National Technical University of Athens

Interdisciplinary Master's Programme in

Data Science and Machine Learning

Fourth Lab in the course of
Geospatial Big Data and Analytics

Video classification with RNN's and Transformers

Written by: Christos Nikou
ID: 03400146

email:christosnikou@mail.ntua.gr

May 29, 2022

1 Introduction

Continuing our research into Deep Learning in this Fourth lab we turn our attention to the problem of video classification. In particular, we are interested in comparing two different Neural Network architectures. A *Recurrent Neural Network (RNN)* with an additional gating mechanism, the *Gated recurrent unit (GRU)* and a *Transformer* consisting only of its *Encoder* part.

In detail, we train the two aforementioned models on a data-set consisting of 818 action videos, collected from YouTube, separated in five different categories according to the depicted action. The data-set we use in this Lab is a simplified version of the data-set UCF101 consisting of 13320 videos from 101 action categories. The action categories of the original data-set can be divided into five types: 1) Human-Object Interaction, 2) Body-Motion only, 3) Human-Human Interaction, 4) Playing Musical Instruments and 5) Sports. The simplified version of the data-set contains videos from the following five actions: 1) Cricket Shot, 2) Playing Cello, 3) Punch, 4) Shaving Beard and 5) Tennis Swing.

As we shall shortly see, the mechanism of the Transformer provides an alternative way of processing sequential data; like time series, videos (sequential frames), DNA sequences etc. The key difference of the Transformer architecture as compared to the various RNN architectures is that the former has the ability to process every element of the sequence in parallel. The parallel process of the sequence decreases the computational cost dramatically by giving more flexibility and space to other important operations during the training procedure of the model. Moreover, the *self-attention* mechanism of the Transformers allows the model to maintain the information of the whole sequence by rewarding elements of high importance.

2 The RNN model

2.1 The basic structure

In this subsection, before we move to the more complex architecture of GRU we give a short description of the RNN model which serves as the baseline of more complex networks used for operating on sequential data. According to [2] a recurrent neural network (RNN) is any network that contains a cycle within its network connections, meaning that the value of some unit is directly, or indirectly, dependent on its own earlier outputs as an input. The simplest form of RNN is referred to as Elman Network [1]. Although more complex architectures have been developed nowadays; these type of Networks capture the main theme of RNN's and serve the basis for the other architectures like *Long short-term memory (LSTM's)* and *Gated recurrent units (GRU's)*.

The main two differences of a recurrent neural network as opposed to the classic architecture of the *Feed forward neural networks (FFNN's)* is that the former 1) process one item at a time and 2) contain a recurrent link in their hidden units that augments the input to the computation at the hidden layer with the value of the hidden layer from the preceding point in time. Formally, the RNN takes as input a sequence of vectors $x_1, x_2, \ldots, x_t$ (features) and returns as an output a sequence $y_1, y_2, \ldots, y_t$ (predictions). A classic example that can be modeled by an RNN is the prediction of upcoming words in a sequence of words. In this case, the sequence of input $x_1, x_2, \ldots x_t$ consists of the words contained in a sentence and the output $y_1, y_2, \ldots, y_t$ consists of the prediction of the next words; if x_i is the i-th

word in the sentence then y_i corresponds to the prediction of the model for the word x_{i+1}. In Figure 1 we see a visual representation of this procedure.

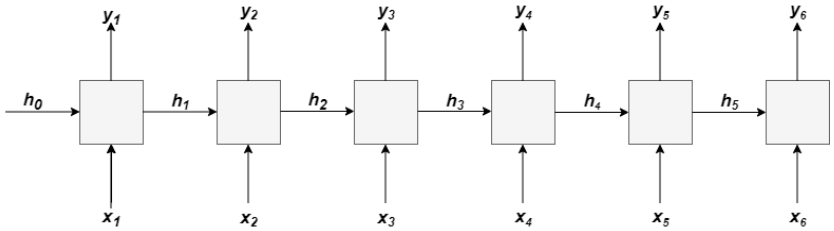

Figure 1: The sequential process of RNN's.

Figure 1 shows how an RNN process the elements of the sequence one by one. Now let's see in more detail the calculations that take place inside the hidden units (gray boxes). Denote by d_{in} the dimension of the input x_i and by d_h the dimension of the vectors h_i. Now, in the case of soft classification each output y_i consists of a probability vector where each coordinate j represents how likely is the next word to be the j-th word from the collection of all possible outcomes. Denote by d_{out} the number of all possible outcomes (set of words in our language example). Then a forward pass of the RNN proceeds as follows: We first initialize a vector $h_0 \in \mathbb{R}^{d_h \times 1}$ and three matrices $W \in \mathbb{R}^{d_h \times d_{in}}$, $U \in \mathbb{R}^{d_h \times d_h}$, $V \in \mathbb{R}^{d_{out} \times d_h}$. Then, the input vector x_1 is multiplied with W and h_0 with U; then we add the resulting vector in $\mathbb{R}^{d_h \times 1}$ to produce a vector $z_1 = Wx_1 + Uh_0 \in \mathbb{R}^{d_h \times 1}$. Then, $h_1 = g(z_1) = g\big(Wx_1 + Uh_0\big)$ and $y_1 = f(Vz_1)$, where $f, g : \mathbb{R} \to \mathbb{R}$ are two non linear activation functions[1]. In the case of a soft classification, f is the soft-max function given by

$$f(v_1, \ldots, v_{d_{out}}) = \left(\frac{e^{v_1}}{\sum_{i=1}^{d_{out}} e^{v_i}}, \ldots, \frac{e^{v_{d_{out}}}}{\sum_{i=1}^{d_{out}} e^{v_i}} \right). \tag{2.1.1}$$

Now, as shown in Figure 1 the hidden vector h_1 is passed two the second hidden layer and the same computation applies to generate the hidden vector h_2 and the output vector y_2.. We can think of the hidden vector h_i as the information captured from all the elements $x_1, \ldots, x_i$ of the sequence. This information is carried to the $i+1$ layer and it is combined with the next element x_{i+1} of the sequence through the $Wx_{i+1} + Uh_i$ operation. The part Wx_{i+1} corresponds to the new information added by the new input element x_{i+1} and the part Uh_i contains the information from all the preceding elements of the sequence. This procedure continues until all the elements of the sequence have been considered. In the following equations we summarize the general recursive update rule.

