Vision Transformers for Activity Recognition

James Wensel
Computer Science
Kansas State University
jdwensel@ksu.edu

***Abstract* — One of the most important and fastest growing fields in Machine Learning is Computer Vision. Computer Vision is the process of using a computer to interpret the real world through some form of sensor data, most notably image data. It involves using Artificial Neural Networks to determine patterns or features in the sensor data and using these features to inform other processes. This can even be directly applied to video data by treating each frame in the video as a separate image and performing subsequent processing on each frame. An important sub-field of Computer Vision is Activity Recognition, which seeks to determine the activity an individual or group of individuals in a video are performing. The applications of this field ranges from generating highlight videos in sports, to intelligent surveillance and gesture recognition. Most Activity Recognition systems rely on a combination of Convolutional Neural Networks to perform feature extraction from the data and Recurrent Neural Networks to determine the time dependent nature of the data. I propose the application of a Vision Transformer, a specialized neural network used to make predictions on sequences of data, as well as a nonspecialized Transformer to improve speed and scalability of Activity Recognition Neural Networks without sacrificing performance.**

I. Introduction

Computer Vision is a task that is central to modern Machine Learning and serves to illustrate many of the difficulties associated with it. The standard approach for Computer Vision begins with Convolutional Neural Networks (CNN) for image and data processing. CNNs are a type of Neural Network that are well suited to image classification and recognition tasks [1]. Traditionally, Computer Vision is performed with Deep Convolutional Neural Networks which contain long chains of CNNs of different densities and filter sizes. As early as 2012, this task led to Very Deep implementations of image classifiers [2], and even more recently, Deep CNNs with residual connections that provided even better results [3]. This final approach created ResNet [3], a Residual Network which uses a residual connection to recombine the extracted features with previous inputs at different time steps in the model. This approach helped create more robust models that could be trained successfully in nonideal systems. These models, once trained, can even be transferred to domains in which they did not train, called transfer learning [4]. This process has been done with relative success depending on both the dataset used for initial training and the domain of the testing data [5]. Convolutional Neural Network models, including the ResNet classifier, have seen recent success and have been the dominate models used for Computer Vision tasks. However, the high model complexity of Deep CNNs [6] are not ideal in all systems and show a potential need for new model architecture. For example, on the Internet of Things (IoT), a network of physical devices which generates data from multiple different sources [7], there is a need to process the data generated efficiently. In many of these scenarios, the devices used have limited resources to apply to the image classification or object detection tasks. Even in state-of-the-art designs such as Fog computing [7], smaller edge devices are used to keep the data close to the end user, necessitating more efficient and lightweight implementations.

The Computer Vision subtask of Activity Recognition is no exception to this. Activity Recognition is a task that is more complex than image classification or object detection. Often the activities that are being classified involve some form of time dependence and cannot be determined from a single frame. For example, an alley-oop dunk, one of the most exciting highlights in all of sports, consists of the combination of a drive from a player, a pass to said player, a catch, and a dunk. There are few points, if any, at which a single frame of this sequence would give enough information to determine the action being performed, making it hard to classify this highlight. This is similarly true in Soccer [8], as well as non-sports domains such as intelligent surveillance [9] where most actions consist of a combination of smaller actions that must all be similarly classified, and these classifications are then combined to determine the overall action from the data. This is done by combining the features extracted by a CNN with the time-dependent feature analysis of a Recurrent Neural Network.

Recurrent Neural Networks (RNNs) maintain hidden states that allow for greater impact of time dependencies on their data [10]. Many variants of RNNs exist, with the two most used in Activity Recognition being Long-Short Term Memory (LSTM) units, and Gated Recurrent Units (GRUs). LSTMs have an internal hidden layer that contains multiple memory cells to "remember" previous inputs. These cells inform three different output gates which serve as the output and updaters for future LSTMs [11]. GRUs are much the same but maintain fewer internal states and have two output gates instead of three [12]. However, they maintain similar performances to LSTMs [13]. While RNNs have seen some success, they also suffer from the same complexity issues as CNNs. RNNs are also not obviously parallelizable, as an RNN layer contains multiple RNN units that compute the output of the layer sequentially. As a result, RNNs often have long training times [14]. This can lead to issues when operating in real-time systems or on edge devices. To fix this, Attention Mechanisms were constructed, which allow the model to "Pay Attention" to certain parts of the input and ignore others. Attention was fist used in conjunction with RNNs in sequence-

to-sequence models but was later used on its own in development of the Transformer [14]. The Transformer was shown to not only be more efficient and lightweight, but to have similar performance to traditional methods when trained on a sufficiently large dataset.

In this paper, I propose the use of a specialized Vision Transformer Neural Network to replace the Convolutional Neural Network in the typical Activity Recognition network chain, and a regular Transformer Neural Network to replace the Recurrent Neural Network to increase speed and efficiency of current Activity Recognition models. Sequence-to-sequence models, as well as Transformers and Attention will be described in the next section, along with how they can be applied to the Activity Recognition problem. The implementations of these networks will be described in the Methodology and Implementation sections, followed by some preliminary Experiments and Results for the original implementation of these models.

II. Related Research

A. *Activity Recognition*

Activity Recognition generally consists of two main phases: Feature extraction on the input data sequence, and a combination of the extracted features into a time-dependent classifier. The standard version of this model uses a Convolutional layer that is applied to each time step of the input. The input features extracted from this layer are kept in sequence and collectively input into a Recurrent layer (usually using LSTM units or GRUs), and then passed to a Dense layer for classification [15]. This model structure is outlined in **Fig 1**, and **Fig 2** expands on the CNN layer, looking directly at a single CNN block applied to a single time step. **Fig. 3** expands on the RNN layers, showing the sequential nature of RNNs:

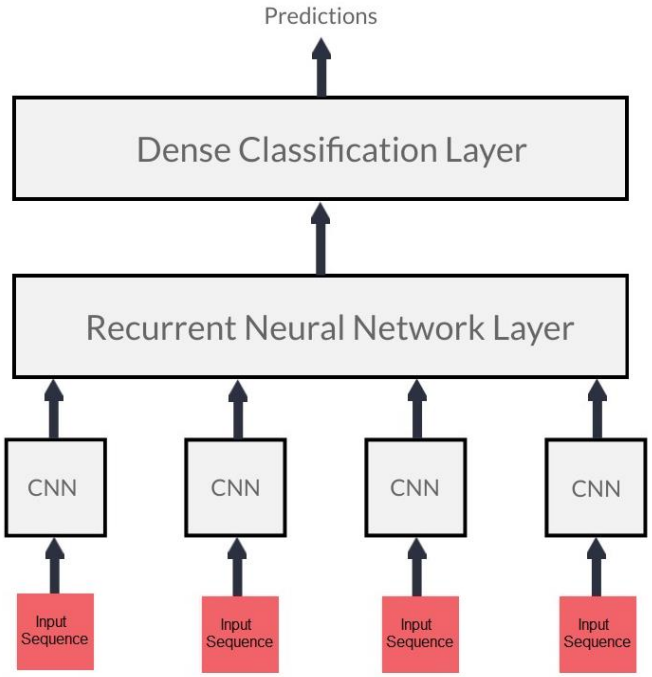

Fig. 1. General Activity Recognition Model Flow

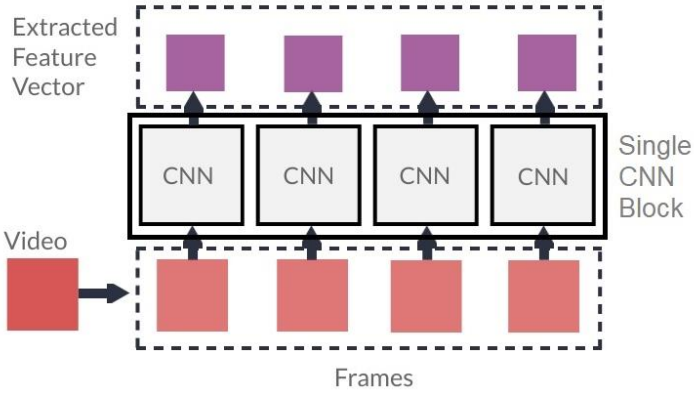

Fig. 2. A Single CNN Block in the CNN Layer of Activity Recognition Model

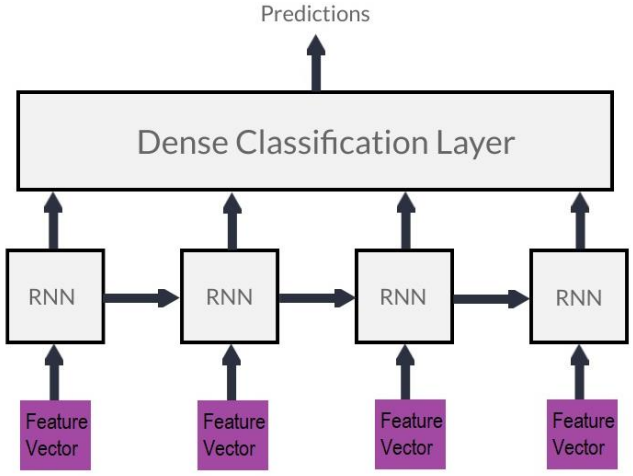

Fig 3. RNN and Output Layer of Activity Recognition Model

The CNN layer will usually contain some CNN chain and can even be a Deep CNN for improved performance. Similarly, the RNN layer can be made up of any number RNN units, and there are usually at least 2 RNN layers stacked on top of each other, feeding into a Dense layer that can also have any number of hidden layers and internal hidden units.

This is the general structure used for Activity Recognition, no matter the type of input data. This model can be applied to signal data from accelerometers and gyroscopes [16], body pose data [17], or image data [18]. Some Activity Recognition Networks will split the image data into multiple channels, such as Russo et. al. [18] who used RGB, Optical Flow, and Slack Foreground Masks to make predictions on Low-Resolution or Extremely Low-Resolution video files. Each of these models expanded upon the structure of the Activity Recognition Model to better suit the needs of the domain they were working in, but all maintained the same general structure.

While this model structure is very common in Activity Recognition, it has flaws that are inherent to the layers themselves. For convolutional layers, in order to be a good classifier on datasets with many classification categories, some form of Deep and / or Residual CNN is necessary [3].

This causes complexity and speed issues on devices with limited capabilities, such as IoT or edge devices. The Recurrent layers have similar problems, as the final RNN in the chain is dependent on all previous units in the layer. This means to calculate each output, all previous outputs must be calculated first, creating a bottleneck that can impact performance [14], especially on edge devices close to the users. To alleviate this, I propose the use of two types of Transformers to replace both the CNN layers and the RNN layers in the model.

B. Positional Encodings

When handling sequential data, like videos, it is important to make sure the information about the position of each element in the sequence is preserved. For example, if the initial sequence is the sentence "I had baked a cake", it holds very different meaning to the sentences "I had a cake baked", and the only difference is the position of a few elements in the sequence. For networks with recurrent connections, this information will inherently be conserved, but for networks without recurrences, that information must be preserved [14]. Generally, this is done by adding some Positional Encoding to each element in the sequence. This can be any number of relevant functions, including offset sinusoidal functions that use the sequence to generate different values [14]. While there is no one way to apply positional information to data, it is a crucial step for sequence-to-sequence models when there are no recurrent connections.

C. Sequence-to-Sequence Models

Sequence-to-sequence models are state-of-the-art neural networks that operate by learning to map an input sequence to an output sequence one step at a time [19]. Once trained, they are able to generate an output sequence from an input sequence, maintaining the meaning of the input by generating an internal representational vector that is used to inform the generation of the output sequence. This process was later expanded to create the Encoder-Decoder structure that is the basis for modem sequence-to-sequence models. The Encoder-Decoder model has two distinct parts that operate independently: an Encoder and a Decoder. The Encoder takes an input sequence and encodes it into a vector that contains a representation of the meaning of the initial sequence. This encoding is then fed into the Decoder which converts it into an output sequence in the target domain [20]. This general Structure can be seen in **Fig. 4**. Some Encoders (those built from RNNs) generate the internal encoding sequentially, beginning with the first element and proceeding one element a time [20]. Newer Encoder architectures (specifically the Transformer architecture) can generate the entire input encoding in one step (for the Transformer, this is done by using Self-Attention) [14]. The Decoder takes the encoding, and at the first time-step generates a single element in the output sequence. The previously generated elements are used as inputs to the Decoder at the next time step, which combines these elements with the encoding to generate the next element in the sequence. This process is repeated in the Decoder until the entire sequence is generated.

