokens are then processed through an encoder composed of L transformer layers, where each layer includes:

- **Multi-Headed Self-Attention (MSA):** Captures relationships between tokens.
- **Layer Normalization (LN):** Stabilizes the training process.
- **MLP (Multilayer Perceptron):** Processes the token representations.

The process for each layer is as follows:

1) Apply self-attention and add the result to the input tokens.
2) Use an MLP on the updated tokens and add the result again.

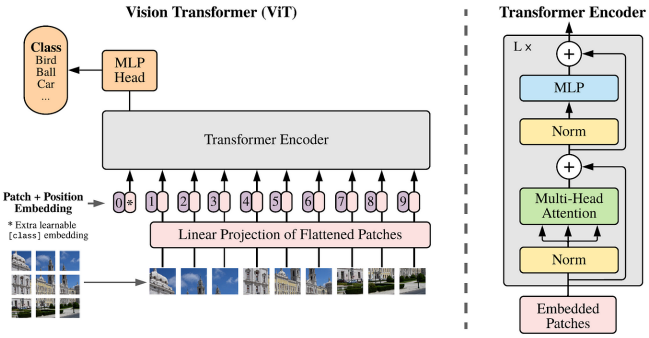

Fig. 1: Vision Transformer model overview

C. Tubelet Embedding

Tubelet Embedding captures temporal information from videos differently. First, it extracts volumes of video that include both frame patches and temporal details. These volumes are then flattened to create video tokens.

For a tubelet with dimensions $t \times h \times w$:

- $n_t = \frac{T}{t}$: Tokens are taken from the temporal dimension.
- $n_h = \frac{H}{h}$: Tokens are taken from the height.
- $n_w = \frac{W}{w}$: Tokens are taken from the width.

Using smaller tubelets creates more tokens, which increases the computational cost. This method combines spatial and temporal information during tokenization, unlike "Uniform Frame Sampling," where temporal information from different frames is combined later by the transformer [1] .

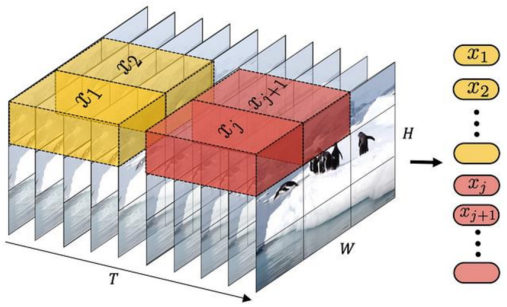

Fig. 2: Tubelet Embedding

D. Transformer Model: Spatio-Temporal Attention

A video sample is first tokenized using a Tubelet embedding-based approach, with each Tubelet treated as a token. These tokens are then passed through a patch embedding layer, with positional encoding added. All tokens are subsequently processed through a standard transformer encoder.

In the original model, in addition to the Tubelet tokens, a learnable parameter known as the CLS token is included and passed through the transformer. The output from the encoder is the CLS token, which is then fed into a multi-layer perceptron (MLP), a simple feed-forward network. Finally, a softmax activation is applied to produce a probability distribution of the target label for the video [1].

III. Experiments

A. Dataset

The ViViT paper tests the model using large video datasets. Kinetics-400 has around 240,000 training videos and 20,000 validation videos with 400 action classes, while Kinetics-600 expands to 600 classes with over 390,000 training videos. Something-Something V2 includes 169,000 training and 24,000 validation videos, focusing on object interactions. Jester is smaller, with 148,000 clips for hand gesture recognition across 27 classes. Epic-Kitchens features over 39,000 action segments in first-person views, emphasizing detailed actions in kitchen environments. These datasets test the model's ability to handle both simple and complex video tasks [1].

In this project, I used OrganMNIST3D for training and testing. OrganMNIST3D is part of the MedMNIST [5] collection, specifically designed for 3D medical image classification. This dataset consists of 3D CT (Computed Tomography) images, each represented as a 3D array with dimensions of 28×28×28. It focuses on the classification of 11 distinct organs. The dataset is derived from the same source as the OrganA, C, SMNIST datasets and utilizes 3D bounding boxes extracted directly from the CT images. These bounding boxes are then resized to uniform dimensions for consistency.