$$h_t = g\big(Uh_{t-1} + Wx_t\big) \tag{2.1.2}$$

$$y_t = f(Vh_t) \tag{2.1.3}$$

The weight matrices U, V, W are shared across all hidden blocks shown in Figure 1 during the forward pass and they are updated via *Backpropagation Through Time (BPTT)* (see [3],[4],[6]). To write the

[1]For a matrix $A \in \mathbb{R}^{n \times m}$ and a function $f : \mathbb{R} \to \mathbb{R}$, by $f(A)$ we denote the $\mathbb{R}^{n \times m}$ matrix resulting by applying f element-wise on A.

output sequence $y_1, \ldots, y_n$ more formally we can think of the generated combination vectors $z_1, \ldots, z_n$ produced by a function $F : \mathbb{R}^{d_{in}} \times \mathbb{R}^{d_h} \to \mathbb{R}^{d_h}$; i.e.

$$z_t = F(x_t, h_{t-1}) = Uh_{t-1} + Wx_t, \tag{2.1.4}$$

for every $1 \leq n$. Then, the output y_n can be calculated in terms of x_1, h_0 through the following recursive formula

$$y_i = V \cdot f(F(x_i, h_{i-1})) = V \cdot f(Uh_{i-1} + Wx_i) \tag{2.1.5}$$

$$= V \cdot f(Ug(F(x_{i-1}, h_{i-2})) + Wx_i). \tag{2.1.6}$$

Now, for the above architectures there are a few things that might cause them to be ineffective in real applications. The main problem with this architecture is that if the length of sequence is large; the recursive calculations in the hidden layers increase accordingly causing the model to forget information from past elements of the sequence. Another problem arises during the training procedure. Training in RNN models is essentially the same as in the case of FFNN's with the only exception that the computation of the gradients cannot be parallelized since the process of calculations is sequential. This property of RNN's makes them highly demanding when it comes to the computational resources. Even if the computation can be carried out; another problem that might appear is the *vanishing gradients problem (or exploding gradients).*

2.2 Stacked and Bidirectional RNN architectures

2.2.1 Stacked RNN's

As we saw in the previous subsection the basic RNN takes as input a sequence of vectors $x_1, \ldots, x_n$ and returns again a sequence $y_1, \ldots, y_n$ of items. For example, in the problem of predicting the next word the y_i's corresponds to the prediction word followed by $x_i's$. In the case of the stacked architecture the output sequence $y_1, \ldots, y_n$ is a sequence of real vectors that serve again as input sequence to another RNN architecture as shown in Figure 1. In Figure 2 you can see a visual representation of such architecture consisting of stacked layers as described in Subsection 2.1.

These stacked architectures have been proven to perform better than the simple ones. One reason for this success seems to be that the network includes representations at differing levels of abstraction across layers. However, as the number of stacks is increased the training costs rise quickly [2].

2.2.2 Bidirectional RNN's

As we stated before, one of the drawbacks of the model described in Subsection 2.1 is that for large input sequences the model forgets the information that was received in early stages and therefore; the output sequence in the last stages of the model would not contain any information from the early stages of the input sequence. In problems where the whole input sequence is known in beforehand, we can process the sequence in two ways: 1) one way is to process with an RNN the sequence as $x_1 \to x_2 \to \cdots \to x_{n-1} \to x_n$ and 2) as $x_n \to x_{n-1} \to \cdots \to x_2 \to x_1$. RNN architectures that use this two-way sequential processing are called bidirectional.

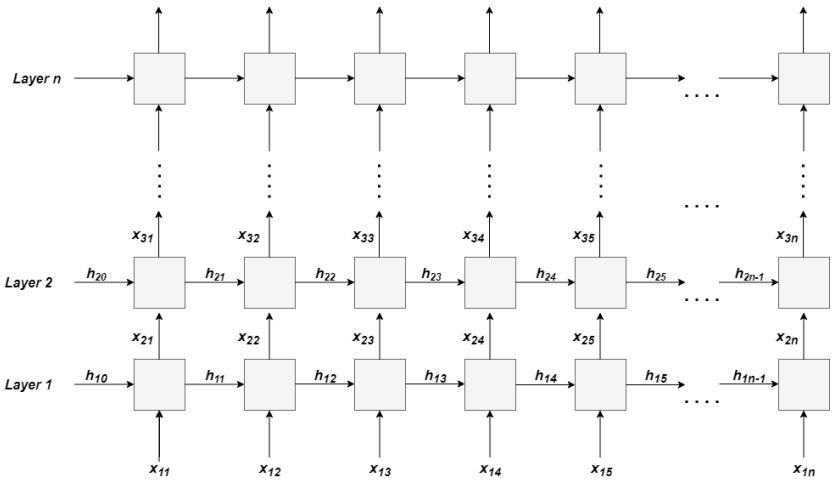

Figure 2: The architecture of a stacked RNN.

The forward pass in the bidirectional case is rather straightforward. We first pass the sequence from a classic RNN architecture as shown in Figure 1 in two ways; in the first one the input is processed from the start to end, and in the second one from the end to start. The normal hidden state at time t for the classical forward pass is a vector $h_t^f = RNN_{forward}(x_1, \ldots, x_t) \in \mathbb{R}^{d_h \times 1}$, representing everything the network has learned from the sequence so far. Similarly, if we process the sequence in reversed order; at time t the hidden state would be a vector $h_t^b = RNN_{backward}(x_t, \ldots, x_n) \in \mathbb{R}^{d_h \times 1}$, representing everything the network has learned about the sequence from t to the end of the sequence. Having calculated the hidden states h_t^f, h_t^b for all the time steps t; we define the hidden state at t in the bidirectional case to be the concatenated vector

$$h_t = h_t^f \oplus h_t^b \in \mathbb{R}^{2d_h \times 1}. \tag{2.2.1}$$

Now, the vector h_t in (2.2.1) contains the same portion of information from the early stages as the late stages of the sequence. Now, the vectors h_t are multiplied by a matrix $V \in \mathbb{R}^{d_{out} \times 2d_h}$ to produce the output sequence y_t; which in turn can be fed to another bidirectional RNN layer or serve as an output for a soft-classification operation. However, these architectures are more restricted as opposed to the classical architectures described in Subsection 2.1 as they presuppose the knowledge of the whole sequence and hence can only be used in specific situations. For example, it is impossible to use a bidirectional RNN in order to make predictions for a continuous stream of data. Another important drawback of these architectures is the high computational cost. The main reason for this is that the forward propagation requires both forward and backward recursions in bidirectional layers and that the backpropagation is dependent on the outcomes of the forward propagation. Hence, gradients will have a very long dependency chain. In practice bidirectional layers are used very sparingly and only for a narrow set of applications, such as filling in missing words, annotating tokens (e.g., for named entity recognition), and encoding sequences wholesale as a step in a sequence processing pipeline (e.g., for machine translation).