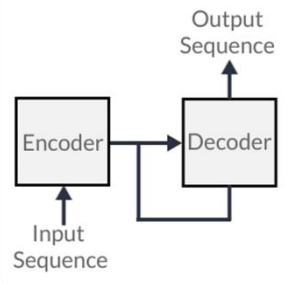

Fig 4. Overview of Encoder-Decoder Structure

This type of model is most commonly associated with Natural Language Processing, and most specifically Machine Translation. In Machine Translation, the input sequence is a sentence in some language that will be translated by the model. The Encoder takes a sentence of positionally embedded words and creates an encoded representation of the sentence. This is either done all at once or one word at a time. This encoding is then given to the decoder, which uses it to generate the first word in the target language. It then uses the first word in the output sequence and combines it with the same encoding from the first step to generate the second word. Then the second word is used to generate the third, and so on. This process is repeated until the entire sentence has been generated in the target language. The two halves of the networks are trained together, and therefore complement each other in the learning process. However, the basic Encoder-Decoder structure, using RNNs for each, has issues as the length of the sequence gets longer [21], and it has trouble with interpreting and generating sentences longer than the sentences used to train it. This is accounted for and improved upon by the Transformer model, which will be discussed in the next subsection.

Since their inception, the Encoder-Decoder model structure has become the basis for the entire field of Natural Language Processing. However, it is not immediately apparent how this type of model can be applied to the field of Activity Recognition. The direct application of sequence-to-sequence models to Activity Recognition will be describe in detail in the Methodology section.

D. Transformers

The transformer is a special case of sequence-to-sequence model introduced in 2017 [14]. The Transformer introduced a special type of Self-Attention called "Scaled Dot-Product Attention". The particulars of this will be discussed in the next subsection. The Self-Attention of the Transformer completely replaces the RNNs in both the Encoder and Decoder and allows the model to use the entire input sequence in the Encoder at once. This alleviates one of the larger issues associated with RNNs, their inability to be parallelized. The entire Encoder can be run in parallel, which allow for large speedups of the network in both training and predicting. The Decoder in the Transformer is still sequential and thus must be run one element at a time, but this eliminates half the bottleneck entirely. Models based on the Transformer design, such as BERT [22], are extremely robust and have accuracies rivaling other sequential models.

The Transformer has a very similar flow to the general Encoder-Decoder, but with a few key differences. First, the input sequences are given a numerical representation and positional information is added, similar to the other models. This input is then passed into a Self-Attention layer that learns to "Pay Attention" to certain aspects of the input sequence. The Attention the Encoder uses is also said to be "Multi-

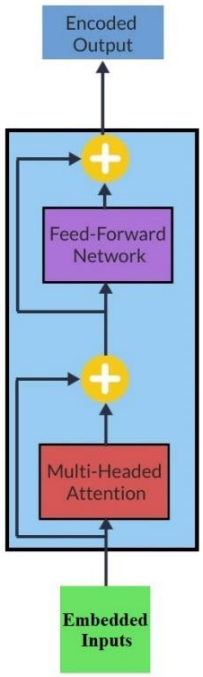

Fig 5. Transformer Encoder

Headed Attention", as the encoder uses multiple Attention blocks. This is to allow each block to pay attention to different aspects of the input sequence and, ideally, preserve different levels of meaning at each. These are then passed into a Feed-Forward network that combines all the information learned in each Attention head. After both the Multi-Head Attention block and the Feed-Forward network, a residual connection is added to improve performance of the model, like the residual connections in some Deep CNNs like ResNet [3]. The output from each section is combined with the input, and the result is normalized. This allows the Encoder to be more robust and allows it to learn better at each step. The Encoder structure can be seen in **Fig 5**.

At this point, the encoded input sequences have been generated, and they are ready to be passed to the Decoder. The Decoder first passes all previously generated outputs to itself and applies Multi-Headed Masked attention to itself which will also be described in the next subsection. This is then fed into another Multi-Headed Attention block, but this time it is combined with the encoded output from the Encoder block. The output of this layer is fed to a Feed-Forward network, similar to in the Encoder block. There are again residual connections around both Multi-Headed Attention blocks and the Feed-Forward network that are combined with the original input and normalized. Finally, the Decoder output is passed to a Linear layer with a Softmax activation function to get the probability distribution as the output. This is then passed back to the beginning of the Decoder as the input of the next layer. The Decoder structure can be seen in **Fig 6**, and the overall structure of the Transformer can be seen in **Fig 7**.

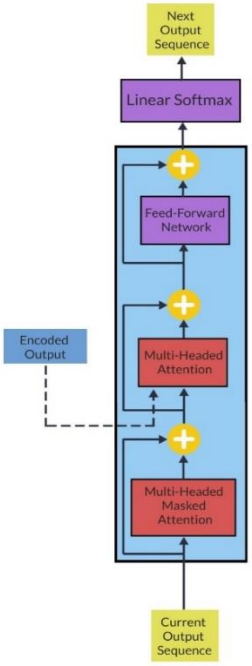

Fig 6. Transformer Decoder

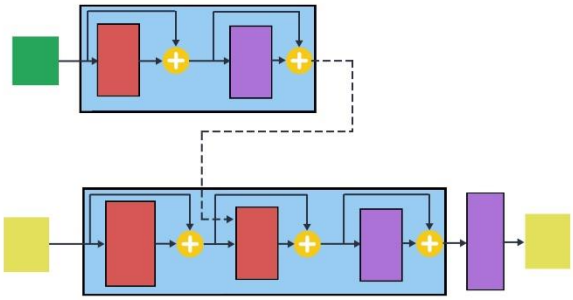

Fig 7. Transformer

The Transformer has quickly become the most important neural network for sequence-to-sequence tasks. While it is clear it is extremely powerful, it is still not obvious how this can be applied to video data or Activity Recognition as a whole. This will be described in detail in the Methodology section.