The organ classes represented in the dataset include the liver, right kidney, left kidney, right femur, left femur, bladder, heart, right lung, left lung, spleen, and pancreas. The dataset is divided into three subsets: a training set with 971 samples, a validation set with 161 samples, and a test set with 610 samples [5].

B. Pre-processing

Each 3D CT image undergoes preprocessing to standardize its dimensions, ensuring compatibility with deep learning models while preserving the structural and spatial characteristics of the organs. This preprocessing step facilitates effective training and evaluation of models in 3D medical image classification tasks [5].

C. Implementation

I implemented the code for this study based on the approach outlined by Gosthipaty and Thakur [4], using their work as a foundation and adapting it to meet the specific requirements of this research.

The model was implemented using Python 3.10.12 and PyTorch 2.5.1. Training and evaluation were performed on NVIDIA GA100 [A100 PCIe 80GB] GPUs, ensuring high performance for deep learning tasks.

Different hyperparameters, which are the model's adjustable factors, were tested. These included learning rates, patch sizes, projection dimensions, the number of attention heads, and the number of transformer layers.

For learning rates, values of 0.0001, 0.001, and 0.00001 were explored. Patch sizes of (4,4,4), (6,6,6), and (8,8,8) were evaluated. The projection dimensions tested were 64, 128, and 256. The number of attention heads was set to 4 and 8, and transformer layers of 6 and 8 were analyzed.

The best performance was achieved using a patch size of (4,4,4), a projection dimension of 256, 8 attention heads, 8 transformer layers, and a batch size of 16. This setup provided the optimal results. Smaller patch sizes facilitated the extraction of detailed features, while larger projection dimensions and more attention heads enabled the recognition of complex patterns in the 3D CT data.

D. Training Details

During training, the AdamW optimizer is utilized with an initial learning rate of 10^{-4}. For learning rate scheduling, multiple approaches were implemented to optimize the training process and enhance model convergence. The **StepLR** scheduler was utilized to gradually reduce the learning rate by a specified factor (γ) at fixed intervals (step_size epochs). This method allowed for consistent adjustments in the learning rate, supporting steady progress during training. Additionally, the **CosineAnnealingLR** scheduler was employed, which follows a cosine decay pattern to smoothly decrease the learning rate from an initial value to a defined minimum (η_{min}) over a set number of epochs (T_{max}). This approach ensured a gradual reduction, promoting stable convergence without abrupt changes. Furthermore, **Cosine Annealing with Warm Restarts** was implemented, wherein the learning rate periodically resets to its initial value after specific intervals (T_{cur}), with these intervals doubling after each restart. This method facilitated the exploration of new regions in the loss landscape, helping the model avoid suboptimal local minima. However, none of the schedulers provided better results in practice.

E. 3D CNN Model

To make a comparison, I chose the 3D CNN (Convolutional Neural Network) model. This model processes videos by applying 3D convolutions, which capture both spatial and temporal features simultaneously.

1) Model Layers and Their Functions

The input layer of the model accepts data with the shape (C, D, H, W), where C represents the number of channels (e.g., 3 for RGB images), D denotes the depth or temporal dimension, and H and W are the spatial height and width, respectively. The input data is passed through the network for feature extraction and classification. The first convolutional block begins with a 3D convolutional layer that applies 32 filters with a kernel size of $(3, 3, 3)$, stride 1, and padding 1, capturing local patterns in both spatial and temporal dimensions. The output is normalized using a 3D batch normalization layer to reduce internal covariate shift. A ReLU activation introduces non-linearity, followed by a 3D max-pooling layer that downsamples the feature map by a factor of 2 in all dimensions, using a kernel size of $(2, 2, 2)$ and a stride of 2.

The second convolutional block has a similar structure to the first but uses 64 filters in its 3D convolutional layer, enabling

the extraction of more abstract and higher-level features. The third convolutional block further increases the model's capacity with 128 filters in its convolutional layer, capturing even more complex patterns. Following the convolutional blocks, a global average pooling layer is employed using the `AdaptiveAvgPool3d` operation, which reduces the feature map to a fixed size of $(1, 1, 1)$, ensuring compatibility with the fully connected layer. Finally, a fully connected layer maps the 128 features from the pooled output to the number of classes (`num_classes`), producing the final class probabilities.

