LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN
Department "Institut für Informatik"
Lehr- und Forschungseinheit Medieninformatik
Prof. Dr. Heinrich Hußmann

Bachelorarbeit

Action Quality Assessment of Trampoline Routines

Niklas Lühr
n.luehr@campus.lmu.de

Bearbeitungszeitraum: 04.01.2021 bis 24.05.2021
Betreuer: Stefan Langer
Verantw. Hochschullehrer: Prof. Dr. Andreas Butz

Abstract

The goal of action quality assessment (AQA) is to assess the quality of human action, e.g. in sports or surgical tasks, based on video data. A problem with current AQA datasets is that they are all very small, which makes it hard to learn meaningful features with a deep learning approach. However, a well-engineered AQA system could be a great help in training and assessing athletes or medical students and giving them feedback. Moreover, it could aid the judges to reduce subjectiveness in important competitions. In this work, I propose an AQA framework for the yet unstudied discipline of trampoline of the AQA-7 dataset, which contains 83 trampoline videos. I base my work on the C3D-LSTM model, which uses a pre-trained 3D convolutional neural network (C3D) to extract features and an LSTM to aggregate them. At first, I explain why these components are well suited for the task of AQA. After that, I test more sophisticated feature extractors as well as LSTM variations. As key modifications, I employ temporal augmentation by shifting the start frame of the video and spatial augmentation by applying geometric distortions. My final model achieves a Spearman's rank correlation coefficient of 0.8485. This shows that even very small datasets can be utilized for AQA tasks with sufficient augmentation and the help of pre-trained feature extractors. The code is available at `https://github.com/niklasluehr/tramp-aqa`.

Zusammenfassung

Das Ziel von Action Quality Assessment (AQA) besteht darin, die Qualität menschlicher Aktionen, z. B. im Sport oder bei chirurgischen Aufgaben, anhand von Videodaten zu bewerten. Ein Problem von aktuellen AQA-Datensätzen ist, dass sie alle sehr klein sind, was es schwierig macht, sinnvolle Features mit einem Deep Learning Ansatz zu erlernen. Ein ausgereiftes AQA System könnte jedoch eine große Hilfe darstellen, um Sportler oder Medizinstudenten zu trainieren, zu bewerten und ihnen Feedback zu geben. Darüber hinaus könnte es den Kampfrichtern helfen, die Subjektivität bei wichtigen Wettkämpfen zu verringern. In dieser Arbeit schlage ich ein AQA-Framework für die bisher noch nicht untersuchte Disziplin Trampolin des AQA-7 Datensatzes vor, der 83 Trampolinvideos enthält. Ich stütze meine Arbeit auf das C3D-LSTM Modell, das ein vortrainiertes 3D Convolutional Neural Network (C3D) zur Feature-Extraktion benutzt und die Features mit einem LSTM aggregiert. Zunächst erkläre ich, warum diese Komponenten für ein AQA-System gut geeignet sind. Anschließend teste ich ausgeklügeltere Feature-Extraktoren sowie LSTM-Variationen. Als Schlüsselmodifikationen verwende ich zeitliche Augmentation durch Verschieben des Startbildes des Videos und räumliche Augmentation durch geometrische Verzerrungen. Mein finales Modell erreicht einen Spearman'schen Rangkorrelationskoeffizienten von 0,8485. Dies zeigt, dass mit ausreichend Augmentation und der Hilfe von vortrainierten Feature-Extraktoren selbst sehr kleine Datensätze für AQA genutzt werden können. Der Code ist unter `https://github.com/niklasluehr/tramp-aqa` verfügbar.

Contents

List of Figures

List of Tables

1 Introduction

If I asked you to judge which of ten sprinters was the fastest, you could easily figure that out. You would simply let them race head to head and whoever crossed the finish line first would win. Moreover, if the finish was tight, you could rely on an automatic timing system to determine the ranking. But what if I asked you to rank 10 trampoline athletes? First, you would need to know complex criteria after which to assess them. And even assuming you knew them, your judgement would always be subject to your individual perception.

This is an issue all performance sports like gymnastics, diving or figure skating face. Even when the judges are highly experienced, there is always a bit of subjectiveness left. After all, the athletes perform rapid movements in split seconds, so it can be hard to detect minor errors like a bent knee. On top of that, research has shown that certain biases exist among judges. For instance, sometimes athletes of the same nation are favoured [9], or more difficult performances tend to be rewarded higher execution scores [28].

This shows that it would be helpful to have an objective, automatic scoring system, just like the automatic time measurement in sprinting. The research area of *action quality assessment (AQA)* is dedicated to this issue. Its goal is to automatically assess human performance; not only in sports but also in other actions like surgical tasks. In this work, I focus on AQA in sports, particularly the discipline of trampoline.

During a trampoline routine, the athlete performs 20 tricks within 20 consecutive jumps. The scoring is based on the difficulty of the tricks, their execution, and the height of the jumps. More flips and twists, better execution and higher jumps lead to a higher score.

To my best knowledge, this is the first work regarding AQA on the discipline of trampoline. However, various works have examined other sports before. Most of them employed an end-to-end deep learning approach and based on their success I do the same. So first, I will explain basic concepts of deep learning, namely the functionality of neural networks and some special variants of them. This understanding is required to follow the rest of the paper.

1.1 Neural Network Structure & Forward Pass

I will introduce neural networks with the common example of handwritten digit classification. Imagine we have a big dataset of 28×28 pixel greyscale pictures of handwritten digits. A few examples are shown in figure 1.1 [31]. The goal for the network is now, when we input a matrix of pixel values, to output the digit these pixels represent.

Before we can understand how a neural network learns this, we first have to take a look at how a forward pass through the network works, i.e. what happens between inputting an image and receiving the output.

Figure 1.1: A few samples of handwritten digits from the MNIST dataset [31].

A neural network usually consists of several *layers* and each layer has a certain number of

neurons. For now, simply imagine a neuron as a thing that holds a real number. The value of a neuron is called its *activation*. Herein lies the analogy to the human brain that gave neural networks their name: The higher the activation of a neuron, the more likely it is to fire. Every neuron of one layer is connected to every neuron of the subsequent layer and each of these connections is assigned a *weight*, a real number that determines the influence of the neurons of one layer on those of the next layer. Such layers are also called *fully connected* layers. Figure 1.2 depicts the structure of a fully connected neural network. In state-of-the-art neural networks, a few dozen layers with several hundred neurons each are not uncommon.

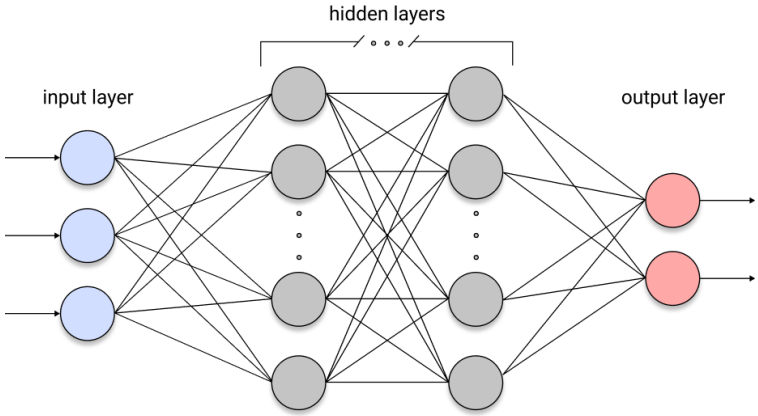

Figure 1.2: Schematic structure of a neural network with fully connected layers.

In the digit classification case, the first layer of the network needs to have 784 neurons, one for each of the 28×28 input pixels. The output layer needs to have 10 neurons, one for each digit class, whose activations are the probabilities of the input image depicting the corresponding digit. In between there are typically several *hidden layers*, that can have an arbitrary number of neurons. The activations of the first layer are simply the corresponding pixel values of the greyscale image, normalized to a range from 0 to 1. The activations of all subsequent layers are then calculated according to the weights and activations of the previous layers. Let us therefore consider the activation $a_j^{(l)}$ of the j-th neuron in layer (l). It is connected to each neuron k of the previous layer $(l-1)$ with a weight $w_{jk}^{(l)}$. At first, we compute the weighted sum

$$z_j^{(l)} = \sum_k w_{jk}^{(l)} a_k^{(l-1)}. \tag{1}$$

Since the weighted sum can technically be any number, we apply a so-called *activation function* on top of it. A common choice is the sigmoid function, which outputs values between 0 and 1:

$$\sigma(z) = \frac{1}{1+e^{-z}}. \tag{2}$$

As you can see in figure 1.3, the function maps very positive inputs to 1, very negative inputs to 0 and increases steadily around 0. Nowadays, an even simpler activation function is more popular and usually leads to faster convergence during training: The rectified linear unit (ReLU)

$$g(z) = max(0,z). \tag{3}$$

The ReLU function is plotted in figure 1.4. All negative inputs are mapped to zero and positive inputs stay the same.

Why do we need these activation functions? As you may have noticed, they are both non-linear functions. A neural network is mathematically just a function with a lot of parameters. In our example, it takes in a 784-dimensional vector and outputs a 10-dimensional vector and the

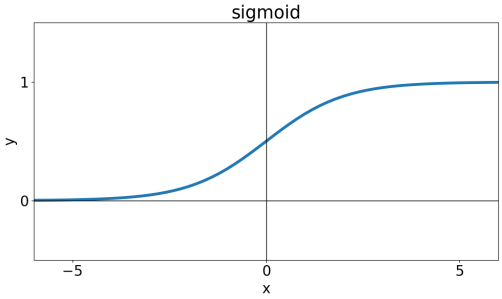

Figure 1.3: The sigmoid function.

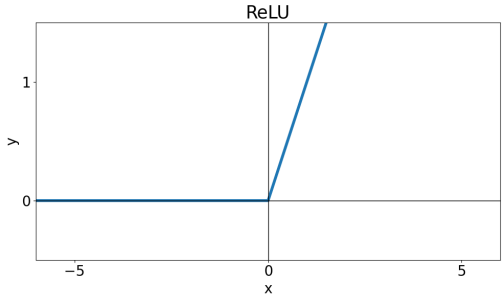

Figure 1.4: The ReLU function.

weights are the parameters. By tuning these parameters, we want to approximate a (hypothetical) underlying *true function* that maps input images to the respective digits. This underlying true function, if existent, is most likely very complex. But if we only use the weighted sums without non-linear activation functions, we cannot represent complex functions as all the weighted sums are linear operations. Therefore, we employ activation functions to introduce non-linearities and allow for more complexity.

Now maybe we only want a neuron to be active if the weighted sum exceeds a certain threshold. For this, we use a *bias* parameter, which is added to the weighted sum before the activation function is applied. Each neuron has its own bias parameter. So the final formula is

$$z_j^{(l)} = \sum_k w_{jk}^{(l)} a_k^{(l-1)} + b_j^{(l)}, \tag{4}$$

where $b_j^{(l)}$ is the bias of the neuron. So if $b_j^{(l)}$ is a very negative number for example, the activation of the neuron will only be close to 1 (in the case of sigmoid) or non-zero (in the case of ReLU) if the weighted sum is very positive.