E. Self-Attention

Self-Attention is a process by which a neural network learns which parts of the input it should focus on. While there are many kinds of attention, such as additive attention [21], this section will focus on Scaled Dot-Product Attention [14] as it is the basis for the transformer models as created by Viswani, et. al. [14]. Attention is derived from database queries. The process begins with a set of key-value pairs and involves mapping a set of queries to those keys and combining the result with the initial values to get the output. When given a set of queries Q, keys K, and values V, where Q, K, and V are all matrices, the Scaled Dot-Product Attention can be calculated by **eq. (1)** [14, eq. (1)]:

$$Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \qquad (1)$$

While not immediately obvious, this equation is based on the equation for cosine similarity [23]. Looking at that equation will help to better explain what Attention is doing. The cosine similarity of two vectors A and B is given by **eq. (2)**:

$$CosineSimilarity(A,B) = \frac{A \cdot B}{\|A\|\|B\|} \qquad (2)$$

This equation returns a value between +1 and -1, describing how similar vectors A and B are, which is the equivalent of the Cosine of the angle between the two vectors. First the dot product the two is computed, then it is scaled by the product of the magnitude of A and B. In the case of Scaled Dot-Product Attention, the Queries matrix and the transpose of the Keys matrix serve as A and B, and instead of scaling by the magnitude of each, the result is scaled by the root of the dimensionality of the keys. This scaling value was chosen as an extension to Dot-Product attention to give it better performance as the keys matrix becomes larger. This was done because the dot product at higher dimensionalities of the key matrix will be very large, having a negative effect on the performance of the softmax function [14]. Using matrices for both A and B instead of individual vectors allows for the computation of multiple dot

products at the same time. The result of the scaled dot-product is then passed through a softmax layer, giving us a probability matrix that is used as the weights for the values matrix (when instead performing Masked Multi-Headed Attention, the mask is applied before the softmax layer, but the result is the same). This process can be seen visually in **Fig 8**. This can also be written to resemble typical neurons more closely, as in **eq. (3)**:

$$y_k = W x_k$$
$$\text{where } W = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right) \text{ and } x_k = V \qquad (3)$$

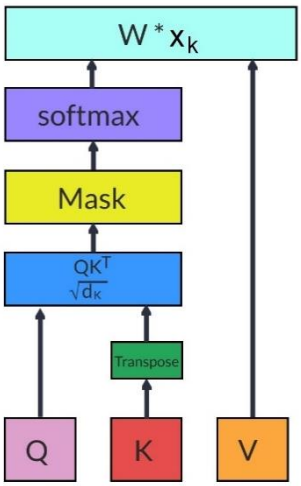

Fig 8. Scaled Dot-Product Attention

The Encoder and Decoder attention blocks each have different inputs for the queries, keys, and values. For the Encoder, the queries, keys, and values matrices are 3 copies of the input sequence. This seems counter intuitive that all 3 would be the same matrix, but the benefit of this is shown clearly by an example. Assume the input is an encoding of a M-word sentence with positional information added. Each row in the matrix corresponds to the vectorized representation of a word in the sentences. When this is transposed and the scaled dot-product is calculated, the result will be an M-by-M attention matrix where each row still corresponds to a word in the initial M-word sentence, but the data now represents something different. Column *j* in row *i* now corresponds to how similar word *i* is to word *j* in the original sentence. Once the softmax is applied to the attention matrix, it will give the similarity of *i* and *j* as a value between 0 and 1, with 1 meaning they are completely alike, and 0 meaning they have nothing in common. In the final attention matrix, if row 1 column 3 has a value of 0.92, that means the meaning of the first and third words are heavily tied together. If instead row 2 and column 5 have a score of 0.2, the meaning of the two words is extremely unrelated in this sentence. This final matrix is then used to weight the values to give the output of the attention block. Combining the values with the weights matrix will create a final matrix where each row still corresponds to the initial words in the input sentences. However, the data at each row will now be a weighted dot product that combines how much each word in the sentences impacts the word at the current row index. The output of this layer is demonstrated in **Fig 9**.

The Decoder Attention Blocks operate the same way but with 2 key distinctions. First, for the Masked Multi-Headed Attention block, the queries, keys, and values are the current output sequence that has been generated (or in the case of training, the label sentence). The only difference at this block

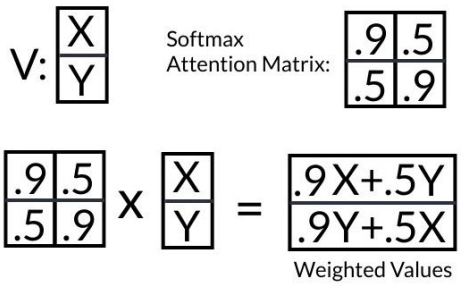

Fig 9. Example Weighted Values Matrix

is the attention matrix is masked before the softmax function is applied. The mask simply sets all values in the sequence beyond the last generated output element equal to 0. This is done because, during training, the labeled output sequence is fed into the decoder instead of the generated output. The labeled output sequence is used to ensure correct learning, even after incorrectly predicting the earlier elements of the sequence. At each step, even if the Decoder previously made an incorrect prediction, it will have the correct and necessary information to predict the next element in the sequence. The mask is then used to allow the Decoder to select how much of correct sequence to use. It will make the value 0 for all elements later in the sequence than the Decoder has made predictions.

Finally, the result of this first Attention Block will be a weighted output sequence matrix. This weighted output will be a matrix that contains all contextual information that is currently known about the output sequence. This matrix is used as the values for the second Multi-Headed Attention block in the Decoder, and the queries and keys are the encoded output of the Encoder block (remember after the attention block, the sequence is normalized and fed into a Feed-Forward Network and is again normalized). While this is no longer *exactly* the output from the Attention block in the encoder, it has been transformed into an encoding of the final weighted values that still maintains its contextual information. Using the encoded output as the queries and keys allows us to calculate a set of weights directly from the Encoder output, and when those weights are applied to the weighted output matrix, it will apply the encoded contextual information of the input sequence to all currently know contextual information about the output sequence. This gives us a contextual matrix that contains all known context from the input sequence and current output sequences. This is then combined with a residual connection, normalized, and passed through one last Feed-Forward network, normalized one last time, and passed into a linear classification layer with softmax activation function to predict the next word in the output sequence. These final layers all apply extra preprocessing layers to help improve consistency of the output predictions.

For a more detailed description of both the Transformer Network and Scaled Dot-Product attention, see Viswani et. al, [14]. Much like the Transformer Network itself, Scaled Dot-Product Attention is the current state-of-the-art Attention method that is applied to many domains, and is even used beyond Transformer Networks [17]. However, the direct application of the Transformer model to Activity recognition is still not obvious and will now be discussed.