2.3 RNN's with gated recurrent units

To tackle the problems that appear in the RNN's and the bidirectional RNN's more sophisticated architectures have been developed for sequential processing. One of these architectures is the *Recurrent Neural Network with gated recurrent units (GRU).* GRU's are trying to eliminate the vanishing gradient problem appearing in classical architectures and at the same time maintain the most important information that can be extracted from the sequence.

To solve the vanishing gradient problem of a standard RNN, GRU uses, so-called, *update gate* and *reset gate.* Basically, these are two vectors which decide what information should be passed to the output. In this way, GRU's can maintain the relevant information needed for larger steps as opposed to the previous architectures. As in the case of the simple RNN's let's break down the operations handled in the hidden state. Suppose that the input sequence consists of the vectors $x_1, \dots, x_n$, where $x_i \in \mathbb{R}^{d_{in} \times 1}$. Then, during a time step t the *update gate* z_t is given by

$$z_t = \sigma\big(W^{(z)} x_t + U^{(z)} h_{t-1}\big), \tag{2.3.1}$$

where $\sigma : \mathbb{R} \to (0,1)$ is the sigmoid function defined by $\sigma(x) = \frac{1}{1+e^{-x}}$, $W^{(z)} \in \mathbb{R}^{d_h \times d_{in}}$, $U^{(z)} \in \mathbb{R}^{d_h \times d_h}$ are two matrices consisting of learnable parameters and $h_{t-1} \in \mathbb{R}^{d_h \times 1}$ is the vector passed from the previous hidden layer representing the information gained from the points $x_1, \dots, x_{t-1}$. The *reset gate* is given by

$$r_t = \sigma\big(W^{(r)} x_t + U^{(r)} h_{t-1}\big). \tag{2.3.2}$$

Notice that z_t, r_t are calculated by the same operation. However, in the following equations it will be clear why the matrices $W^{(z_t)}, U^{(z_t)}$ are different from $W^{(r_t)}, U^{(r_t)}$. Now, the reset gate is used to determine the amount of information to be passed to the next stage and the amount of information to be discarded. First we calculate the *current memory content* given by

$$h'_t = \tanh\big(W x_t + r_t \odot U h_{t-1}\big), \tag{2.3.3}$$

where by $x \odot y$ we denote the element-wise product of the two vectors x, y[2], and $W \in \mathbb{R}^{d_h \times d_{in}}$, $U \in \mathbb{R}^{d_h \times d_h}$ are two matrices consisting of learnable parameters. The vector h_t which holds the information for the current unit and passes it down to the network is given by

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot h'_t. \tag{2.3.4}$$

Now lets interpret the above calculations one by one by starting from the reset gate r_t in (2.3.2). As it easily seen, $W^{(r)} x_t + U^{(r)} h_{t-1}$ is a vector in $\mathbb{R}^{d_h \times 1}$. Suppose the extreme case where all the coordinates of $W^{(r)} x_t + U^{(r)} h_{t-1}$ consist of large negative values; then the application of the sigmoid function will zero out the coordinates of $W^{(r)} x_t + U^{(r)} h_{t-1}$ and hence r_t will be close to 0. Now, the term $r_t \odot U h_{t-1}$ in (2.3.3) will be negligible and hence $h'_t \approx \tanh\big(W x_t\big)$, which means that h'_t will contain information only from the current point x_t. Now, if it happens at the same $z_t \approx 0$ then we observe that the information h_t in (2.3.4) which will be passed to the next unit will be approximately equal to $h'_t \approx \tanh(W x_t)$ and hence the network will forget all the information received up to time t. In similar manner, if $z_t \approx 1$ then $h_z \approx h_{t-1}$ which means that the element x_t does not contain any valuable

[2]This is known as the Hadamard product.

information to be carried out. Another way to think of the equation in (2.3.4) is as a weighted average of the past information h_{t-1} and the current memory content h'_t. Indeed, since the vector z_t is produced after the application of the sigmoid function; then all of its coordinates will be real numbers in $(0, 1)$. Therefore, we can think of z_t as the proportion of h_{t-1} to be carried out and $1 - z_t$ as the corresponding proportion for current memory content h'_t. In Figure 3[3] you can see a visual representation of the preceding operations.

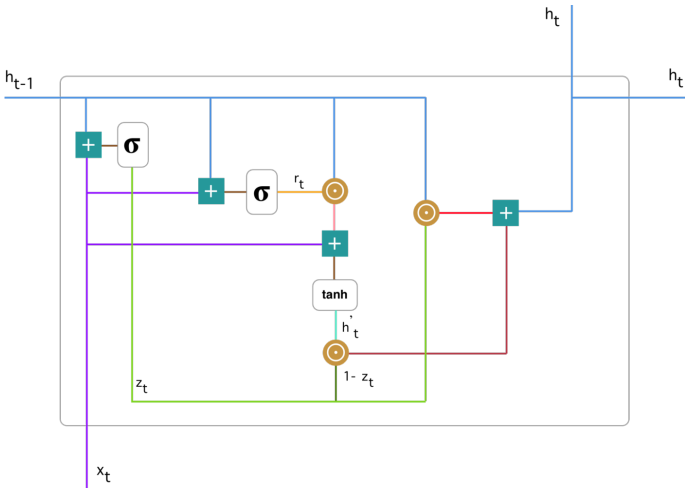

Figure 3: A hidden Gated Recurrent Unit.