In conclusion, the activation of a neuron is calculated as

$$a_j^{(l)} = g(z_j^{(l)}) = g(\sum_k w_{jk}^{(l)} a_k^{(l-1)} + b_j^{(l)}), \tag{5}$$

where g(·) is the activation function (see figure 1.5). This is done for every single neuron from the second layer to the output layer. Now we understand how a forward pass through the network works. But how does it actually learn? I will explain that in the next section.

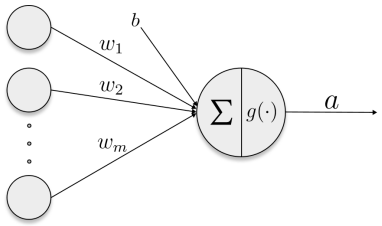

Figure 1.5: Functionality of a neuron.

1.2 Gradient Descent & Backpropagation

The handwritten digit task is an example for classification, where each sample belongs to one of a few pre-defined classes. Let us now consider a regression task as it is present in this work. This means the neural network gets a video as input and outputs a score corresponding to the performance of the athlete. So only a single number is predicted and the last layer accordingly has only one neuron.

In the beginning, all the weights and biases of the neural network are initialized with random values. So if we pass a video through it, it will output a completely random prediction. How do we now train the network to predict a score that resembles the athlete's performance? Training a network means adjusting its parameters in an iterative process with the goal that the predictions get better and better.

So at first, we need a measure to determine how good or bad the performance is, in other words, how much the predicted scores differ from the true scores. This difference is measured with a so-called *cost function* (also referred to as *loss function*). Let $\hat{y}$ denote the output of the last layer, i.e. the predicted score, and let y denote the true score. Then we can compute the cost as

$$C(w,b) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2, \tag{6}$$

where n is the number of samples in the dataset and w and b are the weights and biases of the network. This function is the *mean squared error cost function* (MSE). There are other cost functions for different tasks, but in this work I use MSE. With the cost function, we obtain a qualitative measure for the performance of the network. But not only that. We can even utilize it to improve the performance.

As the output of the cost function indicates how big the error is, our goal is to minimize the value of that function with respect to its input. The input in this case are all the parameters of the network. For simplicity, let us consider a function with a single input and a single output to demonstrate the process of finding the minimum. The first idea might be to compute it explicitly by setting the derivative to 0. Unfortunately, that method is not feasible for such complex functions as neural networks represent them. But there is a different method to at least find a local minimum. Starting at the random position corresponding to the random initialization of the parameters, we calculate the slope at that position. Then we move a bit towards the direction in which the function decreases, i.e. downwards the slope. When repeating this for many steps, we will gradually approach the closest local minimum. By setting the size of each move proportional to the steepness of the slope, we can make sure that the moves get smaller towards the valley and we do not overshoot it.

In our simplified case, the slope at each position is determined by the derivative. In the case of the multivariate neural network function, the equivalent is called the *gradient* of the cost function. It indicates the direction of the steepest increase of the function from a given point. So if we move in the opposite direction, we will move directly towards the local minimum. This optimization method is called *Gradient Descent*. Concretely, each weight and bias is updated according to the rule

$$w \leftarrow w - \alpha\frac{\partial C(w)}{\partial w}, \tag{7}$$

where the fraction is the gradient of the cost function with respect to parameter w and α is the *learning rate*. The learning rate determines how big each step towards the minimum is and needs to be set manually. If it is too low, the model takes a long time to learn, and if it is too high, we might go back and forth, each time skipping past the minimum. It is also common to gradually decrease the learning rate during the training process. The algorithm of updating the parameters is known as *backpropagation*, as it is implemented in a way that it goes backwards through the network.

So with this process, the model is supposed to learn a function that maps an input video to a reasonable score. But if we train the model on all the samples of the dataset, we might have learned a representation that only exactly fits the existing data but produces bad results on novel, unseen videos; so it would be biased in a way. However, we want the model to perform well on unseen data, too. This is called generalization ability. To evaluate this ability, we split the dataset into a training set and a test set. Then we train the model on the training set by adjusting the parameters in each iteration and evaluate it on the test set afterwards. During testing the parameters are not

updated. If the model achieves a very low error on the training set but the error on the test set remains high, this is called *overfitting*. The model has learned a suitable mapping for all the samples in the training set but cannot generalize to unseen samples from the test set. This is a common problem especially with little data at hand and I will introduce and utilize mechanisms to counteract overfitting later in this work.

I previously stated, that in each iteration, we calculate the cost by averaging over all training samples (equation 6). However, that is not practical for datasets with hundreds of thousands of samples. Instead, we use a technique called *mini-batching*. That means we randomly shuffle the training set and divide it into batches, each consisting of a fixed number of samples. These mini-batches are then fed into the network one after the other. And in each step, the cost is calculated only over the samples of the current mini-batch and the parameters are adjusted according to the cost. This is referred to as *Stochastic Gradient Descent (SGD)*. When all the mini-batches and thus all the training samples have been processed once, this is called an *epoch*. The model is then usually trained for multiple epochs, i.e. multiple cycles through the full training dataset.

1.3 Convolutional Neural Networks (CNNs)

In section 1.1 I introduced fully connected layers. For the digit classification example, the first layer needed to have 784 neurons, one for each pixel. Now imagine we have a much larger input image of 1000×1000 pixels with 3 colour channels. Then we would already need 3 million neurons just for the first layer. And if the first hidden layer had 1000 neurons, we would need 3 billion weights in between them. To address this problem of fully connected layers, we can use a different kind of layer: The convolutional layer. Not only does it reduce the number of parameters by using shared weights, but it is also able to capture spatial relationships in an image that could get lost in a fully connected layer.

Let us first consider the convolution operation on a greyscale image. The input is an image organized as a 2D matrix. The output will also be a 2D matrix. Now we need a *filter kernel*, which is simply a $k \times k$ matrix with real values, where k is usually a small odd number like 3, 5 or 7. This filter kernel is slid across the whole image pixel by pixel, such that it lies on top of every possible $k \times k$ patch of the image once. For each position, the dot product of the filter matrix and the underlying patch is calculated and the result is stored at the corresponding position of the 2D output matrix. This output matrix is called a feature map. The red and blue squares in figure 1.6 show two example positions and their corresponding output value in the feature map.

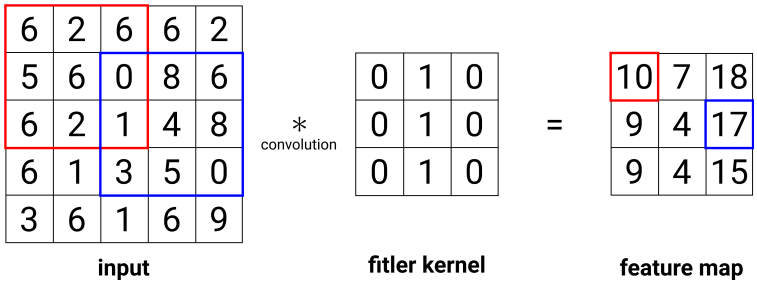

Figure 1.6: The convolution operation with two example positions. If the filter kernel is aligned with the big red square, this yields the output value inside the small red square. Same for blue.

In the case of a multi-channel image, the input is a 3D tensor, where the 2D matrices of each colour channel are placed on top of each other. The convolution principally works in the same way, only that the $k \times k$ filter kernel now has a third dimension, which is equivalent to the number of channels c. This $k \times k \times c$ filter kernel is then aligned with each $k \times k \times c$ block of the input tensor, the dot product of the two tensors is calculated and the resulting value is stored at the corresponding position of the output. So the output is still a 2D matrix. Note that even for multi-

channel images, we usually speak about $k \times k$ filters and the third dimension is added implicitly according to the number of channels.

In each layer, there are multiple filter kernels, which detect different structures, and each of them yields a different feature map. All these feature maps are then stacked on top of each other and form the input for the next layer. That means, the filters of one layer all need to have the same size. The resulting output tensor is then again treated like a multi-channel image by the next layer, where the number of feature maps corresponds to the number of channels. So if we apply 64 filters in the first layer, the filters of layer two will implicitly have a dimension of $k \times k \times 64$.

Two important terms for convolutional networks are *padding* and *stride*. Padding means adding a border of values, usually zeroes, around the input. Without padding, a 3×3 filter on a 5×5 input for example could only be applied to 3×3 positions since otherwise it would protrude beyond the border (see figure 1.6). With multiple layers and bigger filters, this would lead to a rapidly shrinking image size and neglect the border region of the image. With padding, the filter can process regions further outwards, so each pixel is given the chance to be in the centre of the filter, and the image size is kept constant. The padding has to be adjusted depending on the filter kernel size. The stride determines by how many pixels the filter is moved from one position to the next. The standard value is 1 for both dimensions. A bigger stride will lead to fewer possible positions and thus a smaller resulting feature map. This is sometimes used to downsample the image and decrease computational cost.

Now, how do convolutional layers actually capture features in an image? The idea is that filters on the first layers detect low-level features like lines or edges. The filter in figure 1.6 could for example detect vertical lines. Intermediate layers can then detect more complex features by combining the low-level features, so for example, a horizontal edge and a vertical edge combined could form a corner. This way the features get more complex with each layer and in the end, complex structures like faces can be detected.

Thanks to the mechanism of gradient descent and backpropagation, the filter kernels do not need to be handcrafted but can be learned automatically by the network. So the values of the filter matrices act like the weights in a fully connected network. Additionally, each filter kernel has a bias parameter which is added to the dot product and after that, a non-linear function is applied, in convolutional networks mostly ReLU. These parameters are then updated as usual by stochastic gradient descent.

The reasoning behind having filters that focus on local regions is that nearby pixels are likely dependent on each other while there is most likely no dependency between pixel patches far apart. This way the spatial properties of an image are retained throughout the network. Local filters also have the advantage that they can share weights, i.e. we do not need a different filter for every possible $k \times k$ patch of the image, but we rather have a fixed number of filters where each of them detects a certain structure that is present all across the image. So this decreases the number of parameters drastically. Especially for big input images, since the number of parameters is not dependent on the input size.

Another type of layer that is typically used in convolutional neural networks is the pooling layer. It is used to reduce the spatial dimension of the feature maps to reduce computational cost and to select the most dominant features. In a pooling layer, a $k \times k$ window is slid across the input and for each position, the maximum value inside that window is set as the value at the corresponding position of the output feature map. The window usually slides across the image in a non-overlapping way, so the stride in pooling layers is equal to k. The window is also referred to as kernel, but it has no parameters. For example, a pooling layer with a 2×2 kernel and a stride of 2 halves the size of the feature map. Figure 1.7 shows a max-pooling layer. Sometimes, average pooling is used instead of max-pooling, which means that instead of the maximum, the average of all values inside the kernel is passed on.