III. METHODOLOGY

A. RNN Layers

The application of the Transformer to the RNN Layers of the generalized Activity Recognition Model seen in **Fig 1** is the simplest of the two applications attempted. While the Transformer was designed for, and is most used for, Machine Translation tasks, there is nothing inherent to the model requiring word embeddings to be used as the inputs to the model. The Encoder can take any sequential input, so long as the relevant positional information has been encoded in the sequence. It will extrapolate out the relative hidden meaning from it. So, if instead of a sequence of words, the input was a sequence of frames, where each frame was embedded with the information relevant to its order in the video, the Encoder would be able to use it to calculate an output containing the "meaning" of the video. While the application of the encoder to get meaning from the frames means video data can be used as the input to the Encoder, an important question arises: What sequence will be generated by the Decoder? Shockingly, the answer is none.

In the case of video data (or any time series data that does not generate an output sequence), there is no need to use the Decoder at all. Instead, the output of the Encoder in my model is fed directly into a flattening layer to make the data 1-dimensional, and then to a Dense layer to make predictions. Conceptually, this structure makes sense. After the Multi-Headed Attention layer, the Encoder has created a matrix from the initial sequence that has been weighted by what the model has determined is important. This is then passed into a Feed-Forward layer to create an encoded output. This output can be thought of as an encoded representation of the "meaning" of the input sequence. So instead of using said "meaning" to generate a sequence, this "meaning" can instead be used immediately to classify the input, which is exactly what my model does. And because my model does not use the Decoder, there is no bottleneck in this section. Unlike when RNNs are used to generate the final representation of the data, there is no need to wait on the completion of any chains of values waiting on previous states. Even in the application of the Transformer to sequence-to-sequence models, the data gets bottlenecked at the decoder step (though this bottlenecking is much less impactful than the general bottlenecking of RNNs). Since all values needed to compute the output of the Transformer are present at the start of computation, it is easily parallelizable, which allows for increased computation speed. The Transformer is also a more lightweight model and would operate better in a limited environment. While the application of the Transformer model to any time series data would follow similarly, it is not easily apparent how to apply this to non-time dependent data, such as single images, in a way that it could replace CNNs.

B. CNN Layers

The Transformer needs to be applied to data in a sequence. However, the Transformer itself does not learn anything from the positional data of the input sequence. This is all done externally by the positional embedding of the input sequence data. So, if the non-time series data was somehow made to be sequential, with the information about its locality still applied by the positional embedding, then the data could be used in the same manner as any time series data. It is by this process that my model applies the Transformer to the individual frames of the video data. To turn an image into sequence data for a Transformer, it is first split into patches, or smaller subsections of the original image. This process can be seen in **Fig 10**:

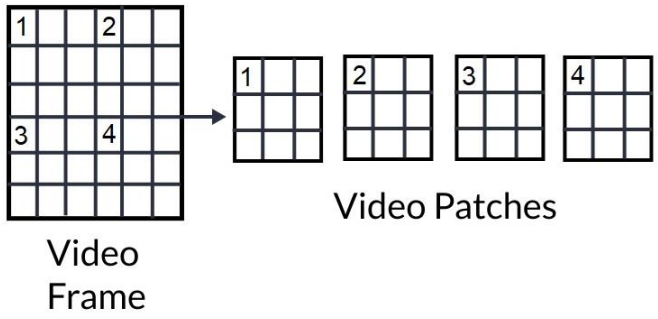

Fig 10. Outcome of Splitting an image into Patches

These patches are then grouped in sequence to be used as the input to the Transformer. The transformer will then perform a pseudo feature extraction, by determining the internal "meaning" of the image, and using a Dense layer to classify this output, in the same manner as for time series data. The patching of an image and using it as a sequential input to a Transformer is the process taken by a Vision Transformer [24]. Vision Transformers have been shown, when trained on a sufficiently large dataset, to perform as well as state-of-the-art Deep Residual CNN networks [24], while remaining more lightweight and efficient than long CNN chains. The last important step in the application of Transformers for Activity Recognition is utilizing the Vision Transformer alongside the Transformer.

C. RNN and CNN Layers

An important note before this section is it is not necessary to implement a Transformer model for both halves of the Activity Recognition model. The Transformer can replace the RNN layers only and train on the features extracted by a Deep Residual CNN more efficiently and with as much accuracy as the typical LSTM or GRU chains. Or the Vision Transformer can be used to replace the Deep Residual CNN to perform more efficient feature extraction from the frames and pass the data to the LSTM chains like normal. Still, combining the Transformer and Vision Transformer will prove to alleviate the largest amount of complexity and efficiency issues encountered by current Activity Recognition models.

To use a Vision Transformer, it should first be trained separately on a large image recognition dataset, such as ImageNet [25]. The Vision Transformer needs a large dataset to train, and there are no sufficiently large Activity Recognition datasets that would allow the Vision Transformer to achieve adequate performance if trained in conjunction with a Transformer. Once trained, it will be transferred to be the backbone of the feature extraction section of the New Activity Recognition model. Then, for a specific video input, a copy of the trained Vision Transformer will be applied to each frame. The output from this layer will be a series of encoded image data generated from the patches of each frame in the sequence.

This will then be in a form that is able to be processed by the Transformer that performs the final classification. In the next section, I outline my implementation of both a Transformer model and a Vision Transformer model as outlined in this paper.

IV. Implementation

My model is built in Python 3.8.4 using TensorFlow through the Keras API, utilizing Numpy for data manipulation and OpenCV for video processing. It is broken into a 3 key sections: Data Preprocessing, Custom Layers, and Model Controllers. These sections will be discussed in detail below. Afterwards, the important Hyperparameters for each model will be discussed.

A. Data Preprocessing

There are two important data preprocessing files: The video preprocessor and the image preprocessor. The video preprocessing file is used to get the videos in the training and testing datasets ready to be run by the model. It subsamples a select number of frames from the videos to create consistent length sequences. The entire video is not used for 2 reasons. It is unnecessary, as most frames contain the same data as their surrounding frames and using an entire video would significantly increase model complexity with little to no tangible benefit to accuracy. The selected frames are then resized to either 224x224 pixels or 64x64 pixels. 224x224 pixels is used as it is the input size of ResNet [3], which is implemented alongside the Vision Transformer. 64x64 pixels is used as it is the size of the videos in the dataset, but both ResNet and the Vision Transformer can accept video files of any size. After resizing all selected frames, they are appended to an array containing all processed video files, and the associated video label is appended to the same index in a labels array for later training and testing. These two arrays, along with an array containing the file path to each video file are saved for use later.