IV. Results and Discussion

The ViViT paper discusses Model 1, which uses spatio-temporal attention and is tested on two datasets: Kinetics-400 and Epic-Kitchens (EK). The model scored 80.0% accuracy on Kinetics-400, showing it can effectively handle large datasets with complex patterns. However, its accuracy on EK was only 43.1

A. Hyperparameter Test

The hyperparameter tuning experiment was conducted to evaluate the performance of the model across various configurations. The tested hyperparameters include three learning rates (10^{-4}, 10^{-3}, and 10^{-5}), three patch sizes ("4,4,4", "6,6,6", and "8,8,8"), and three projection dimensions (64, 128, and 256). Additionally, two different numbers of attention heads (4 and 8) and two different numbers of transformer layers (6 and 8) were explored. These combinations aim to identify the optimal configuration that maximizes model accuracy while maintaining computational efficiency.

The number of layers, patch size, and attention heads in ViViT models are important factors that affect their performance and efficiency. More layers allow the model to capture complex patterns in video data, which can improve accuracy for tasks that need deep feature extraction. However, deeper models require more computing power and may overfit, especially with small datasets. Shallower models are faster and more efficient but may have trouble capturing detailed spatiotemporal relationships in videos.

Patch size determines how detailed the video representation is. Smaller patches help the model capture finer details, which is useful for tasks like action recognition and medical video analysis. However, they also increase computing costs because the model has to handle more tokens. Larger patches save on computing resources but can overlook important details, which may hurt performance in tasks that need precise information.

The number of attention heads affects how well the model can focus on different features at the same time. More attention heads generally improve performance by capturing a wider range of input data. However, too many heads can cause overfitting, increase computing costs, and slow down training. It is important to optimize these settings based on the specific dataset and task to get the best performance and efficiency from ViViT models.

The Table I presents the highest accuracy achieved during the hyperparameter testing. The best configuration achieves a Top-1 Accuracy of 88.36% and a Top-5 Accuracy of 99.84%, with a learning rate of 0.001, a projection dimension of 64, 8 attention heads, 6 transformer layers, and a patch size of (4,4,4). However, this configuration has a training loss of 0.11, indicating that while it achieves the best accuracy, it does not reach the lowest training loss.

For a learning rate of 0.0001, configurations achieve slightly lower Top-1 Accuracy (up to 87.54%) but consistently exhibit better training loss values, stabilizing as low as 0.05. On the other hand, very low learning rates (e.g., 0.00001) result in slower convergence, with higher train loss (up to 1.66) and a substantial drop in Top-1 Accuracy, with the worst performance being 47.54% for certain configurations.

The drop in test accuracy, along with the decrease in training loss from 0.05 to 0.01 for models with a learning rate of 0.0001, projection dimension of 256, 4 attention heads, and 6 or 8 transformer layers using patch sizes of (6,6,6) and (8,8,8), shows that the model is overfitting. The lower training loss suggests that the model learns the training data well, but the decline in test accuracy indicates that it struggles to perform well on new data.

Several factors cause this problem. Larger patch sizes, like (6,6,6) and (8,8,8), reduce the number of tokens. This makes it harder for the model to capture important details in both time and space. As a result, although the model does well with the training data, it fails to understand the complexity of the test data. Moreover, the low learning rate (0.0001) and deeper architecture (8 layers) help the model learn but make it more vulnerable to overfitting by focusing too much on the specifics of the training data.

The big gap between the very low training loss (0.01) and the lower test accuracy clearly indicates overfitting. To fix this, using smaller patch sizes, like (4,4,4), can help improve feature extraction. Regularization methods such as dropout or data augmentation might also reduce overfitting. Additionally, using early stopping during training can prevent the model from fitting too closely to the training data. This analysis shows how important it is to balance model complexity and generalization when tuning hyperparameters.

Overall, while the highest accuracy (88.36%) is achieved with a learning rate of 0.001, the most balanced configurations in terms of accuracy and training loss come from a learning rate of 0.0001. Larger projection dimensions (256) enhance the model's ability to represent features effectively, while smaller patch sizes (4,4,4) allow for better spatial feature extraction, which aligns with the small size of the videos in the dataset. Lastly, deeper models with 8 layers consistently perform better than shallower ones with 6 layers.

B. Model Comparison

To compare the performance of the two models, I evaluate their training and validation loss curves as well as their confusion matrices. The following figures illustrate these comparisons.