A convolutional network usually consists of a combination of convolutional and pooling layers which are employed to learn the features of an image. After that, the filter maps are most of the

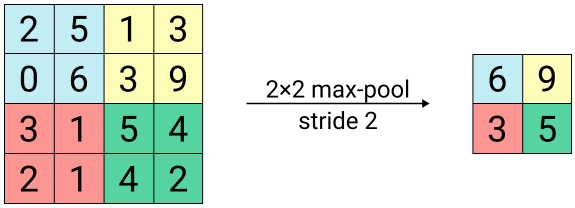

Figure 1.7: Max-pooling.

time flattened into a vector so one or more fully connected layers can perform the regression or classification task.

1.4 Recurrent Neural Networks (RNNs)

While CNNs are good for processing images, recurrent neural networks (RNNs) are good for sequential data. They are mostly used in the field of natural language processing. In CNNs we assume that the inputs are independent of each other, but for sequential data that is not true. For example, the same word might have different meanings depending on what words precede it in a sentence. An RNN does not consider each input separately but makes a decision based on the current input and the inputs it has seen before. It takes in a sequence of elements and performs the same task for each element, thus the name.

Figure 1.8 shows the structure of an RNN with a loop (left) and unfolded in time (right). In the unfolded version the network is displayed in an extended way to demonstrate how the sequence is processed element by element. In this example, the input sequence would have three elements as there are three states.

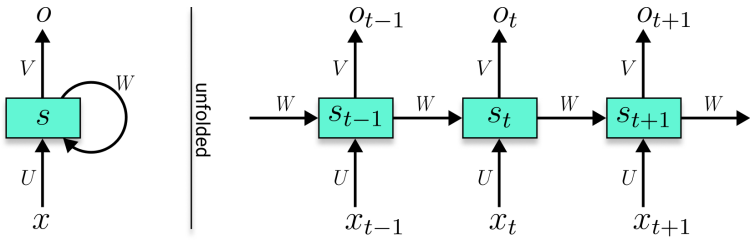

Figure 1.8: RNN structure with unfolding in time.

The state s can be thought of as the memory of the RNN. It consists of several neurons like one layer of a feed-forward network but is shown as one box here for simplification. The vector x_t is the input of one time step, i.e. one element of the sequence, and o_t is the corresponding output vector. To calculate the value of a state s_t, we compute a weighted sum that factors in the current input as well as the previous state s_{t-1}. The formula is

$$s_t = g(Ux_t + Ws_{t-1}), \tag{8}$$

where $g(\cdot)$ is the non-linear activation function and U and W are weight matrices. Note how this is similar to equation 1, only that matrix notation is used here. The difference is, that in addition to the weighted sum Ux_t, the previous state is multiplied with a weight matrix W and added before the activation is applied. This is where the memory effect happens. The third weight matrix V is used to calculate the output from a given state s_t. There is only one set of weight matrices in an RNN, and their weights are refined with each element that is processed. This leads to a significantly lower number of parameters. Since RNNs take in one element at a time, the input sequences can have variable length and the number of parameters does not depend on the sequence length.

The RNN parameters are tuned with a mechanism called *backpropagation through time*. It is similar to the backpropagation algorithm used in standard neural networks, but since an output in the RNN depends not only on the current input but also on the previous ones, the backpropagation has to happen backwards in time. However, standard RNNs can have problems capturing long-term dependencies, i.e. dependencies between distant time steps. This is due to the phenomenon of *vanishing or exploding gradients*, which essentially means that the gradients calculated during backpropagation get so small that the network learns really slow or not at all, or that they get so large that the network is unstable and learns ineffectively. Vanishing or exploding gradients are more likely to happen when the gradient is accumulated over multiple time steps because the gradients are multiplied during backpropagation. So the product of many very small gradients gets exponentially smaller and the product of many very high gradients gets exponentially larger. This issue is addressed by Long Short-Term Memory cells, which I will introduce in the next section.

1.5 Long Short-Term Memory (LSTM)

LSTM networks, often referred to as *LSTMs*, are a special type of RNNs that were proposed by Hochreiter and Schmidhuber [12] to tackle the problem of vanishing or exploding gradients. The general structure is similar to the one shown in figure 1.8, but the way a state is calculated and passed on to the next time step is more sophisticated. The mechanism that happens inside the box is shown in figure 1.9 [2].

The key elements of an LSTM cell are its cell state vector C_t and its three gates. The cell state conveys the information from one time step to the next, and the gates decide what information to keep or remove and what new information to add from the current input. So in theory, information from the first time step could be kept all the way to the last time step, which is the reason why the LSTM can capture long-term dependencies better than a standard RNN.

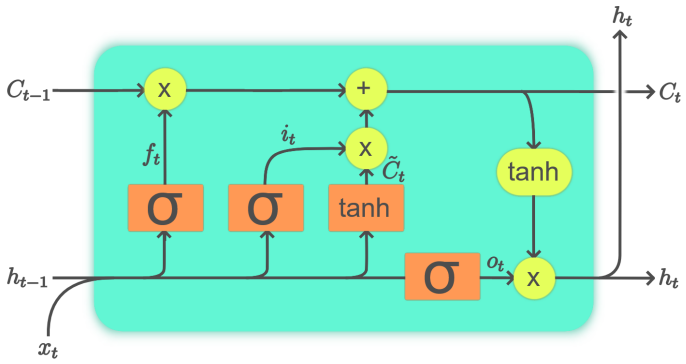

Figure 1.9: An LSTM cell, adapted from [2].

Let me explain how exactly this works. All the lines "transport" vectors. The orange boxes represent standard neural network layers with the respective activation function, where σ is the sigmoid function (figure 1.3) and tanh is the hyperbolic tangent function; another activation function that outputs values between -1 and +1 (plotted in figure 1.10). All the orange boxes take in the last hidden state vector h_{t-1} and the current input vector x_t. The yellow circles represent element-wise operations, where (x) denotes the dot product and (+) denotes element-wise addition and the yellow tanh oval means that the tanh function is applied to each element.

I will go through the cell (figure 1.9) from left to right. The first gate is the *forget gate* f_t. The orange box learns weights to determine which features of the cell state to keep and which features to forget based on the previous hidden state and the current input. Since the sigmoid function outputs values between 0 and 1 and these values are then multiplied with the values of the cell state, a 0 would lead to forgetting the corresponding feature and a 1 would lead to keeping it.

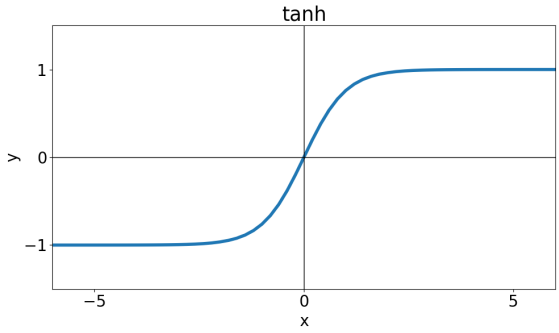

Figure 1.10: The hyperbolic tangent function.

The second gate is the *input gate* i_t. At first, the candidate vector $\tilde{C}$ is computed by a neural network layer with tanh activation. The tanh activation is used to compress the numbers into a range between -1 and 1 and the resulting candidate vector $\tilde{C}$ holds possible candidates that could be added to the cell state. The actual input gate has a neural network layer with sigmoid activation that learns which of the candidates should be added to the cell state. With the dot product, the candidates that are determined to be relevant are then selected and the result is added onto the cell state.

The last gate is the *output gate*. At first, the current cell state C_t is passed through a tanh function (yellow oval box). So here there is no neural network layer and hence no weights. The tanh function is simply used to compress the values of the cell state into a range from -1 to 1. After that, the dot product is applied, in which the features of the cell state that are believed to be relevant are selected and outputted.

$$f_t = \sigma(W_f h_{t-1} + U_f x_t + b_f) \tag{9}$$

$$i_t = \sigma(W_i h_{t-1} + U_i x_t + b_i) \tag{10}$$

$$\tilde{C}_t = \tanh(W_c h_{t-1} + U_c x_t + b_c) \tag{11}$$

$$C_t = C_{t-1} \otimes f_t + i_t \otimes \tilde{C}_t \tag{12}$$

$$o_t = \sigma(W_o h_{t-1} + U_o x_t + b_o) \tag{13}$$

$$h_t = o_t \otimes tanh(C_t) \tag{14}$$

The above equations show the formulas for all gates, the cell state and the output. The dot product is denoted by $\otimes$. W_* denotes the respective weight matrix holding the weights from one time step to the next and U_* the respective weight matrix holding the weights from the input to the cell; b_* is the bias vector of each neural network layer.

The output dimensions of the four neural network layers (orange boxes) are all the same and equal to the dimension of the cell state vector C_t and the hidden state vector h_t. This dimension is often referred to as the number of *hidden nodes* or *hidden units* of an LSTM.

1.6 A Deep Learning Approach to Action Quality Assessment (AQA)

Now that I have established the functionality of neural networks, especially CNNs and LSTMs, let me explain the general idea behind my approach. The goal of this work is to develop an end-to-end deep learning AQA system on the trampoline videos of the AQA-7 dataset [24]. The main challenge is to cope with the small dataset of only 83 videos.

In section 2 I summarize various works on and related to AQA and most of them operate on the same fundamental principle: At first, the video is split into clips of a fixed number of frames because due to computational cost the video cannot be processed as a whole. From these clips,

features are extracted with a 3D convolutional network, a certain variant of CNNs that is better suited for video analysis. After that, the features are aggregated and the final score is regressed. I employ the same principle in my approach and use the C3D-LSTM model proposed by [23] as my baseline model. In section 3 I explain how exactly C3D-LSTM, and thus the basic principle, works.

Based on the C3D-LSTM model, I propose several modifications to improve the performance on the trampoline videos. The modifications are inspired by the approaches introduced in section 2 and include the use of different feature extractors and aggregation methods. I also test human pose estimation for feature extraction. Furthermore, I address another problem inherent in the field of AQA: The lack of annotated data. While for action classification, videos can be annotated easily by non-experts or even automatically from meta-information, the annotation in AQA requires scoring from expert judges. Moreover, for classification, it is sufficient to use short snippets from a video, so the number of samples can easily be increased, whereas for assessment always the whole video needs to be considered. To counteract the lack of data, I employ augmentation techniques. I describe all the modifications in section 4.

After that, I evaluate my additions and adjustments to the baseline model in detailed experiments to see if they yield an improvement. I describe these experiments and discuss the results in section 5.

I summarize my contributions as follows:

- I build the first AQA framework dedicated to the discipline of trampoline.
- Despite the very small dataset (<100 samples), I manage to reach competitive performance by employing extensive augmentation and utilizing pre-trained feature extractors.
- I experimentally compare different pre-built components and different settings of common deep learning building blocks to obtain the best possible framework for my use case.

2 Related Work

In the last decade, several works regarding action quality assessment on sports videos were published. They studied various disciplines like diving [26, 23, 24, 19, 25, 22, 33, 14, 6], figure skating [26, 23, 37, 19, 39], gymnastics vault [23, 24, 19, 22, 33, 14], ski and snowboard big air [24, 22, 33] and rhythmic gymnastics [39]. Although [24] introduced a new dataset containing trampoline videos, no one so far has conducted AQA experiments on that data. Hence, the only research on trampoline data available [10, 3] deals with the classification of trampoline skills.