The other preprocessing file prepares the images used to train the Vision Transformer. It simply resizes the images in the dataset and saves them. This is done to make sure the images used to train the Vision Transformer are the same size as the expected size of the frames from the video. Similar to the video frame size, any size could be selected, but 64x64 pixels was used.

Instead of completing these preprocessing tasks first, they could instead be applied as layers at the beginning of the model, but I chose not to do this. I wanted to save myself time each time I trained the model, as the data processing took a lot of time (sometimes over 40 minutes on its own). I also wanted to get a more accurate representation of the time spent training or making predictions in the model instead of time spent processing data, so I chose to complete the preprocessing once beforehand.

B. Custom Layers

To implement the Transformer and Vision Transformer, 3 Custom Layers were created, as well as a function wrapper to create a Transformer: PositionalEncoding, Encoder, Patches, and BuildEncoder function

1) PositoinalEncoding: This layer applys the positional data to the input sequence. It takes as an input the length of the sequence and the dimensionality of the linear projection layer. The length of the sequence will either be the number of frames extracted from each video, or the number of patches extracted from an image, depending on whether the Transformer or Vision Tranformer is being used. The layer creates a positional encoding space and adds the associated positional data to each element in the sequence to capture the positional data. It also linearly projects the input to any size chosen through the linear layer (Usually this is left the same or slightly downsampled, but any could be selected).

2) Encoder: This layer is the layer that serves as the Transformer for these models. Since none of the data generates a sequential output, the Decoder is not implemented. The layer first takes as input the dimensionality of the input (which is the same as the dimensionality of the output from the PositionalEncoding layer). The input dimensionality will be conserved through the encoder as it is used as the dimensionality of the final Dense layer in the Encoder. The next input is the dimensionality for the internal Dense layer. This can be anything, but is usually set to twice the input dimensionality. The next input is the number of heads for the Multi-Headed Attention in the Encoder, usually 6 but could be any value. Finally, the activation function for the internal dense layer can be passed but is by default an ReLU activation function.

This layer first computes the Attention matrix from the given inputs using the built in MultiHeadAttention function from Keras. The output of this is the weighted values matrix. This is than normalized and passed into the dense layer, normalized again, and returned according to the Transformer architecture outlined in this paper.

3) Patches: This layer splits an image into patches. It is given an input of the length of one side of the patch in pixels (8 was used but could be any value, smaller patch sizes will create more patches). This layer uses TensorFlow's built in patch extraction function to create patches, then reshapes the resulting tensor to maintain the correct shape of the output (if the input is a (Batch_size, 64, 64, 3) shaped image tensor and patches are 8x8, it will return a tensor with shape (Batch_size, 64, 192) containg all 64 created patches from the image [192 = 8x8x3]). The patches are not kept in a (8,8,3) tensor, but instead the pixel data from each pixel in the patch is placed into a 1-Dimensional tensor. This is done to ellimnate future reshaping of each individual patch.

4) BuildEncoder: This is a functional wrapper used to build a specific Transformer layer according to user specifications. It takes as input the sequence length (number of frames) and the input dimensionality of the data tensor. It also takes as input all relevant information for the PositionalEncoding layer (embedding layer dimensionality) and Encoder layer (dense layer dimensionality and number of attention heads). It then creates and returns a Keras Model type Transformer according to this specification. To use this model, it will be passed an input tensor with the expected shape, which will be passed to

the positional encoding. After the positional data is applied, it will be passed to the Encoder where it performs Attention and creates a weighted values matrix, which serves as our output matrix.

C. Model Controllers

The model controllers are files that are used to either create the models, run them, or output test data from them. Some function parameters are not mentioned with their associated functions. Any parameters that are not mentioned have previously been mentioned and will be covered again in the Hyperparameters subsection.

1) Create Models: The model creation file has 4 functions, each creates a different model. The ResNet_Model function generates an instance of the pretrained ResNet50 with ImageNet weights and global average pooling. This is applied over an input sequence using the Keras TimeDistributed wrapper, and used to perform general feature extraction. The LSTM_Model function generates a specified number of LSTM layers following the general Activity Recognition RNN layer architecture. The last layer will have a specified number of LSTM units, and each proceeding layer will have twice as many units as the following layer. VisionTransformer_Model function creates a model ready to train for image recognition. It will have a specified number of transformers stacked on top of each other. Finally the Transformer_Model function creates a Transformer model with the specified number of transformer layers for classification of video data.

2) Run Model: The model running file contains 4 functions that are used to train, test, and evaluate a model, as well as making predictions and outputing relevant data. First, the FitModel function is passed a model and the data to train on, as well as the batch size and number of epochs to use. It trains the given model with the data provided, with the specified batch size and number of training epocs. The PlotWholeModel function is a helper function that visualizes a model as well as all functional layers. When a functional layer is found, the function calls itself, passing the function layer instead of the whole model to print the model hidden in the functional layer. PredictModel is used to make predictions on a model without training. The model and prediction data are passed. Finally, OutputModelResults is used to pass training and timing data to the file that handles data output.

3) Output Model Data: The data outputing file contains 3 functions used to output data obtained from training any of the different model types. The plot_metric function prints the training results to a graph, using the print_plot function as a helper to peroform the printing, and the print_time function prints the timing results of the tests out to the console.

D. Hyperparameters

This section will contain a listing of each Hyperparameter for the model, and a brief description of each, including the values used when training the model.

1) SEQUENCE_LENGTH: The number of frames to keep from each video (Usually 20).

2) IMAGE_HEIGHT: The height in pixels to resize either the images for the Vision Transformer image classificaiton or the video frames for Activity Recognition to (Usually 224 or 64).

3) IMAGE_WIDTH: The width in pixels to resize both the images for the Vision Transformer image classificaiton or the video frames for Activity Recognition to (Usually 224 or 64).