Having explained how the hidden vector h_t is calculated; then the forward pass is essentially the same as in case of the classic RNN's. First, the vector $h_0 \in \mathbb{R}^{d_h \times 1}$ and the matrices $W^{(z)}, W^{(r)}, W \in \mathbb{R}^{d_h \times d_{in}}$, $U^{(z)}, U^{(r)}, U \in \mathbb{R}^{d_h \times d_h}$ are initialized. Then by using (2.3.1), (2.3.2), (2.3.3) the vectors z_t, r_t, h'_t are calculated. Finally, the vector h_1 is calculated by (2.3.4) and it is passed to the next unit. An output y_1 can be considered also at this point by initializing a matrix $V \in \mathbb{R}^{d_{out} \times d_h}$ and calculate $y_1 = f(Vh_1)$ for some non-linear activation function f. Now, as in the case of the RNN's y_1 can be used for a soft-classification or passed as an input to the first unit of another GRU. This procedures continues until the whole sequence $x_1, \dots x_n$ has been passed through the network. The matrices $W^{(z)}, W^{(r)}, W \in \mathbb{R}^{d_h \times d_{in}}$, $U^{(z)}, U^{(r)}, U \in \mathbb{R}^{d_h \times d_h}$ and $V \in \mathbb{R}^{d_{out} \times d_h}$ are shared across all units. The adaptation of the parameters in the above matrices, as in the case of classic RNN's, is done via *Backpropagation Through Time (BPTT)*. BPTT follows the reverse direction of the forward pass and calculates the gradient of a loss function $L(y, \hat{y})$ with respect to the entries of the aforementioned matrices. In $L(y, \hat{y}); y = (y_1, \dots, y_n)$ is the output sequence and $\hat{y} = (\hat{y_1}, \dots, \hat{y_n})$ is the target sequence.

As we saw in the calculations above the gate mechanism of GRU's filters the information by penalizing all the irrelevance might contained from the previous calculations. However, one of the main problems of these architectures still remains, i.e. the forward pass cannot be parallelized and hence the training of such models might take long. In Section 3 we will study the architecture of the Transformers who try to tackle this problem by maintaining at the same time the information filtering aspect of GRU's. In the next subsection we examine the performance of GRU's to the problem of video classification.

[3]Image and information about GRU's taken from this nice article.

2.4 GRU - Appication to Video classification

In this subsection we use a GRU architecture for the classification of videos from the data-set UCF101. We use only a subset of the whole data-set consisting of 818 videos from the following five categories: 1) Cricket Shot, 2) Playing Cello, 3) Punch, 4) Shaving Beard and 5) Tennis Swing. Splitting the data-set we use a total of 594 videos for training and the remaining 224 videos for validation. The code was developed in Python using PyTorch. A video consists of a sequence of frames (images). We can represent the frames of a video as a $4D-$tensor of shape $F \times C \times H \times W$, where F is the number of frames (images), C is the number of channels (RGB), W the width of the image and H the height. Before we feed the frames to the GRU model we use ResNet18 to transform each frame of the video to a $\mathbb{R}^{512\times 1}$ vector. Therefore, each video is represented by a sequence $x_1, \ldots, x_n$, where $x_i \in \mathbb{R}^{512\times 1}$, and n is the number of frames. In the case of batch training, the input is a tensor $3D-$tensor of shape $N \times M \times 512$, where N is the batch size and M is the number of frames of the video with the maximum number of frames out of the N videos. In the implementation N was equal to 16. The GRU model that we used consists of 2 hidden layers and a 32-dimensional hidden state, i.e. $d_h = 32$. The second hidden layer output a sequence of elements $h_1, \ldots, h_n \in \mathbb{R}^{32\times 1}$, where n denotes the number of frames in a particular video. From these outputs we keep only the last element h_n of the sequence. Now, h_n is a vector in $\mathbb{R}^{32\times 1}$. For the classification purposes we use a small FFNN. We first map the output vector with a linear transformation to a vector $y_n \in \mathbb{R}^{16\times 1}$, i.e. $y_n = Wh_n$. where $W \in \mathbb{R}^{16\times 32}$. Then we pass the vector y_n through a ReLU followed by a linear transformation mapping the vector y_n from $\mathbb{R}^{16\times 1}$ to $\mathbb{R}^{5\times 1}$, where 5 corresponds to the number of distinct categories. The CrossEntropy loss function handles the softmax operation to produce a probability vector of length 5. In Table 1 we can see a summary of the results for the classification.

Table 1: Classification report of CNN - GRU.

Category	**Precision**	**Recall**	**F1-Score**	**Support**
CricketShot	1	0.82	0.90	49
Playing Cello	1	0.98	0.99	44
Punch	0.95	0.97	0.96	39
Shaving Beard	0.95	0.95	0.95	43
Tennis Swing	0.84	1	0.92	49
Accuracy			0.94	224
Macro Avg.	0.95	0.94	0.94	224
Weighted Avg.	0.95	0.94	0.94	224

From Table 1 we see that the performance of the model is extremely high and in perfect accordance when it comes to the metrics of accuracy, macro average and weighted average. To elaborate on the results with respect to precision, recall and F1-score let us state for completion the definitions of these metrics. We denote by FP_i, TP_i the *false-positive* and *true-positive* number of predictions made by the model for the class i. In our case $i = 0$ corresponds to CricketShot, $i = 1$ to PlayingCello etc. Then,

the precision P_i of the class i is given by

$$P_i = \frac{TP_i}{TP_i + FP_i}. \tag{2.4.1}$$

Precision represents the proportion of the actual correct identifications out of the total positive identifications made by the classifier. Similarly, by FN_i we denote the *false-negative* predictions made by the model for the class i. Then, the recall R_i of the class i is given by

$$R_i = \frac{TP_i}{TP_i + FN_i}. \tag{2.4.2}$$

Recall represents the proportion of the actual positives identified by the model. The F1-Score F_i for the class i is defined as the harmonic mean of recall and precision, i.e.

$$F_i = 2 \cdot \frac{P_i R_i}{P_i + R_i}. \tag{2.4.3}$$

From Table 1 we see that the precision of CricketShot and PlayingCello actions is 1 which means that the model did not confuse these two categories with another action. The lowest precision is 0.84, corresponding to Tennis Swing. For recall, we see that 0.82 is the lowest value corresponding to CricketShot. In the confusion matrix below we can see the exact distribution of predictions made by the classifier.

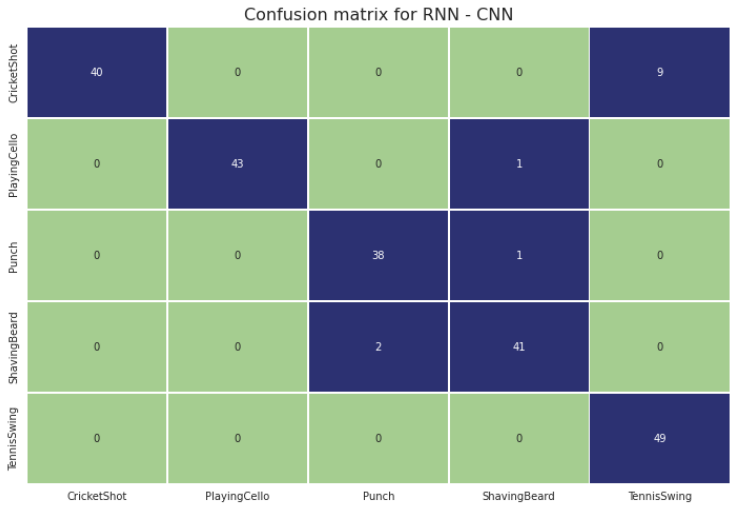

Figure 4: Confusion matrix of CNN - RNN model.