In this section, I give an overview of existing work on trampoline data and action quality assessment on sports videos. I also introduce selected works on action recognition, which set a foundation for the AQA methods, and put my work into context.

2.1 Trampoline Skill Classification

The work of Helten et al. [10] differs from all the remaining ones in so far that instead of video data they used data from wearable motion sensors. They attached seven sensor units providing acceleration, turn rate, and orientation data all over the athlete's body. Firstly, they automatically segmented the motion sequence into separate jumps utilizing the absolute acceleration of the body. Secondly, they calculated compound features from a combination of specific sensor values to form *motion templates* that represent a certain trampoline skill. An unknown jump was then labelled with the class of the motion template having the smallest distance to the jump. With this technique, the authors were able to reliably classify jumps despite significant style variations of different athletes.

Connolly et al. [3] classified jumps based on human pose information. They first masked out the athlete from the image data. From this, the authors segmented the sequence according to the athlete's position and then extracted 2D poses with a CNN-based pose extractor and smoothed them temporarily. Features were formed from joint angles and limb orientations. Similar to [10] they manually annotated reference skills and classified unknown jumps with a nearest neighbour classifier. A limitation of this method is the dependency on accurate pose estimation, which can be tricky when the athlete assumes unusual body positions or people in the background are mistakenly detected (cf. section 4.6).

2.2 Action Recognition

A cornerstone for many works in the field of action recognition and AQA was the three-dimensional convolutional network introduced by Tran et al. [34], commonly referred to as *C3D*. They studied different 3D CNN architectures that learn spatio-temporal features to build a compact and efficient generic video descriptor. I give more details on C3D in section 3.1.

This idea was taken to the next level by Carreira and Zisserman [1], who took advantage of the success of ImageNet architectures that had been thoroughly developed over many years. They proposed a method to convert these architectures into 3D space by *inflating* them to make them suitable for video analysis, resulting in the *I3D* network. In section 4.1 I provide greater insight on I3D.

In a following work, Tran et al. [35] explored different architectures of deep residual networks (ResNets) [8] for video analysis. They found that 3D ResNets significantly outperform 2D ResNets on video classification tasks. They also proposed a new (2+1)D ResNet architecture *R(2+1)D*, where the 3D convolution operation is separated into a 2D spatial convolution and a subsequent 1D temporal convolution. I further explain this in section 4.2.

2.3 Sports AQA

Pirsiavash et al. [26] were one of the first to propose a generic AQA system on sports data, namely diving and figure skating videos. They calculated features based on human pose estimation and Discrete Cosine Transform and used Support Vector Regression (SVR), a specific regression method, to predict scores. By employing pose information as features their system could give interpretable feedback on which joints to adjust in which way to improve the score. Like in [3], their approach was limited by the challenge of accurate pose estimations. On top of that, important aspects like the amount of splash in diving could not be factored in.

Parmar and Morris [23] made use of C3D as a feature extractor. They argued that directly processing visual information instead of noisy pose information yields better results. They compared SVR, Long Short-Time Memory (LSTM) and the combination of both for feature aggregation, resulting in three different frameworks: C3D-SVR, C3D-LSTM and C3D-LSTM-SVR. While C3D-SVR yielded the best results on the diving dataset, the second-placed C3D-LSTM-SVR provided the opportunity to detect erroneous segments according to the temporal score evolution throughout the LSTM. The authors also found that *incremental label training*, i.e. backpropagating the LSTM gradients after each processed clip of a video, could produce better results with fewer iterations.

Xu et al. [37] tried to capture both local, technical movements and the global overall performance of figure skaters by feeding the C3D-extracted features into a self-attentive LSTM and a multi-scale skip-LSTM, respectively, which are modified LSTM versions. The self-attentive LSTM was used to select important clip features and the skip-LSTM to save computational cost. They also regressed for both scoring criteria separately.

Li et al. [19] used nine different C3D networks which did not share weights to account for the different phases of diving. They argued that the assessment of diving performance is essentially a task of ranking. To include the ranking constraint of the predicted scores they integrated what they call *ranking loss* into the traditional Mean Squared Error (MSE) loss function.

Parmar and Morris [24] exploited the fact that certain disciplines share common action quality elements and examined whether knowledge transfer was possible. Therefore, they released the AQA-7 dataset consisting of videos from seven Olympic events. Using C3D-LSTM from [23], they found that training a multi-action model (trained on multiple actions) can improve the predictions compared to a single-action model. On top of that, they showed that pre-training a model on multiple actions can provide a good initialization for training on another unseen action class.

In their next work [25], Parmar and Morris proposed a multitask learning approach, arguing that a single score label is not sufficient to describe a complex action. They introduced the MTL-AQA dataset containing three kinds of annotations: the score, the performed dive and a caption generated from expert commentary. Their model accordingly learned three tasks: predicting the AQA score, recognizing the specific dive and captioning its execution. With this additional information, the model achieved better generalization ability, especially on the small datasets prevalent in the AQA domain.

Pan et al. [22] stated that vision-based methods only incorporate visual information while ignoring fine-grained joint information vital for sports assessment. Therefore, they suggested a graph-based model focusing on joint relations, that captures both general body part kinetics and joint coordination via the *Joint Commonality Module* and the *Joint Difference Module*, respectively. I3D was used to extract whole-scene features as well as joint features from estimated human poses. The temporal and spatial relations between the joints were modelled with graphs. Eventually, this framework was able to assign a performance rating to specific joints. The authors explained the improvement over former pose-based approaches with the fact that they considered the interaction of joints rather than only single joints.

Zeng et al. [39] studied AQA of long videos by employing a hybrid architecture that factors in not only dynamic but also static features, i.e. the athlete's posture at certain moments. The authors state these to be an important indicator for action quality that can easily be neglected in

dynamic approaches. Therefore, they extracted I3D features from the video and ResNet features from detected humans in selected frames. These features were each fed into a context-aware *attention module* to explore their relation and assign them a weight, since they were claimed to not be equally important, especially in long videos.

Tang et al. [33] took the probabilistic nature of scores caused by different judges and their subjectiveness into account. Instead of regressing a final score, they suggested learning the parameters of a probabilistic score distribution. Clip-level features were extracted with I3D and aggregated by averaging. The authors also used difficulty labels beside the judges' scores to further improve their model.

Jain et al. [14] criticized that simply regressing the final score does not provide the opportunity to interpret the awarded score. Hence, they presented a Siamese network for deep metric learning that first learns to compute the similarity of any two videos and then calculates AQA scores by comparing a given video to an expert reference video. This comparison occurred clip-wise, such that the contribution of each clip to the final score could be measured and therefore the model provided better interpretability.

Finally, Farabi et al. [6] studied the effect of using 3D and (2+1)D ResNets with different depths and clip sizes. Based on the assumption that not all parts of a video are equally important, they also proposed a novel learning-based aggregation technique, where the weights of each feature are learned with a dedicated shallow neural network, the *Weight-Decider*. They found that a 34-layer-deep (2+1)D ResNet that takes 32-frame input clips yielded the best results. Moreover, the Weight-Decider module outperformed the trivial aggregation by averaging.

In this work, I will address the task of AQA on the yet unstudied discipline trampoline using the AQA-7 dataset, of which the trampoline videos have so far always been excluded. I will utilize various concepts proposed in the aforementioned works and examine their effect on the model performance. I will particularly test different augmentation techniques, some of which did not get much attention in recent works, to cope with the challenge of the small dataset at hand.

3 Baseline Approach: C3D-LSTM

In this section, I explain the functionality of my baseline approach. The C3D-LSTM architecture introduced in [23] by Parmar and Morris shall serve as my baseline model since it was also used in their follow-up work [24], along which the AQA-7 dataset was released. Even though the authors excluded the trampoline videos from their experiments, the model performed well on the other disciplines, so I build my model upon this general architecture, which basically works as follows: The video is split into clips of 16 frames. Each of these clips is fed into the C3D feature extractor to obtain a 4096-dimensional feature vector. All the feature vectors are then passed into the LSTM module as a sequence to aggregate them. The LSTM module consists of the LSTM layer itself with 256 hidden nodes and a fully connected layer which is used to regress the final score. Note that C3D is pre-trained on a large-scale video classification dataset, so its parameters are kept frozen during training and only the parameters of the LSTM and fully connected layer are learned. In the next two subsections, I explain in detail how C3D works and why C3D and LSTM were chosen.

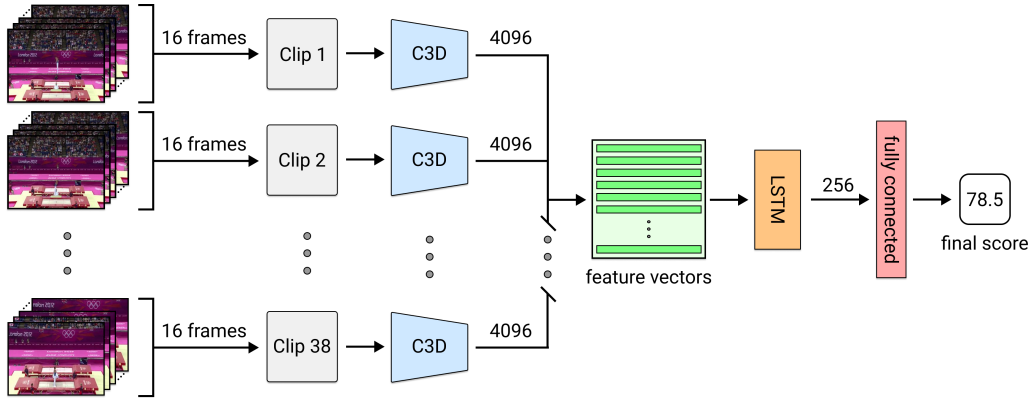

Figure 3.1: The C3D-LSTM model.

3.1 C3D Feature Extractor

The C3D network was introduced by Tran et al. [34] for the task of action recognition. While previous works had tried to classify videos by using 2D CNNs on single frames, the authors proposed 3D CNNs because they preserve temporal information. But what exactly does that mean? In the following subsection, I explain the advantage of 3D convolution over 2D convolution in terms of video analysis.

3.1.1 Advantage of 3D Convolution

So you might ask why we do not simply consider the video as a sequence of images and process each image separately with a 2D CNN. In the end, 2D CNNs have been studied for many years and improved to the point where they can achieve exceptional performance. This is a valid point and in section 4.1 I will pick up this point again. But 2D CNNs have a significant disadvantage over their 3D counterpart, which I will explain in the following.

What is the problem with 2D convolutions? Let us first recall the basic mechanics of a 2D convolution operation. We can imagine it as $k \times k$ filter kernel being slid across the input image. At each position, we calculate the dot product, which determines the value at the corresponding position of the output, i.e. the next layer. In this way, we can preserve spatial information. If we have a multi-channel input image with c channels, a $c \times k \times k$ kernel will be applied, but for

each position it still only calculates one dot product, resulting in a 2D output again. So assuming that we treated a video as a multi-channel image, the 2D convolution would collapse the 3D input volume into a flat 2D image, thus discarding all temporal information. In other words, this means we could randomly re-arrange the video frames, i.e. input a completely different (meaningless) video, and would still always receive the same output. So the model cannot learn what happens from one frame to the next, which is essential to understand the video. This issue is addressed by 3D convolution.