4) projection_dim: The dimensionality of the projection of the input sequence by the PositionalEncoder (usually left at or close to input dimensionailty as possible)

5) dense_dim: The dimensionality of Transformer's Feed-Forward Network's hidden layer (this does not change the output dimensionality as it is projected back after to the input dimensionality after) (normally 2x input dimensionality)

6) activation: The activation function for Transformer's Feed-Forward Network's hidden layer (defalts to ReLU)

7) num_heads: The number of heads for Multi-Headed Attention (usually 6).

8) patch_size: The length of one side of the patch to create in pixels (Usually 8 [creates 8x8 patches from image]).

9) LSTM_layers: The number of LSTM layers in a LSTM Model (Usually 3)

10) LSTM_units: The number of LSTM units at output of layer of a LSTM Model (all proceeding layers will have 2 times their following layer) (Usually 128).

11) transformer_layers: The number of Transformers to stack in a Transformer or Vision Transformer Model (Usually 6)

12) cateogries: The number of video or image categories to train on (determined by the dataset being used)

13) batch_size: The number of examples to train on before updating the weights of model (Usually 128)

14) epochs: The number of times to train using the entire training dataset (Usually 50 or 100)

V. Experiments

Two sets of preliminary experiments were conducted using the models created from the previously described methods. The first involved training a Vision Transformer and testing its performance against a Deep Residual CNN, specifically ResNet50. The second involved using the ResNet50 model for feature extraction of Activity Recognition videos and using these to train both a Transformer and LSTM network, and to compare their results on the shared dataset. The results of these experiments are preliminary, and more work is required to get the Transformer models ready for real world applications. The dataset used for training the Vision Transformer is the CIFAR-100 dataset [26], and the Feature Extraction and Activity Recognition Tasks are done using the UCF-101 dataset [27].

For the first experiment, I created a Vision Transformer with 6 internal Transformer layers. It was trained using the CIFAR-100 dataset contained 50,000 images from 100 different categories. Afterwards, both the Vision Transformer and ResNet50 were used to perform feature extraction on a subset of the UCF-101 dataset with 20 randomly selected video categories, and 2500 video files in total. The models were then timed and these results are shown in the next section.

For the second experiment, I created a Transformer Model with 6 Transformer layers, and an LSTM Model with 6 LSTM layers, with [4096, 2048, 1024, 512, 256, 128] LSTM units per layer from the input layer to the output layer of the model. I first used ResNet50 to perform feature extraction on a subset of the UCF-101 dataset containing 20 random classes, and 2500 videos. The models trained on these classes for 50 epochs. The results of these experiments are shown in the next section.

VI. Results

A. Vision Transformer vs Deep Residual CNN (ResNet50)

The first results of note are the results of the image classification training of the Vision Transformer. The CIFAR-100 dataset contains 50,000 training images and 10,000 testing images of 100 different images classes. It took the Vision Transformer 136 minutes to train on the data over 27 epochs (its training terminated early because of an early stopping callback), and it had a validation accuracy of 67% once its training was completed. The train time is longer than necessary because I did not implement any parallelization for the training of the model. I believe the accuracy is only 67% because the CIFAR-100 dataset does not contain enough images and classes to train the Vision Transformer as much as is necessary. For example, Dosovitskiy et. al. [24] used over 1 million images and 18 thousand classes to train their Vision Transformer. After this, the trained Vision Transformer and ResNet50 were used to make predictions on a subset of the UCF-101 dataset containing 2500 videos and 20 video classes. The videos were preprocessed before being given to the models, and were given the shape (2504, 20, 64, 64, 3) for the Vision Transformer (2504, 20, 224, 224, 3) for ResNet50. These input shapes indicate there were 2504 videos of 20 frames each, containing frames of size 64x64 or 224x224, where each pixel contained RGB values. The time required to build the models and use them to make predictions over the entire dataset is shown in **Table (1)**:

Time	Vision Transformer vs ResNet50	
	ViT	***ResNet50***
Build	2.08 seconds	1.41 seconds
Predict	1 minute 35.64 seconds	20 minutes 35.64 seconds

Table (1)

The difference in speed between the two networks is significant. To make predictions on all 2500 videos, the Vision Transformer needed less than 2 minutes, while ResNet was 10 times slower, running in 20 and a half minutes. This illustrates the potential for the Vision Transformer to be significantly better than Deep Residual CNNs in real-time environments. Right now, the main issue is the accuracy of the Vision Transformer model, as 67% is too low to be considered consistent. If the accuracy of the Vision Transformer was closer to that of traditional Deep CNNs, it could be a very good replacement for them.

B. Transformer vs LSTM Network

For this test, a Transformer Model with 6 Transformer layers and an LSTM Model with 6 LSTM layers were both trained using the results of performing feature extraction on a subset of the UCF-101 dataset. The feature extraction was done using ResNet50 prior to being given to the models, so all times recorded are times directly related to the performance of the two models. They were each trained for 50 epochs, and their build, train, test, and prediction times are shown below, as well as their prediction accuracy on the dataset, in **Table (2)**:

Model	Transformer Vs LSTM				
	Build	***Train***	***Test***	***Predict***	***Accuracy***
Transformer	1.03s	3m 0.88s	0.48s	1.17s	97%
LSTM	1m 20.46s	128m 14.91s	13.52s	14.67s	90%

Table (2)

These results show one important thing: the Transformer is significantly faster than the LSTM network. The difference in accuracy can be explained by the fact that there was only 1 test done. There is the potential that if more tests were run, the accuracies of the two models would be much closer. The most notable difference is the differences in overall model speed. The Transformer was able to train on all 2500 videos in 3 minutes while the LSTM network took 130 minutes, a full 43 times slower than the training time of the Transformer. However, prediction time is much more important than training time, but even in that metric the Transformer significantly outperforms the LSTM network. The LSTM took 14.67s to make predictions, while the Transformer was able to make predictions on the same data in 1.17s, over 10 times faster. This shows that overall, the Transformer is a much more efficient network that does not sacrifice any accuracy for the speed it gains.

VII. Conclusions

Activity Recognition is a newer field in Computer Vision that involves recognizing the actions taken by humans through sensor data. Currently, Activity Recognition relies on very dense and complex networks. While these networks work well in theory, they do not translate well to real-world applications. Deep Residual CNNs and RNN chains, the backbone of modern Computer Vision and Activity Recognition, can suffer from complexity and speed issues, making real-time applications or computations in limited resource environments difficult. There is a need to advance Activity Recognition models beyond what currently exists, and I believe this can be done with the Transformer.