3 Transformers

3.1 The Architecture

As we discussed in the previous section the sequential dependence between the elements in the RNN architecture makes it impossible for the computation to be carried out in parallel; and thus training such models can be difficult. Moreover, although the gates that we used in GRU's mitigated the problem of information loss for large sequences they did not completely solve it. These considerations led to the development of transformers [5] - an approach to sequence processing that eliminates recurrent connections and returns to architectures reminiscent of the fully connected networks.

Transformers map sequences of input vectors $x_1, \ldots, x_n$ to sequence of output vectors $y_1, \ldots, y_n$ of the same length. The key innovation is the self-attention operation taking place in the self-attention layers of the transformer. In Figure 5 we see a visual representation of these layers.

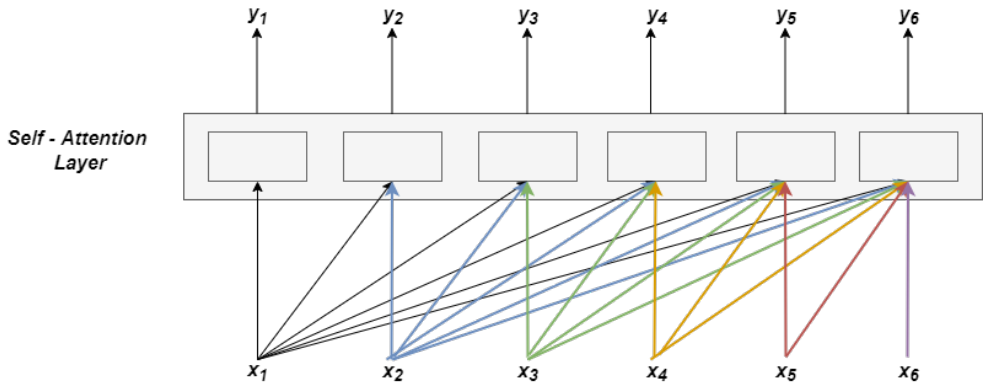

Figure 5: The self-attention layer of a transformer.

From the previous figure we can see that the i-th self-attention unit accepts as an input all the elements of the sequence up to the x_i vector, i.e. the elements $x_1, \ldots, x_i$.

We now describe how self-attention works and how it fits into larger transformer blocks. Suppose that the input sequence consists of the vectors $x_1, \ldots, x_n$. Each input x_i has three different roles to play:

(i) The first one is its part that contributes to the information when combined with the preceding vectors $x_1, \ldots, x_{i-1}$ (*current focus of attention).* We will refer to this role as *query.*

(ii) The second one is its role as a preceding input to the future attention units. For example, in Figure 5 the input x_2 serves as input to $3, 4, 5, 6$, attention units. We will refer to this role as *key.*

(iii) And finally, it's the role as a *value* used to compute the output for the current focus of attention.

Formally, if our input sequence vectors lie in $\mathbb{R}^{d\times 1}$ then the *query* of x_i is given by $q_i = W^Q x_i$, where $W^Q \in \mathbb{R}^{d\times d_k}$, the *key* by $k_i = W^K x_i$, where $W^K \in \mathbb{R}^{d\times d_k}$ and finally the *value* by $v_i = W^V x_i$, where $\in \mathbb{R}^{d\times d_v}$. One way to think of these three roles is as being projections of the input vector x_i; each one serving for a different purpose. The weight matrices W^Q, W^K and W^V are shared across all units of the self-attention layer.

Having explained how these three vectors are generated we proceed by explaining what is happening inside a self-attention unit. For simplicity, we ignore for the moment that the following computation can be carried out in parallel and we proceed by operating in a sequential way the input vectors. First the vector x_1 gets multiplied by the matrices W^Q, W^K, W^V to produce the vectors q_1, k_1 and v_1. Then, the score $\alpha_{11} = \frac{\langle q_1, k_1 \rangle}{\sqrt{d_k}}$[4] is calculated and the output vector y_1 is given by $y_1 = v_1$. Moving to the second input vector x_2 we calculate the scores α_{21} and α_{22} by

$$\alpha_{21} = \frac{\langle q_2, k_1 \rangle}{\sqrt{d_k}}, \quad \alpha_{22} = \frac{\langle q_2, k_2 \rangle}{\sqrt{d_k}}. \tag{3.1.1}$$

Now, the score α_{21} measures how relevant is the *query* part q_2 of x_2 with the *key* part k_1 of x_1. For example, if $\langle q_2, k_1 \rangle = 0$ then q_2, k_1 will be perpendicular. Now, from α_{21}, α_{22} we create a probability vector (p_{21}, p_{22}), where p_{21} contains the portion of similarity of q_2 and k_1 compared to the portion of similarity of q_2 and k_2. To calculate p_{21} and p_{22} we use the soft-max function:

$$p_{21} = \frac{e^{\alpha_{21}}}{e^{\alpha_{21}} + e^{\alpha_{22}}}, \quad p_{22} = \frac{e^{\alpha_{22}}}{e^{\alpha_{21}} + e^{\alpha_{22}}}. \tag{3.1.2}$$

Now, y_2 is the weighted average of v_1 and v_2 with weights determined by p_{21}, p_{22}, i.e. $y_2 = p_{21}v_1 + p_{22}v_2$. Generalizing the above process; in the i-th unit the input vector x_i gets multiplied by W^Q, W^K, W^V to produce the vectors q_i, k_i and v_i. Then, the score α_{ij} is given by $\alpha_{ij} = \frac{\langle q_i, k_j \rangle}{\sqrt{d_k}}$ for every $1 \leq j \leq i$. The probabilities p_{ij} are now given by

$$p_{ij} = \frac{e^{\alpha_{ij}}}{\sum_{k=1}^{i} e^{\alpha_{ik}}}, \quad \text{for every } 1 \leq j \leq i. \tag{3.1.3}$$