How does 3D convolution work? For 3D Convolution, the input is a 3D volume, in our case a video, i.e. a sequence of images. Let us first consider a video with one colour channel. Like in the 2D case, a filter kernel is used, only that in this case, it has the dimensions $d \times k \times k$, where d denotes the temporal depth and k is the spatial size. This cuboid kernel is now slid across the volume, this time moving in three dimensions. Again, we compute the dot product for each kernel position (a dot product of two 3D tensors), but since the kernel moves in three directions, the output now has also three dimensions. So while sliding across the temporal axis, for each d frames the $d \times k \times k$ cuboid has processed, it will output a different value considering the relationship between the $k \times k$ patches of these d frames. This enables us to keep track of temporal information, as shuffling the video frames would yield a different output. Note that in the case of c channels, we have c cuboids, each responsible for one channel. They are always moved synchronously and at each position the dot product is computed over all the cuboids and the pixels they overlap, leaving us with a 3D output. With this mechanism, we can now better represent videos by learning spatio-temporal features instead of only spatial ones.

3.1.2 C3D

In their C3D paper, Tran et al. [34] experimented with different kernel sizes in different layers and found that kernels of size $3 \times 3 \times 3$ across all layers perform best. They also found that half the input resolution of 112×112 pixels produced similar results to the full resolution. The network takes in clips of size $3 \times 16 \times 112 \times 112$, where 3 denotes the number of channels and 16 is the number of frames per clip. All the convolutions in C3D use $3 \times 3 \times 3$ kernels, a stride of $1 \times 1 \times 1$ and a padding of 1 in every dimension to keep the image size constant. All pooling layers except the first one use max-pooling with kernel size $2 \times 2 \times 2$ and a stride of 1. The first max-pooling layer has kernel size $1 \times 2 \times 2$. So here no pooling is performed along the temporal axis so that the temporal information does not get merged too early. After the last pooling layer, the 4-dimensional feature tensor is flattened out to an 8092-dimensional vector, of which the dimension is in turn reduced to 4096 with a fully connected layer. I follow the work of [23] and [24] and use the C3D network up to that fully connected layer, to obtain a 4096-dimensional feature vector. Figure 3.2 shows the full architecture of the C3D feature extractor.

In accordance with the theory, Tran et al. [34] experimentally showed that C3D outperforms 2D CNNs on different video analysis tasks. It detects spatial features in the first few frames and then focuses on salient motion. It is also very easy to use and runs fast on today's GPUs. Compared to a frame-wise 2D CNN approach, it provides a way more compact representation as it always processes 16 frames at once. Moreover, it is superior to pose-based features, since it factors in visual information like the amount of splash in diving. For these reasons, it was repeatedly used as a building block in subsequent works on AQA [23, 37, 19, 24, 25, 14] with great success.

Since all of these worked on datasets that are very small for an end-to-end deep learning approach (171 to 1412 samples), they employed pre-trained versions of C3D. These were pre-trained on one of two large-scale action recognition datasets, either UCF101 [30] with 13k videos from 101 classes or Sports-1M [16] with 1M videos from 487 classes. The idea behind this is that the pre-trained network is already able to accurately describe a video after it has processed this multitude of videos. The weights are then frozen after pre-training, i.e. they are not optimized further

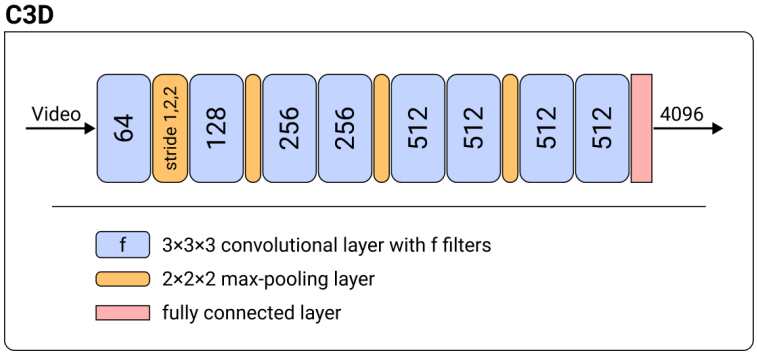

Figure 3.2: The C3D feature extractor. All convolutional layers have a stride of 1 and padding of 1 in every dimension. All pooling layers except the first have a stride of 2 in every dimension.

through SGD during the AQA training phase, and only the remaining parts of the respective network are trained. With this technique, even datasets of only a few hundred samples are sufficient to achieve good performance. I further explain this in section 5.3.

3.2 LSTM Feature Aggregation

The C3D feature extractor outputs a feature vector for every 16-frame clip. How do we now aggregate all these features to receive a video-level description? An easy way would be to simply average them as it was done with the C3D-SVR [23] or C3D-AVG [25] models. But as some authors [37, 39, 14, 6] state, not every segment of an action contributes equally to the final score. This was also experimentally shown by Li et al. [19] when they zeroed all segments except one at a time to measure their contribution. Therefore, different approaches to consider the importance of each segment were proposed in the past, e.g. reference videos [14], attention mechanisms [37, 39], or a dedicated *Weight-Decider* network [6]. With reference videos, each segment is compared to an expert segment and the similarity is calculated. Attention mechanisms and the Weight-Decider network learn how relevant each segment is to the final score and accordingly assign weights.

Another approach is to use an LSTM for feature aggregation. This was already done in the work on video classification by Ng et al. [21], who proposed to treat extracted features as an ordered sequence and use LSTM cells to generate a video-level description. They extracted 2D CNN features from single frames and used five stacked LSTMs with 512 hidden nodes each for aggregation. Ye and Tian [38] proceeded similarly: They extracted clip features with C3D and used one LSTM with 256 hidden nodes to aggregate them. Inspired by these works, Parmar and Morris [23] designed their C3D-LSTM architecture, using an LSTM with 256 hidden nodes. In an earlier work on video classification, Donahue et al. [5] had also tested feature aggregation with 256, 512 and 1024 hidden units, but they had all performed similarly. Parmar and Morris [23] came to the same conclusion and found that adding a second LSTM layer did not improve performance either. Hence, I will use a single LSTM layer with 256 hidden units for my baseline model. However, we will later see why a second layer could indeed be useful for longer videos (section 4.4).

The usage of LSTM has several advantages. First, due to the nature of LSTMs and RNNs in general, they can take in sequences of variable length while maintaining a fixed number of parameters. This could be used to assess videos of different lengths with the same model. Moreover, the LSTM allows us to model the sequential characteristics of a video, as opposed to averaging techniques which are insensitive to the order of extracted features. Finally, it gives us the possibility to provide feedback by identifying segments that have led to a lower score.

4 Designing a Trampoline AQA Framework

In this section, I outline the design process of my trampoline AQA framework. Starting from the baseline approach (C3D-LSTM), I step by step introduce the modifications I tested to boost the model's performance. These concern the quality of extracted features and the best way to aggregate them and dealing with the data scarcity.

4.1 I3D Feature Extractor

While C3D has proven to be an effective feature extractor for spatio-temporal features, there is a prominent counterpart among past AQA works: The I3D network by Carreira and Zisserman [1]. They proposed a novel architecture for spatio-temporal feature extraction, which relies on the success of ImageNet [29] architectures. ImageNet is a huge image classification and object detection dataset on which many experiments are conducted. The authors point out that image classification networks have been developed over many years through a tedious process, such that today they achieve excellent performance. So why should we repeat this whole process for videos if we can instead leverage the 2D architectures and apply them to video analysis? This is what they did with their I3D network. They drew on the successful Inception-v1 network [13] and *inflated* it to the third dimension, thus the name. The success of Inception-v1 relies on two novel concepts: Its architectural design and a concept called *batch normalization* [13]. I will first explain both of these in the following sections to provide the fundamental understanding before I will finally cover the I3D feature extractor.

4.1.1 The Inception Architecture

Inception nets are deep convolutional networks for image classification and object recognition tasks. The architecture was introduced by Szegedy et al. [32] to enable deeper and wider networks while keeping the computational cost constant. While increasing network depth and width is an obvious approach to improve performance, it comes with two drawbacks: Firstly, bigger networks usually have more parameters, therefore posing the risk of overfitting. Secondly, they will need much more computational power. The Inception architecture addresses these issues with two novel concepts.

The first deals with the question of which kernel sizes to use for which layers. The answer is the so-called *Inception module* (see figure 4.1). It takes the input from the previous layer and simultaneously performs multiple convolution operations with different kernel sizes as well as a max-pooling operation. The outputs from all these operations are then concatenated. In this way, the model itself can learn which of them are helpful. Concretely, one Inception module uses 1×1, 3×3 and 5×5 convolutions and 3×3 max-pooling. Now with this module configuration there arises another problem: The 3×3 and even more the 5×5 convolutions are very high in computational cost, especially when performed on a layer with many filters.

Here the second idea comes into play. Inexpensive 1×1 convolutions are applied before the 3×3 and 5×5 convolutions and after the max-pooling operation with the single reason to reduce the dimensionality, so the following operations can work with fewer filters. Besides that, they also increase the number of non-linearities. With these dimensionality reductions, Inception-v1 has considerably fewer parameters than prior state-of-the-art ImageNet models. The full network is then composed of several Inception modules, with common layers (convolutional, pooling, linear) before, in between and after them. This network-in-network structure gave the architecture its name. With GoogLeNet, a deep network built after the Inception Architecture, the authors could produce state-of-the-art results on the ImageNet challenge by a significant margin.

4.1.2 Batch Normalization

The second concept responsible for the success of Inception-v1 is batch normalization. It was proposed by Ioffe and Szegedy [13] to shorten deep network training time by reducing what the authors refer to as *internal covariate shift*. This is essentially the phenomenon that the input distribution for each layer changes with each training step because the parameters of the previous layers change. It has been common practice to normalize training data as a preprocessing step to enable faster convergence of the model. The authors now suggest applying such a normalization step to individual layers, thus making it a part of the architecture, to normalize the inputs of the respective layers. Concretely, this step takes place before the activation function is applied.