The Transformer is one of the most influential Neural Networks created in the last decade. It has revolutionized sequence-to-sequence modeling and shown that it is possible to create very accurate and lightweight models with nothing more than matrix multiplications and Fully Connected Artificial Neural Networks. This paper sought to show the potential applications of Transformers beyond sequence-to-sequence models. It was also shown that there is potential for

Transformers to create significant improvements in Computer Vision and Activity Recognition by replacing both sets of complex networks it currently relies on. The Transformer, and its extension the Vision Transformer, show promise in these domains, with the potential to significantly improve on current state-of-the-art models. Their application into nonsequential and non-time dependent datasets could prove to make huge improvements in Machine Learning tasks that otherwise would not have benefited from the improvements made to sequence-to-sequence models.

VIII. Future Research Directions

This research will be extended by completing more exhaustive testing of the Transformer and Vision Transformer in Activity Recognition and Image Recognition. Training a Vision Transformer on a large dataset such as ImageNet and applying the newly trained Vision Transformer to Activity Recognition tasks could show its practicality over Deep Residual CNNs. More exhaustive testing over the entire UCF-101 dataset for different configurations of the Transformer and LSTM networks could also serve to further solidify the Transformers usefulness in nonsequential domains. A logical next step is the combination of the Vision Transformer and Transformer to apply to Activity Recognition and compare with current state-of-the-art methods. Finally, applying parallelization methods and using this model in real-time and limited resource environments to push this model architecture to its limits would show the true power of the Transformer as a whole.

IX. References

[1] D. Ciresan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," *2012 IEEE Conference on Computer Vision and Pattern Recognition*, 2012.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[3] He, Kaiming, et al. "Deep Residual Learning for Image Recognition." *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[4] S. Amos, "When Training and Test Sets Are Different: Characterizing Learning Transfer," *Dataset Shift in Machine Learning*, pp. 2–28, 2008.

[5] F. Timme, J. Kerdels, and G. Peters, "On the Robustness of Convolutional Neural Networks Regarding Transformed Input Images," *Proceedings of the 12th International Joint Conference on Computational Intelligence*, 2020.

[6] D. Strigl, K. Kofler, and S. Podlipnig, "Performance and scalability of GPU-based Convolutional Neural Networks," *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, 2010.

[7] A. Munir, P. Kansakar, and S. U. Khan, "IFCIoT: Integrated Fog Cloud IoT: A novel architectural paradigm for the future Internet of Things.," *IEEE Consumer Electronics Magazine*, vol. 6, no. 3, pp. 74–82, 2017.

[8] K. Muhammad, H. Ullah, M. S. Obaidat, A. Ullah, A. Munir, M. Sajjad, and V. H. de Albuquerque, "Ai-driven salient soccer events recognition framework for next generation IOT-enabled environments," *IEEE Internet of Things Journal*, pp. 1–1, 2021.

[9] J. Qin, L. Liu, Z. Zhang, Y. Wang, and L. Shao, "Compressive Sequential Learning for Action Similarity labeling," *IEEE Transactions on Image Processing*, vol. 25, no. 2, pp. 756–769, 2016.

[10] A. Graves, A.-rahman Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013.

[11] S. O. Ojo, P. A. Owolawi, M. Mphahlele, and J. A. Adisa, "Stock market behaviour prediction using stacked LSTM Networks," *2019 International Multidisciplinary Information Technology and Engineering Conference (IMITEC)*, 2019.

[12] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder–decoder approaches," *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, 2014.

[13] J. C. Heck and F. M. Salem, "Simplified minimal gated unit variations for recurrent neural networks," *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2017.

[14] A. Viswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention Is All You Need," *31st Conference on Neural Information Processing Systmes (NIPS), Long Beach, CA, USA.* , 2017.

[15] S. Yao, S. Hu, Y. Zhao, A. Zhang, and T. Abdelzaher, "DeepSense," *Proceedings of the 26th International Conference on World Wide Web*, 2017.

[16] H. J. Ma, W. Li, X. Zhang, S. Gao, and S. Lu, "AttnSense: Multi-level attention mechanism for multimodal human activity recognition," *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, 2019.

[17] B. Debnath, M. O'Brient, S. Kumar, and A. Behera, "Attention-driven body pose encoding for human activity recognition," *2020 25th International Conference on Pattern Recognition (ICPR)*, 2021.

[18] P. Russo, S. Ticca, E. Alati, and F. Pirri, "Learning to see through a few pixels: Multi Streams Network for extreme low-resolution action recognition," *IEEE Access*, vol. 9, pp. 12019–12026, 2021.

[19] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to Sequence Learning with Neural Networks," *Advances in Neural Information Processing Systems*, pp. 3104–3112, 2014.

[20] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder–decoder for statistical machine translation," *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014.

[21] D. Bahdanau, K. H. Cho, and Y. Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate," *Proceedings of ICLR*, 2015.

[22] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," *Proceedings of NAACL-HLT*, pp. 4171–4186, 2019.

[23] Hedu - Math of Intelligence. "Visual Guide to Transformer Neural Networks - (Episode 2) Multi-Head & Self-Attention," *YouTube*, Dec. 8, 2020 [Video file]. Available: https://www.youtube.com/watch?v=mMa2PmYJlCo. [Accessed: Dec. 12, 2021].

[24] A. Dosovitskiy , L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An Image Worth 16x16 Words: Transformers for Image

Recognition at Scale," in *International Conference on Learning Representation*, 2021.

[25] J. Deng, W. Dong, R. Socher, L. -J. Li, Kai Li and Li Fei-Fei, "ImageNet: A large-scale hierarchical image database," *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248-255, doi: 10.1109/CVPR.2009.5206848.

[26] A. Krizhevsky. Learning multiple layers of features from tiny images. *Tech Report*, 2009.

[27] Khurram Soomro, Amir Roshan Zamir and Mubarak Shah, UCF101: A Dataset of 101 Human Action Classes From Videos in The Wild, CRCV-TR-12-01, November, 2012.

X. Appendix

Code used to create the models that served as the basis for this paper can be found at: https://github.com/JamesWensel/TranformerActivityRecognition