Then, the output vector y_n is the weighted average $y_n = \sum_{k=1}^{i} p_{ik} v_k$. Now observe that the previous computations can be performed in parallel. During the forward pass, the vectors q_i, k_i and v_i are computed for every $1 \leq i \leq n$, independently, by multiplying every x_i with the matrices W^Q, W^K and W^V. From now on, since the units in a self-attention layer are independent, we have everything we need to calculate the output y_i on the i-th self-attention unit. In practice, the above calculations are performed by packing the whole input sequence into a $2D$-tensor X of size $S \times D$. S corresponds to the sequence length and D to the dimension of the input vectors. To calculate for every element x_i its *query, key* and *value* we simply multiply the tensor X with the initialized $2D$-tensors W^Q, W^K of size $D \times D$ and W^V of size $D \times D$, respectively. Therefore, the *query* of the element x_i is the i-th row of the tensor $Q = X \cdot W^Q$, the *key* the i-th row of $K = X \cdot W^K$ and the *value* the i-th row of $V = X \cdot W^V$. One could easily be confused by the change in the order of the multiplication of $Q = X \cdot W^Q, K = X \cdot W^K$ and $V = X \cdot W^V$ as opposed to the case where we handled the computation by passing the elements of the sequence one by one leading to $q_i = W^Q x_i, k_i = W^K x_i$ and $v_i = W^V x_i$. The reason why there is no problem in changing the order is because the tensors W^Q, W^K, W^V will adjust their values accordingly during backpropagation in either case. Now, to calculate the scores we simply calculate the product $Q \cdot K^T$, where K^T is the transpose matrix of K. However, there is a small problem in this calculation. The rows of Q, K correspond to the *queries* $q_i \in \mathbb{R}^{D \times 1}$ and *keys* $k_i \in \mathbb{R}^{D \times 1}$ for each element of the sequence. Therefore, by multiplying Q with K^T we get that

[4] By $\langle x, y \rangle$ we denote the inner product of the vectors x, y.

$$Q \cdot K^T = \begin{pmatrix} q_1^T \\ q_2^T \\ q_3^T \\ \vdots \\ q_{S-1}^T \\ q_S^T \end{pmatrix} \cdot \begin{pmatrix} k_1 & k_2 & k_3 & \dots & k_{S-1} & k_S \end{pmatrix}$$

$$= \begin{pmatrix} q_1^T k_1 & q_1^T k_2 & q_1^T k_3 & \dots & q_1^T k_{S-1} & q_1^T k_S \\ q_2^T k_1 & q_2^T k_2 & q_2^T k_3 & \dots & q_2^T k_{S-1} & q_2^T k_S \\ q_3^T k_1 & q_3^T k_2 & q_3^T k_3 & \dots & q_3^T k_{S-1} & q_3^T k_S \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ q_{S-1}^T k_1 & q_{S-1}^T k_2 & q_{S-1}^T k_3 & \dots & q_{S-1}^T k_{S-1} & q_{S-1}^T k_S \\ q_S^T k_1 & q_S^T k_2 & q_S^T k_3 & \dots & q_S^T k_{S-1} & q_S^T k_S \end{pmatrix}.$$

Now, the elements that we care about are the elements $q_i^T k_j$ for every $1 \leq j \leq i$, since these inner products correspond to the score for each query value to every key value, *excluding those that follow the query.* By observing that the elements $q_i^T k_j$ for $1 \leq j \leq i$ correspond to the elements in the lower-triangular portion of the matrix we can zero out the upper-triangular part with the softmax application by setting these values to $-\infty$. Therefore, we slightly modify $Q \cdot K^T$ and write

$$Q \cdot K^T = \begin{pmatrix} q_1^T k_1 & -\infty & \dots & -\infty \\ q_2^T k_1 & q_2^T k_2 & \dots & -\infty \\ \vdots & \vdots & \ddots & \vdots \\ q_S^T k_1 & q_S^T k_2 & \dots & q_S^T k_S \end{pmatrix}.$$

Now the output of the self-attention layer is a $2D$-tensor of size $S \times D$ given by applying the soft max function row-wise to $Q \cdot K^T$, normalizing by $\sqrt{D}$ and finally multiplying with V. In other words the output of the self-attention layer is given by

$$SelfAttention(Q, K, V) = softmax\left(\frac{Q \cdot K^T}{\sqrt{D}}\right)V. \tag{3.1.4}$$

3.2 Transformer Blocks

The self-attention calculation lies at the core of what's called a transformer block, which, in addition to the self-attention layer, includes additional feedforward layers, residual connections, and normalizing layers. In Figure 6 we see a visual representation of these operations carried out in the transformer block. The input and output dimensions of these blocks are matched so they can be stacked just as was the case for stacked RNNs. The forward pass through a transformer block is the following: first the input matrix $X \in \mathbb{R}^{S \times D}$[5] passes through the self-attention layer and an output $SelfAttention(X)$ of same dimensions according to (3.1.4) is produced.

[5] S is the sequence length, D the dimension of the sequence elements.

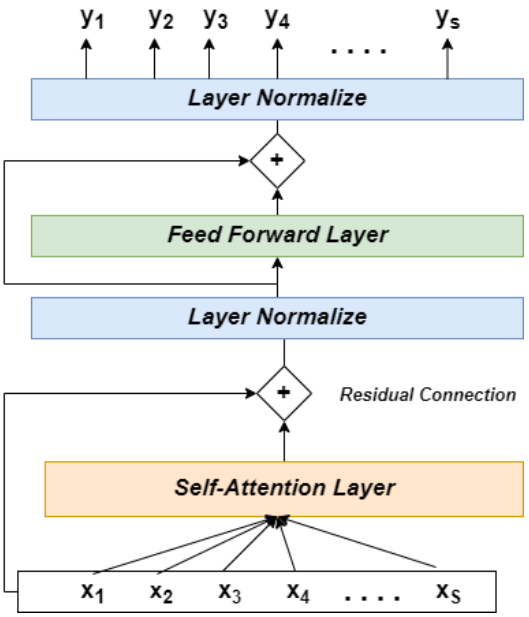

Figure 6: The operations carried out in a transformer block

Then, the residual connection takes place, i.e. $X + SelfAttention(X)$. The next operation is layer normalization; the output vector $Y = X + SelfAttention(X)$ is an $S \times D$ matrix. In the normalization operation we calculate the mean value for each row of Y and the respective standard deviation. In particular, if we write the matrix $Y = X + SelfAttention(X)$ as

$$Y = \begin{pmatrix} Y_{11} & Y_{12} & \dots & Y_{1D} \\ Y_{21} & Y_{22} & \dots & Y_{2D} \\ \vdots & \vdots & \ddots & \vdots \\ Y_{S1} & Y_{S2} & \dots & Y_{SD} \end{pmatrix}.$$