So let us for simplicity consider a standard neural network with fully-connected layers. Let $z^{(1)}, ..., z^{(m)}$ denote the intermediate values (weighted sums, see equation 1) of some layer l where batch normalization should be applied. We first calculate mean and variance of these $z^{(i)}$:

$$\mu = \frac{1}{m}\sum_i z^{(i)} \tag{15}$$

$$\sigma^2 = \frac{1}{m}\sum_i (z^{(i)} - \mu)^2. \tag{16}$$

After that, each $z^{(i)}$ is normalized as follows:

$$z^{(i)}_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}. \tag{17}$$

Note that ε is a constant added for numerical stability. Now all the normalized $z^{(i)}_{norm}$ have mean 0 and variance 1. However, this is not desired for all neurons at any time as sometimes it might be beneficial if they have a different distribution. Therefore, for each neuron i of layer l, two new model parameters $\gamma^{(i)}$ and $\beta^{(i)}$ are introduced. These are used to scale and shift the normalized values:

$$\tilde{z}^{(i)} = \gamma^{(i)} z^{(i)}_{norm} + \beta^{(i)} \tag{18}$$

The parameters γ and β are updated just like any other model parameter. So basically, this normalization process ensures that the neurons have some fixed mean and variance which are learned by the model. Finally, the activation function is applied on the $\tilde{z}^{(i)}$ to yield the output of layer l. In the case of convolutional layers, the mean and variance are calculated for each single feature map. Every feature in that feature map is then normalized with the corresponding values. Moreover, there is one pair of γ and β parameters for each feature map. So this is how the normalization part works.

The actual training process is then realized with mini-batches. The normalization takes place only on the data of the current mini-batch and each time after one mini-batch was passed through the network the parameters are updated with backpropagation. Batch normalization allows us to be less thorough with the search of hyperparameters like the learning rate since it makes the network more robust. Even very deep networks can be trained easily and the same results can be achieved faster than without batch normalization because it allows us to use a higher learning rate [13].

4.1.3 I3D

Carreira and Zisserman [1] tested different architectures for action recognition and I3D performed best among them. As stated above, they simply took the Inception-v1 network and inflated its filters. That means they transformed the $k \times k$ 2D components into $d \times k \times k$ 3D components by adding a third dimension. Moreover, some minor changes to certain kernel sizes and stride values were applied. All the $k \times k$ receptive fields from Inception-v1 were left at the same size. Now

they needed to find suitable values for the newly added temporal dimension d, for which they proposed the following configuration: The first two max-pooling layers do not perform temporal pooling, so they have a kernel size of $1 \times 3 \times 3$ and stride of $1 \times 2 \times 2$. Note that this is similar to the C3D architecture (cf. section 3.1.2) and ensures not to merge the temporal information too quickly. The two other pooling layers use symmetric kernels and strides, i.e. the same value for each dimension. The final average pooling layer has a kernel size of $2 \times 7 \times 7$. Furthermore, each convolutional layer is followed by a batch normalization layer and a ReLU activation function. The full architecture (without batch normalization and ReLU) is shown in figure 4.1.

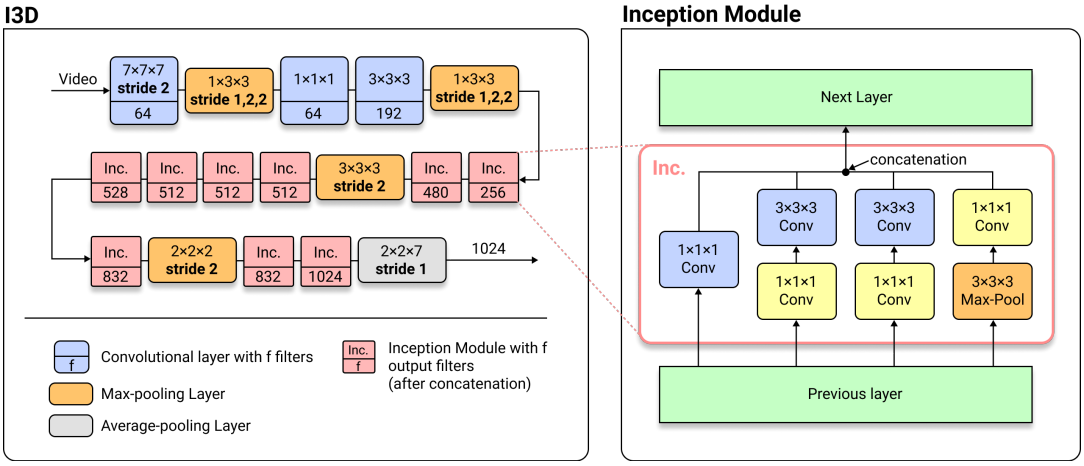

Figure 4.1: Architecture of the I3D feature extractor with the Inception Module. Each convolutional layer is followed by batch normalization and ReLU.

There is another important advantage of inflating the ImageNet model beyond exploiting its successful architecture. Namely, we can even adopt its parameters after it was pre-trained on ImageNet. This is done by simply stacking the parameters of the 2D filters N times along the temporal dimension and dividing by N to scale them. With this method we trivially receive a 3D model with weights pre-trained on the largest computer vision dataset. As the authors of [1] show, these weights do indeed offer a valuable initialization. On top of that, I3D is pre-trained on the Kinetics dataset [17], a human action recognition dataset with over 400 videos from 400 classes, where each sample is from a unique YouTube video. Presumably due to pre-training on the huge amounts of data from ImageNet and Kinetics and its superior architecture, I3D was shown to outperform C3D, even when batch normalization was added to C3D. On top of that it takes in input frames of 224×224 pixels, so the increased resolution may be another advantage.

I3D has been used in more recent works on AQA to extract features from the video [22, 39, 33]. Pan et al. [22] also used it to extract features from local patches around the joints of the athlete. While in the original paper, Carreira and Zisserman used 64-frame long clips to train the model, the three AQA papers worked with 16-frame input clips, which yields a 1024-dimensional feature vector. In my experiments, I also used 16-frame clips as input. The results are presented in section 5.4.

4.2 R(2+1)D Feature Extractor

Deep Residual Networks (ResNets), introduced by He et al. [8], are among the best performing architectures for image classification. So it seems obvious to utilize them for video analysis tasks like it was done with Inception net and I3D (cf. section 4.1). So let me shortly explain the characteristic mechanism behind residual networks: residual connections. As the authors of [8] state, network depth is of crucial importance to learn features on many different levels and state-of-the-art ImageNet models are all very deep. But with deeper networks they observed a *degradation*

problem: With deeper and deeper models, the performance first saturates and then starts to degrade rapidly. Unexpectedly, this is not caused by overfitting, because the training performance was found to decrease as well. As a solution to this problem, He et al. proposed the concept of *residual learning*. This basically means adding a shortcut connection between two layers to skip the layers in between. The shortcut connection simply performs an identity operation and is then added to the output of the stacked layers (see figure 4.2 [8]). The authors suggest that it is easier to optimize this residual mapping.

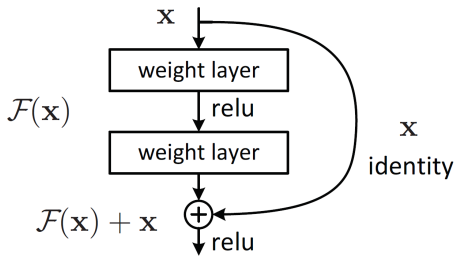

Figure 4.2: A residual connection [8].

ResNets were taken to the third dimension by Tran et al. [35]. They proposed a new (2+1)D architecture, where the 3D convolution operations were split into a 2D spatial convolution and a subsequent 1D temporal convolution. So, for example, when we previously had a $3 \times 3 \times 3$ filter, we now perform $1 \times 3 \times 3$ convolutions first and then convolve temporally with $3 \times 1 \times 1$ filters. The number of $1 \times 3 \times 3$ filters is chosen so that the number of parameters in a (2+1)D block is equal to those of a 3D block. This architecture has the advantage that additional non-linearities can be added between the two operations so that more complex functions can be learned. Furthermore, R(2+1)D was shown to perform on par with 3D ResNets that have three times as many parameters. Finally, a lower training cost indicated that it was easier to optimize.

Farabi et al. [6] used R(2+1)D to extract features from diving videos. In their experiments, they also found that using longer clips might be more beneficial than using deeper networks. Specifically, they found that 32-frame clips outperform 16- or 8-frame clips. Their (2+1)D ResNet had 34 layers and was pre-trained on IG-65M [7] and fine-tuned on Kinetics [17]. The IG-65M action recognition dataset features 65 million web videos that are automatically annotated with the help of hashtags and is intended for pre-training. Despite the automatic annotation it was shown to provide a good initialization for action recognition models. So in my work, I tested the same R(2+1)D network as used in [6] as feature extractor. Different to I3D and C3D it takes in 32-frame long clips and outputs 512-dimensional feature vectors. The results can be found in section 5.4.

4.3 Temporal Augmentation

An inherent problem of currently available AQA datasets is that they have very few samples. The biggest dataset to date is the MTL-AQA dataset [25] containing 1412 samples. Most of the other datasets have sizes in the range of hundreds. The AQA-7 dataset [24] contains 1106 samples from seven different disciplines. But since I only want to work on the trampoline data, I am left with just 83 videos. Compared to state-of-the-art image classification datasets like ImageNet [29] with 14 million samples, this number appears tiny for a deep learning approach. Nonetheless, I will show that with the right techniques it is possible to achieve competitive results even on such a small dataset.

So the challenge is to learn a model that can generalize to unseen data and does not overfit the training data, which is usually harder with less data. One technique to address this is data augmentation. This means simply copying existing samples and slightly altering them to create new artificial samples. That way we do not have to collect new data and annotate it. In the com-

puter vision domain, this could be done by transformations like flipping, rotating or cropping or by changing the brightness or the contrast. Another possibility unique to video data is that we can augment along the temporal dimension. I will introduce two methods for temporal augmentation in the following: Shifting the start frame and having overlapping clips.

4.3.1 Start Frame Shift

For the first mechanism, we have to recall how C3D "sees" the video, namely in clips of 16 frames. Each clip is then processed according to learnt 3D convolutional filters to output a feature vector. Now if we imagine we had a video of 17 frames, we could pass it twice through the network, the first time taking frames 1-16 and the second time taking frames 2-17. C3D would each time output a different feature vector, even though it has seen almost the same video apart from one frame difference at the beginning and at the end. In my application, this difference becomes irrelevant since first of all the videos are over 600 frames long and secondly, the first and last few frames contain actions irrelevant for scoring, like initial jumps to gain height and standing still at the end.

So essentially, we simply copy a video and shift the start frame. The AQA-7 trampoline videos are all 618 frames long and C3D requires clips of 16 frames. So when we split the first 608 frames into 38 clips, there are 10 unused frames at the end. This means we could technically shift the start frame 10 times to create 10 copies. While the resulting copies still represent the same trampoline routine and thus have the same score, they are all new inputs from the model's perspective. By this, the model should achieve better generalization performance, on one hand because it gets more data to learn from and on the other hand because it learns that slight temporal variations of the same athlete performance still lead to the same score.

Due to its simplicity, this technique was also applied in some former works. [19] and [6] shifted the start frame by a random number. [23], [24] and [39] shifted as many times as there were unused frames at the end. Surprisingly though, [23] found that the augmentation worsened the result. In my experiments, I compared the performance when inputting 2, 4, 6 and 8 temporally shifted versions of each video. The results are displayed in section 5.5.1.