TABLE I: Hyperparameter Tuning Results with Top-1 and Top-5 Accuracies

Learning Rate	Projection Dim	Num Heads	Num Layers	Patch Size	Top-1 Accuracy (%)	Top-5 Accuracy (%)	Train Loss (last epoch)
0.0010	64	8	6	(4,4,4)	88.36	99.84	0.11
0.0010	64	8	6	(6,6,6)	82,30	98.69	0.16
0.0010	64	4	6	(8,8,8)	73,93	96.72	0.23
0.0010	64	8	6	(8,8,8)	74.92	97.54	0.20
0.0001	64	8	8	(4,4,4)	77.54	98.69	0.42
0.0001	128	4	6	(4,4,4)	81.64	98.52	0.24
0.0001	128	4	8	(4,4,4)	84.43	99.18	0.22
0.0001	128	8	6	(4,4,4)	84.10	98.52	0.21
0.0001	128	8	8	(4,4,4)	85.74	99.18	0.22
0.0001	256	4	6	(4,4,4)	86.07	99.34	0.14
0.0001	256	4	8	(4,4,4)	86.56	99.67	0.07
0.0001	256	8	8	(4,4,4)	87.54	98.52	0.05
0.0001	256	4	6	(6,6,6)	84.10	97.38	0.05
0.0001	256	4	8	(8,8,8)	82.30	97.20	0.01
0.00001	64	8	8	(4,4,4)	47.54	85.74	1.66
0.00001	256	8	8	(4,4,4)	67.70	95.41	0.73

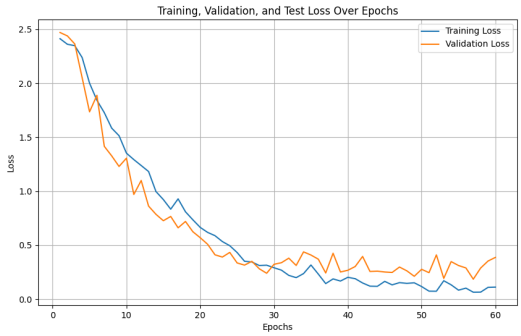

Fig. 3: ViViT: Training and Validation Loss.

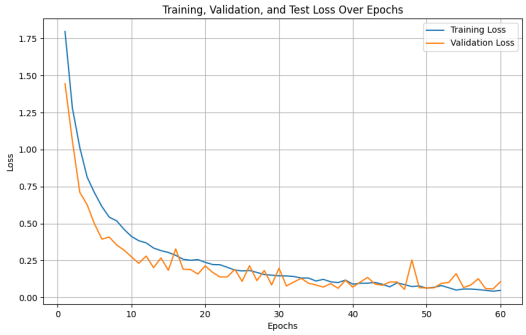

Fig. 4: 3D CNN: Training and Validation Loss.

In Figure 3 the training and validation loss curves for ViViT show slower convergence compared to 3D CNN. Although the losses consistently decrease over the epochs, they stabilize at slightly higher values than those of 3D CNN. The minimal gap between the training and validation losses suggests that ViViT demonstrates good generalization capabilities and is not overfitting.

In Figure 4 the 3D CNN trains more quickly. Both the training loss and validation loss reach lower values earlier in the process. The validation loss stays close to the training loss, showing that it learns effectively and can generalize well.

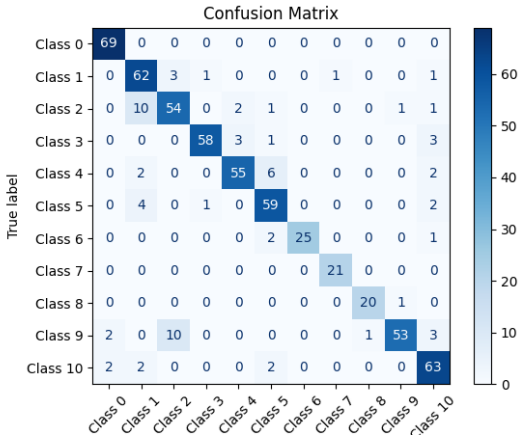

Fig. 5: ViViT: Confusion Matrix.