Then, for every for every row we calculate the mean and standard deviation by

$$\mu_i = \frac{1}{D}\sum_{j=1}^{D} Y_{ij}, \quad \sigma_i = \sqrt{\frac{1}{D}\sum_{j=1}^{D}(Y_{ij} - \mu_i)^2},$$

for every $1 \leq i \leq S$. The operation of layer normalization is described by $LayerNorm(Y) = \gamma\hat{Y} + \beta$, where

$$\hat{Y} = \begin{pmatrix} (Y_{11} - \mu_1)/\sigma_1 & (Y_{12} - \mu_1)/\sigma_1 & \dots & (Y_{1D} - \mu_1)/\sigma_1 \\ (Y_{21} - \mu_2)/\sigma_2 & (Y_{22} - \mu_2)/\sigma_2 & \dots & (Y_{2D} - \mu_2)/\sigma_2 \\ \vdots & \vdots & \ddots & \vdots \\ (Y_{S1} - \mu_S)/\sigma_S & (Y_{S2} - \mu_S)/\sigma_S & \dots & (Y_{SD} - \mu_S)/\sigma_S \end{pmatrix},$$

and γ, β are learnable parameters. Next, the resulting vector $LayerNorm(Y)$ is fed to a *Feed Forward Neural Network (FFNN)* followed by another residual connection and a normalization operation. The FFNN operation acts on every element of the sequence separately. The operations used in the FFNN in [5] are summarized as follows for a sequence element $x_i \in \mathbb{R}^{D\times 1}$:

$$FFFN(x_i) = ReLU\big(x_i W_1 + b_1\big) \cdot W_2 + b_2,$$

where $W_1 \in \mathbb{R}^{D \times d_{ff}}$ (in [5] $d_{ff} = 2048$) and $W_2 \in \mathbb{R}^{df_{ff} \times D}$. Below we summarize the forward pass of an input matrix X of size $S \times D$ through a transformer block:

$$Z = LayerNorm(X + SelfAttn(X)) \tag{3.1.5}$$

$$y = LayerNorm(Z + FFNN(Z)). \tag{3.1.6}$$

3.3 Multi - Head Attention

According to [5] instead of performing a single attention function with D-dimensional keys it has been experimentally proven to be more beneficial to linearly project the queries, keys and values h times with different, linear projections to D_k, D_k and D_v dimensions, respectively. This situation is illustrated in Figure 9.

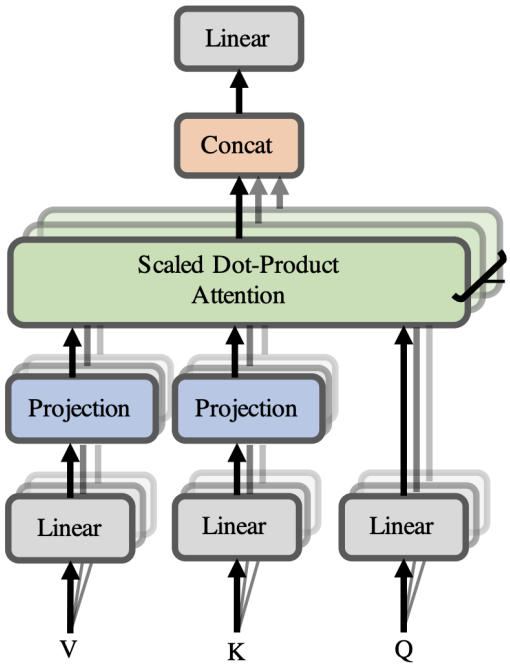

Figure 7: Multi-head Attention.

On each of these projected versions of queries, keys and values the self-attention function is applied in parallel, yielding D_v-dimensional output values. These are concatenated and once again projected, resulting in the final values. More formally, the operation of multi - head attention can be described as follows:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h) \cdot W^0, \tag{3.1.7}$$

where $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$ and $W_i^Q \in \mathbb{R}^{D \times D_K}$, $W_i^K \in \mathbb{R}^{D \times D_K}$, $W_i^V \in \mathbb{R}^{D \times D_v}$ and $W^0 \in \mathbb{R}^{hd_v \times D}$.

3.4 Application to Video Classification

Having described the main structure of the transformer we now apply this model to the classification problem described in Subsection 2.4 and compare the results of these two models. As in the case of 2.4 we again use ResNet18 to represent a video as a sequence of S elements each one of dimension $D = 512$. The encoder part of the transformer consists of 8 heads and 6 sub-encoder layers each of one having the structure depicted in Figure 6. The size of the hidden layer that we used in the feed forward neural network is $d_{ff} = 1024$. Moreover, since the transformer model contains no recurrence and no convolution, in order for the model to make use of the order of the sequence, one must inject some information about the relative or absolute position of the elements of the sequence. To this end, a positional encoding is used to the input embeddings at the bottoms of the encoder stacks. The positional encodings have the same dimension D as the embeddings, so that the two can be summed. The operation of positional encoding is given by

$$\mathrm{PE}_{(pos,2i)} = \sin(pos/10000^{2i/D}), \quad \mathrm{PE}_{(pos,2i+1)} = \cos(pos/10000^{2i/D}), \tag{3.4.1}$$

where pos is the position and i is the dimension. Training for 20 epochs with learning rate 10^{-5} and batch size equal to 24 we get the following graph of training-validation loss.

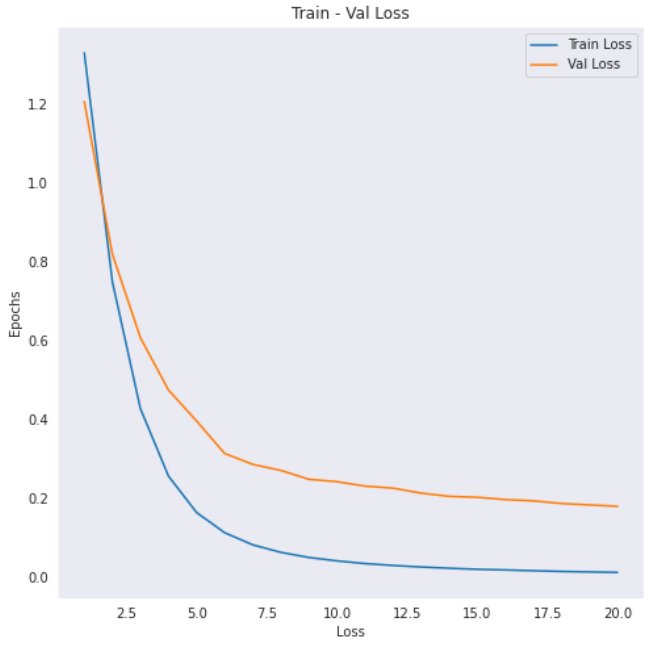

Figure 8: Training and validation loss of the encoder.