4.3.2 Overlap Between Clips

Another way to generate more information from the data is to have overlapping clips. In my baseline model, the clips are sliced with a stride of 16, so there is no overlap between them. By reducing the stride so that the clips overlap we can calculate more feature vectors. Even though there is no actual new data, the C3D network now computes features from different snippets of the video, that lead to different outputs. This technique was already applied in the original C3D paper [34], where the authors applied a stride of 8, i.e. clips were overlapping by 8 frames. Other authors [5, 23, 37, 33] adopted this technique and also used a stride of 8. Parmar and Morris [23] also experimented with different strides and found a stride of 4 to be optimal for their C3D-SVM model. However, on the C3D-LSTM model, a stride of 8 performed worse than a stride of 16.

Remember that once the features are extracted by C3D they are put in a sequence and fed into the LSTM. Now if we use a stride of 8 instead of 16, we will have almost twice as many feature vectors, i.e. a sequence of double the length as LSTM input. In section 4.4, I will explain why a 2-layer-LSTM or a bidirectional LSTM may be better suited to cope with the longer sequence. Accordingly, I tested a stride of 8 and 16 and combined both with a single-layer LSTM and a two-layer LSTM. I report the results in section 5.6.

4.4 LSTM Variations

As stated in section 3.2, prior works [5, 23] found no performance improvement with more than one LSTM layers or more hidden units. But they only worked on videos with less than 150 frames.

The key point why I wanted to test different LSTM variations is the increased length of the feature vector sequence in my framework, which has two reasons: Firstly, the trampoline videos are 6 times as long as all the other videos from the AQA-7 dataset which C3D-LSTM was trained on. Therefore, the sequence is also 6 times as long. Secondly, as explained in section 4.3.2, setting the stride to 8 results in a longer sequence as well.

Zeng et al. [39], who also worked with long videos (more than 2000 frames), suggested using a bidirectional LSTM (Bi-LSTM) with 128 hidden units for this case. A bidirectional LSTM consists of two separate LSTM layers, where one processes the input sequence in its normal order and the other one in reverse order. The outputs of both directions are then combined to form the final output.

Another technique I tried was *dropout*, a commonly used method originally proposed by Hinton et al. [11], to prevent overfitting. Dropout means dropping out a subset of randomly chosen neurons during training. That means they are not considered in a particular forward and backward pass. The connections from and to them are also removed. With each mini-batch that is processed, each neuron of a dropout layer is either kept with a keep-probability p or otherwise dropped out with probability $p-1$. The intuition behind that is, that dropout prevents so-called co-adaptations, which means that some neurons are only helpful in combination with specific other neurons. Dropping out some of these other neurons forces the remaining ones to learn representations that are helpful on their own. In other words, if for a neuron some ingoing connections are removed with each iteration, it eventually has to learn to not only rely on one input but to consider all ingoing connections.

In my experiments, I tested different two-layer LSTM and Bi-LSTM setups. I also studied if increasing the hidden dimension and using dropout could improve performance. The results are shown in section 5.6. The general idea behind this is that with multiple LSTM layers, the network may be able to learn more complex features. On the downside though, there are also more parameters to be learned, which requires more training data. To counteract this problem, I employed data augmentation techniques as presented in sections 4.3 and 4.5.

4.5 Spatial Augmentation

As described in section 4.3, all the AQA datasets are very small compared to image classification or action recognition datasets and can benefit from data augmentation. While temporal augmentation is unique to video data, spatial augmentation is an established method in computer vision that is crucial to improving image classification models. This is because deep networks tend to generalize better when trained on massive amounts of data. On top of that augmentation introduces more variation into the samples, so the model needs to learn a more general representation. In their paper on object detection, Zoph et al. [40] could show that data augmentation was particularly beneficial to smaller datasets because it provided a strong regularization and thus prevented the model from overfitting.

In the image domain, transformations like horizontally flipping the image, rotating it, translating it or cropping it are commonly used. Other techniques include adjusting brightness, contrast and colour. These transformations can be analogously applied to videos. Surprisingly, only a few of the past AQA papers [23, 25, 22, 14, 6] reported using spatial data augmentation at all. And all of them except one only used random horizontal flipping of the input images. This is especially surprising considering the small datasets at hand. Jain et al. [14] used a combination of zooming, varying the brightness, masking of the background and histogram equalization, but no horizontal flipping. They increased the number of pairs for their Siamese network 20-fold which had a positive impact on the performance.

In my experiments, I studied if spatial augmentation yields the expected improvement. Following past AQA works, I employed horizontal flipping. Moreover, I used rotation and horizontal shearing, since these were the transformations that were found to be most beneficial in [4] and

[40]. I report the results in section 5.7.

4.6 Pose Estimation

Pose-based methods were already used in some early works on classification [3] and action quality assessment [26]. However, later papers pointed out that only relying on human poses had two limitations: Firstly, visual information like the splash in diving cannot be factored in and secondly, athletes assume unusual positions during their routines, so common pose detectors fail to recognize these poses properly. So the following works focused only on visual information by employing convolutional networks.

However, Pan et al. [22] and Zeng et al. [39] additionally considered pose information. While [22] introduced a completely new architecture revolving around joint relations, [39] pursued a quite simple approach: As they examined long videos of about 2000 to 4000 frames, they argued that when only considering the dynamic information, incorrect static postures of the athletes would get ignored due to their short occurrence. Therefore, they extracted one frame per second from the video and detected human bounding boxes in it. If one or more humans were detected they took the one with the largest bounding box, cropped the frame and extracted features with ResNet [8] (cf. section 4.2) pre-trained on ImageNet, which were then embedded together with the dynamic features. Despite only one frame per second was extracted, the static information increased the performance when added to the dynamic stream. Even more, on some disciplines, the static stream alone performed better when compared to the dynamic stream.

For these reasons, I wanted to try this method on the trampoline videos. Zeng et al. had used YOLOv3 [27] to detect humans in the frames, an object detection network pre-trained on the MS-COCO object detection dataset [20]. So I used the improved version YOLOv5 [15] and tested the object detection on a few trampoline videos. Unfortunately, the results were not satisfying, as seen in figure 4.3. Each row shows a few successive frames from a trampoline video. In the top row, the camera was placed below and towards one corner of the trampoline. In each frame, a flying object was indeed detected, but it was labelled incorrectly. Now we could simply ignore any objects except persons, but there occurs another problem as we can see in the bottom row, where the camera is filming straight from the side. While the athlete is not detected in every frame, the spotters (who try to keep the athlete safe in case he falls off the trampoline) at the bottom of the trampoline are. But of course, their posture does not contribute to the scoring in any way, so simply extracting features from any detected person would not make sense.

I also tried to detect the athlete with Detectron2 [36], a Facebook AI Research object detection library that also features models for image segmentation or pose detection. Like YOLOv5, it draws the bounding boxes of detected persons and displays the probability. On top of that, it tries to estimate the human pose and draws the joints on the image. Figure 4.4 shows the results analogously to figure 4.3. In the top row, the athlete is detected in every frame with a high probability, although the pose estimations are not accurate. But again, the performance on the side view (bottom row) is bad. Persons in the audience and especially the spotters are detected, the latter with a high probability, whereas the athlete is not detected at all or with low probability.

With both detection systems, I got similarly unsatisfying results on a few other videos, so I decided to discard this approach since it would rather confuse the model instead of adding useful information.

Figure 4.3: YOLOv5 test. Top row/bottom view: Athlete is detected but labelled incorrectly. Bottom row/side view: Athlete is not detected in all frames but the spotters are.

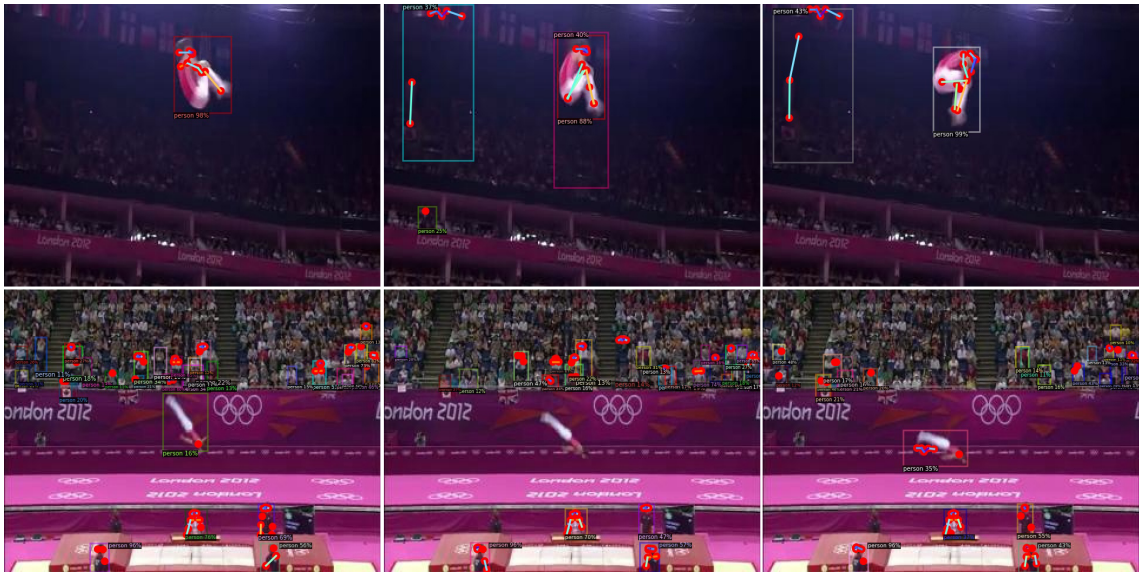

Figure 4.4: Detectron2 test. Top row/bottom view: Athlete is detected with high probability but pose estimations are inaccurate. Bottom row/side view: Athlete is detected with low probability or not at all while persons in the audience and especially spotters are detected in all frames with high probability.

5 Experiments

In this section, I introduce the AQA-7 dataset and explain the technical details of my implementation. After that, I present and discuss the results of my experiments, in which I tested the modifications described in section 4. Finally, I compare my final model to the state-of-the-art results on the AQA-7 dataset.

5.1 AQA-7 Dataset

The AQA-7 dataset was released by Parmar and Morris [24] and contains videos from 7 different Olympic events: Single Diving 10m, Synchronized Diving 10m and 3m, Ski Big Air, Snowboard Big Air and Trampoline. But in their original work and in two later works that used the AQA-7 dataset [22, 33], the trampoline videos were excluded because they are much longer than all the other videos and the routines are composed of multiple tricks/jumps instead of a single one. The dataset contains 1106 videos in total, thereof 83 trampoline videos.

I focused only on the trampoline videos. Since a trampoline routine must consist of 10 jumps, they are all similarly long. They are normalized to a fixed length of 618 frames and zero-padded at the beginning where necessary. With 30 frames per second, this yields a length of 20.6 seconds. The videos have a resolution of 320×240 pixels. They start with the initial jumps (straight jumps where the athlete gains height) and end after the routine when the athlete stands still on the trampoline; except if the gymnast fails and has to abort the routine, then the video ends directly after the fail.

There are two different camera angles: One static angle directly from the side a few meters above the ground and one dynamic angle slightly to the left from ground level. In the dynamic angle, the camera follows the competitor up and down, so they always stay in the middle of the frame. The videos start and end with the side view, but in between the perspective switches to the bottom view for a few seconds. This happens randomly as the videos are taken from TV broadcasting. Examples for both camera angles can be seen in figures 4.3 and 4.4.