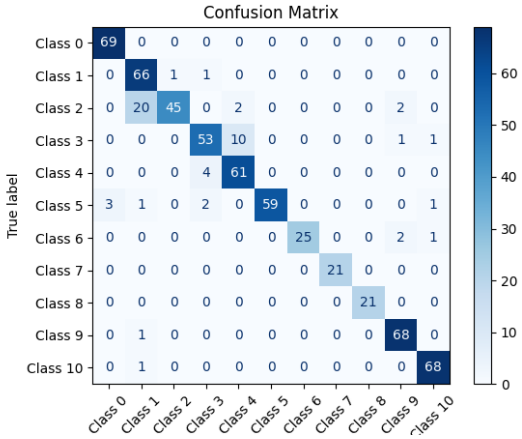

Fig. 6: 3D CNN: Confusion Matrix.

As can bee seen in Figures 5 and 6 the 3D CNN provides better classification accuracy and makes fewer mistakes across all classes. This shows its ability to effectively capture important features over time and space.

In Table II ViViT-1 uses a learning rate of 0.001, a projection dimension of 64, a patch size of (4,4,4), 8 attention heads, and 6 transformer layers. In contrast, ViViT-2 adopts a larger projection dimension of 256 and a deeper architecture with

TABLE II: Comparison of ViViT and 3D CNN Models

Model	Top-1 (%)	Top-5 (%)	Params
ViViT-1	88.36	99.84	304907
ViViT-2	87.54	98.52	6338059
3D CNN	91.15	99.51	279435

8 transformer layers, while retaining the same patch size and number of attention heads as ViViT-1. The 3D CNN performs better than both models, providing results that are faster and more accurate.

The differences, observed in II, performance between ViViT models and 3D CNNs come from their designs and how they work. ViViT-1 has fewer parameters and a smaller projection size, with fewer transformer layers, which makes it less effective than ViViT-2. ViViT-2, on the other hand, has a larger projection size and a deeper structure. This gives it more capacity, but it requires more computing power. 3D CNN has shown better results and the reason of this could be that it is well-suited for small datasets like OrganMNIST3D, which is part of the MedMNIST benchmark and contains 3D medical images with dimensions of 28×28×28 (MEDMNIST). Vision Transformers (ViTs), such as ViViT, often overfit on small datasets unless they are pre-trained on large datasets or combined with advanced techniques like data augmentation and regularization . In contrast, 3D CNNs perform better because they are designed to capture local features and have strong inductive biases, like translational equivariance. These qualities help 3D CNNs generalize more effectively and require fewer parameters compared to ViTs.

V. Conclusion

This study shows that the Vision Video Transformer (ViViT) has great potential for classifying 3D medical images. It can effectively capture important features over time and space without needing special pre-processing. In tests on the OrganMNIST-3D dataset, ViViT achieved a Top-1 accuracy of 88.36

The research revealed that ViViT's success depends significantly on its design choices, such as patch size, projection dimension, number of attention heads, and the number of transformer layers. Using smaller patch sizes (like 4×4×4) helps capture detailed spatial information. Larger projection dimensions (like 256) allow the model to represent data more effectively. Increasing attention heads and transformer layers helps the model understand complex relationships over time and space, but this can slow down processing and increase computational costs.

While ViViT has many benefits, it is slower to converge and requires more computing power compared to 3D CNNs. However, it performs well in generalizing results and can scale effectively, making it a good choice for video classification tasks in medical imaging. Future research could focus on improving efficiency, adding segmentation models to boost accuracy, and adjusting the design for larger, varied datasets. This research highlights the potential of ViViT to improve the analysis of 3D medical images and encourages further innovation in this area.

References

[1] Arnab, A., Dehghani, M., Heigold, G., Sun, C., Lučić, M., & Schmid, C. (2021). ViViT: A Video Vision Transformer. arXiv. https://arxiv.org/abs/2103.15691

[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit,Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In NeurIPS, 2017.

[3] Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., et al. An image is worth 16x16 words: Transformers for image recognition at scale. In ICLR, 2021.

[4] Keras Team, "Video Vision Transformer (ViViT)," Keras.io, Accessed: January 18, 2025. [Online]. Available: https://keras.io/examples/vision/vivit/

[5] Jiancheng Yang, Rui Shi, Donglai Wei, Zequan Liu, Lin Zhao, Bilian Ke, Hanspeter Pfister, Bingbing Ni. Yang, Jiancheng, et al. "MedMNIST v2-A large-scale lightweight benchmark for 2D and 3D biomedical image classification." Scientific Data, 2023.