In the following table we summarize the results obtained in the validation set by the classification report.

Table 2: Classification report of Transformer Encoder.

Category	Precision	Recall	F1-Score	Support
CricketShot	0.90	0.90	0.90	49
Playing Cello	1	0.95	0.98	44
Punch	0.95	0.95	0.95	39
Shaving Beard	0.96	1	0.98	43
Tennis Swing	0.90	0.90	0.90	49
Accuracy			0.94	224
Macro Avg.	0.94	0.94	0.94	224
Weighted Avg.	0.94	0.94	0.94	224

Below we see the corresponding confusion matrix obtained from Table 2.

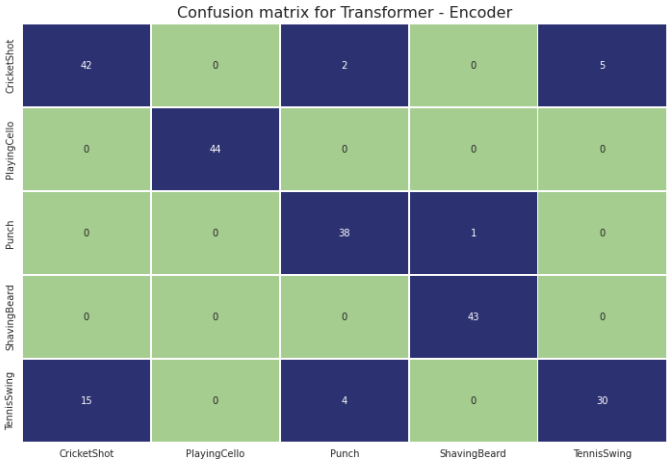

Figure 9: Training and validation loss of the encoder.

As we can see from the table above the transformer-encoder model achieves similar performance with the rnn-gru model. Both models have the same scores as opposed to the accuracy, macro average and weighted average. In the next section we compare the two models by training them and monitoring the results for more than one time.

4 Comparing the models

In this section we present the results of the two models for the video classification problem. We train both models for five times and monitor their training time as well as their scores for the accuracy, macro and weighted average. You can see a summary of the comparison in the following table.

Table 3: Comparing the models for five iterations.

Model	Accuracy (Avg.)	Macro (Avg.)	Weighted (Avg.)	Training Time (Avg.)
RNN - GRU	0.81	0.80	0.79	4 min 11:40 sec
Encoder	0.92	0.92	0.92	2 min 15:08 sec

From the above table we observe that the superiority of the Transformer - Encoder is evident compared to the RNN - GRU model. In all three cases the score of the Encoder is 0.10 higher and as for the training time it takes almost 2 minutes less.

5 Answering questions

5.1 Question 1

In 2.2.2 we described how bidirectional RNN operate. Their purpose is to to mitigate the information loss from the early stages due to the large size of the input sequence. Bidirectional RNN's train two instead one RNN on the input sequence. The first on the input sequence as-is and the second on a reversed copy of the input sequence. This can provide additional context to the network and result in faster and even fuller learning on the problem. However, bidirectional architectures can be used in problems where all timesteps of the input sequence is available. Therefore, the use of a bidirectional RNN architecture in order to make online predictions to continuous streams of videos it is not possible since we don't have access to the whole input sequence. In other words, we don't know the all the elements of the sequence in beforehand to perform the reverse forward pass.

5.2 Question 2

In section 2.4 we defined the precision, recall and F1-score; given in (2.4.1),(2.4.2) and (2.4.3), respectively. As we mentioned before, precision represents the proportion of the actually correct identifications out of the total positive identifications made by the model whereas Recall represents the proportion of the actual positives identified by the model. In applications where a false positive prediction has vital consequences the best model would be the one having the *highest recall* as possible. Indeed, ideally one would ask to have $FP \to 0$. Writing down the formula for the recall

$$R = \frac{TP}{TP + FN},$$

we observe that $R \nearrow 1$ as $FN \searrow 0$ and hence; the model having the highest recall has the fewer false positives. Some cases that having the model with the highest recall is of vital importance is for example the monitoring of suspicious behaviors in public places such as malls, public transportation,

sports facilities etc. The diagnosis whether a cancer is benign or malignant or whether a driver is about to fall asleep or not.

5.3 Question 3

One of the main problems of RNN's is that they exhibit rapid fluctuations when it comes to the values of the gradients. These rapid changes usually lead to vanishing gradients when the values are small or to exploding gradients for large values. In the case where the difference between the training and the validation loss remains large and the values of the gradients are high it a sign of the exploding gradients problem. One way to tackle this problem in RNN architectures is to reduce the size of the input sequence. Having a large sequence size results to many multiplications involved during the back-propagation. If these values are all greater than 1 then the derivative with respect to a parameter will tend to infinity. Therefore, reducing the size of the input sequence will reduce the calculations during backpropagation and might resolve the problem of exploding gradients. Another modification that can be done is based on the technique of gradient clipping. Gradient clipping involves forcing the gradient values to a specific minimum or maximum value if the gradient exceeded an expected range.

References

[1] Jeffrey L Elman. “Finding structure in time”. In: *Cognitive science* 14.2 (1990), pp. 179–211.

[2] Dan Jurafsky. *Speech & language processing*. Pearson Education India, 2000.

[3] Michael C Mozer. “A focused backpropagation algorithm for temporal”. In: *Backpropagation: Theory, architectures, and applications* 137 (1995).

[4] AJ Robinson and Frank Fallside. *The utility driven dynamic error propagation network*. University of Cambridge Department of Engineering Cambridge, 1987.

[5] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017).

[6] Paul J Werbos. “Generalization of backpropagation with application to a recurrent gas market model”. In: *Neural networks* 1.4 (1988), pp. 339–356.