All videos are taken from the London 2012 Olympic Games except three which are from Rio 2016. They feature male and female athletes, but the scoring rules are the same for both genders. The scores range from 6.72 to 62.99.

5.2 Common Implementation Details

In the following, I will describe the common implementation details that were used for the baseline model and remained unchanged in all the experiments I did. I used the machine learning framework PyTorch[1] for the implementation. I split the 83 trampoline videos into a training set with 65 samples and a test set with the remaining 18 samples. The videos were resized and centre-cropped corresponding to the required input size of the respective feature extractor (cf. section 5.4). The training data and the test data were then normalized with mean and standard deviation of the training data. Since the videos are 618 frames long, there are 10 unused frames when working with non-overlapping 16-frame clips (C3D, I3D) or also 32-frame clips (R(2+1)D). Therefore, I ignored the first 10 frames and started from frame 11, because the beginning only contains the initial jumps which are irrelevant for the score.

Features were extracted from each clip, concatenated and passed through the respective LSTM component. After that, I used a fully connected layer, which maps the output of the last LSTM time step to a single value; the final score. As cost function, I employed the mean squared error (MSE). Only the LSTM and fully connected layer parameters were learned, i.e. the components of the LSTM module. Adam optimizer [18], an extension to stochastic gradient descent, was used to adjust them. The initial learning rate was 0.0001 as in [24]. If the training loss did not decrease

[1] `https://pytorch.org/`

for 5 epochs, it was divided by 10. I used a training batch size of 5 and a test batch size of 6. Every experiment was evaluated on (the same) five random splits.

Evaluation Metric As evaluation metric I used Spearman's rank correlation coefficient ρ, which is standard practice among past AQA works. It represents the correlation between the ranks of two series, in this case between the true scores and the predicted scores. It ranges from -1 to +1. The higher ρ, the better. A correlation of +1 would mean that the true scores have the same order as all the corresponding predicted scores. However, they do not need to have the exact same values, nor do they need to have a linear relationship, i.e a twice as high true score does not need to have a twice as high corresponding predicted score, as long as the order is the same. Spearman's rank correlation coefficient (SRC) is calculated as

$$\rho = 1 - \frac{6\sum d_i^2}{n(n^2-1)}, \tag{19}$$

where d_i is the difference between the ranks of corresponding scores and n is the number of samples.

5.3 Baseline Results

For the baseline experiments, I rescaled the 320×240 videos to 171×128 pixels and centre cropped them to 112×112 pixels as done with C3D-LSTM in [24]. I set clip size and stride to 16 and used C3D weights[2] pre-trained on the Sports-1M dataset [16]. I trained the model for 60 epochs on five random splits and averaged the results. This yielded a rank correlation of 0.5436, which indicates that the model did indeed learn certain characteristics of action quality in the trampoline videos.

But how is this possible with only 83 samples? The answer lies within the pre-training of the C3D feature extractor. C3D is pre-trained on the action recognition dataset Sports-1M. As the name suggests, Sports-1M only contains sports videos in which humans play many different sports. To be exact, there are 478 classes and a total of more than 1 million videos. So during pre-training, the model tries to learn the characteristics of human behaviour in various sports scenarios to discriminate between different sports. Furthermore, the classes are very fine-grained. For example, there are 6 different types of bowling, 7 different types of American football and 23 types of billiards. So to distinguish between them, the model also needs to learn fine-grained features of human movement. This information is encoded in the over 61 million parameters of C3D after pre-training. That means the weights I use are already adjusted in such a way that the feature extractor can output a meaningful description of the human activity in a given video, without having seen a single trampoline video before. Of course, this description does not necessarily make a point about the quality of the action, since that is not relevant to determine the action itself and thus was not learnt during pre-training. So this is the challenge for the regression task carried out by the LSTM module, and seemingly, with the inherent information from over 1 million sports videos in C3D, 83 trampoline videos are enough for these components to learn how to discriminate between different grades of action quality.

In my baseline experiment, I did not yet use horizontal flipping as opposed to [24], because I studied spatial augmentations in a separate experiment (cf. section 5.7). However, to compare the trampoline results with the other AQA-7 disciplines, I trained my baseline model again, this time with horizontal flipping to match the approach of [24]. This means I flipped every video with a probability of 0.5. The results are shown in table 5.1. As you can see, the same C3D-LSTM applied to trampoline videos already achieved competitive results, even though they are much longer and composed of multiple tricks. The average correlation of the other six disciplines is still higher, but the trampoline results were already better than the ones for gym vault, skiing

[2] https://github.com/DavideA/c3d-pytorch

and snowboard without any improvements to the model. This may be explained by the high view variation in the videos of the latter three disciplines. It is important to say though, that the single-action results (training and evaluating the model on one discipline each) were not the main task of Parmar and Morris [24], since they studied all-action models (training and evaluating the same model on all disciplines). With this all-action approach they could improve all results except snowboarding and skiing beyond my trampoline rank correlation of 0.5773 (see table 5.1, bottom row). This means there was much room to improve my model to suit the characteristics of the trampoline videos. However, the comparison shows that the general architecture was a promising baseline to build upon.

		Diving	Gym Vault	Skiing	Snowboard	Sync. 3m	Sync. 10m	Trampoline	Avg. Corr.
C3D-LSTM no flip	mine	-	-	-	-	-	-	0.5463	0.5463
C3D-LSTM w/ flip	mine	-	-	-	-	-	-	**0.5773**	0.5773
single-action	[24]	0.6047	0.5636	0.4593	**0.5029**	0.7912	0.6927	-	0.6165
C3D-LSTM w/ flip all-action	[24]	**0.6177**	**0.6746**	**0.4955**	0.3648	**0.8410**	**0.7343**	-	**0.6478**

Table 5.1: Baseline C3D-LSTM results with and without horizontal flipping in comparison with the other AQA-7 disciplines (single-action and all-action).

5.4 Feature Extractor Comparison

In my next experiment, I wanted to find out if the more recent feature extractors I3D [1] and R(2+1)D [6] could provide the expected improvement over C3D. I simply exchanged C3D with them and only made changes that were required by the respective feature extractor.

I3D is designed to process input frames of size 224×224, so I did not resize the videos and only centre cropped them to the required size. This way they were cropped at the same ratio as for C3D. I also needed to change the input dimension of the LSTM to 1024, as that is the dimension of the feature vector outputted by I3D. The clip size remained the same. I used a model with weights[3] pre-trained on ImageNet [29] and fine-tuned on Kinetics [17].

R(2+1)D requires the same input size as C3D but outputs a vector with only 512 dimensions. Furthermore, it processes 32 frames at once. I used a model with weights[4] pre-trained on IG-65M [7] and fine-tuned on Kinetics.

I trained both networks for 60 epochs like in the baseline experiment. Table 5.2 shows an overview of the three feature extractors and the results. As expected, both I3D and R(2+1)D outperformed C3D. I3D had the best overall performance, so I used it in all subsequent experiments.

	Input Size	Center Crop	Clip Size	Output Vector	Pre-Training	SRC
C3D	171×128	112×112	16	4096	Sports-1M	0.5463
R(2+1)D	171×128	112×112	32	512	IG-65M+Kinetics	0.6099
I3D	320×240	224×224	16	1024	ImageNet+Kinetics	**0.6297**

Table 5.2: Feature extractors: Comparison of properties and results.

Obviously, I3D and R(2+1)D are newer models that have been developed to exceed the existing ones. But what exactly makes them better than C3D? Apparently, they produce better descriptions of the actions in the clips so that the LSTM module can in turn learn better representations of action quality.

[3] https://github.com/piergiaj/pytorch-i3d

[4] https://github.com/moabitcoin/ig65m-pytorch

The C3D architecture was developed specifically for action recognition in videos with little research existing prior. On the contrary, I3D and R(2+1)D make use of state-of-the-art image classification models, where thorough research has already been conducted. First, they are much deeper, which is believed to be of crucial importance [8] and is evident in many recent image classification models. The increased number of layers allows learning representations on more levels. Thanks to special architectures like inception modules or residual connections and new concepts like batch normalization, even such deep models can achieve very good results. So while C3D relies on basic 3D convolutions, the other two models can exploit those novel concepts.

Moreover, I3D takes in four times as many pixels as the other two feature extractors, which may in the end make it produce the most meaningful descriptions. The larger input size enables it to detect more detailed features in the first layers and to use more information in general: Since the videos are not resized for I3D and only cropped, almost all the available pixel information containing the athlete (apart from a few pixels at the top and bottom) is exploited.

Another aspect is pre-training. Since I3D is based on an image classification model, we can even transfer the parameters from pre-training on ImageNet to the third dimension, which has been shown to be helpful [1]. After that, I3D is fine-tuned on Kinetics. R(2+1)D is pre-trained on IG-65M, which features 65 million videos, and fine-tuned on Kinetics as well. C3D is also pre-trained, but only on one dataset, Sports-1M. However, Sports-1M might provide a good pre-training basis specifically for the trampoline videos as it only consists of sports videos. On the other hand, Kinetics may be of higher quality because the videos are annotated manually as opposed to Sports-1M. So it would be interesting to exchange Kinetics with Sports-1M and compare the performance. Unfortunately, there are no pre-trained models available.

Finally, the dimension of the feature vectors might play a role. I3D produces a quarter as many and R(2+1)D an eighth as many features. That means they can compress the action description into a more compact representation. This might in turn help the LSTM module to gain a better understanding of action quality. A reduced LSTM input dimension also leads to fewer LSTM parameters, which might be another advantage especially with the small amount of data. However, R(2+1)D performs worse than I3D, even though it outputs fewer features. So maybe it does not work that well together with LSTM. Farabi et al. [6] also used it with a different feature aggregation mechanism. Another explanation might be that the clip size of 32 leads to an only half as long feature vector sequence, which might not be sufficient for the LSTM.

5.5 Temporal Augmentation

In my next experiment, I examined the effect of temporal augmentation by shifting the start frame and by overlapping clips. For all experiments, I used the best-performing feature extractor I3D from the previous experiment.

5.5.1 Start Frame Shift

As explained in section 4.3.1, when using a clip size of 16, we have 10 unused frames in the 618-frame long videos. Since I had discarded the first 10 frames in my baseline setup (section 5.2), I used them now for temporal augmentation. Concretely, I trained the model with $n = 2, 4, 6, 8$ temporally shifted versions of each video. So I inputted each video n times, each time starting one frame earlier; the first time starting at frame 11 and the n-th time starting at frame $11 - n + 1$. I will from now on refer to n as the number of *(temporally) shifted versions* of a video. I trained the model for 30 epochs, where one epoch means that every video and every shifted copy of it has been seen once. The smaller number of epochs was enabled by the increased number of videos seen in each epoch compared to the baseline experiment. Figure 5.1 shows the results together with the result from the I3D experiment without temporal augmentation ($n = 1$).

n	SRC
1	0.6297
2	0.6999
4	0.6904
6	0.7181
8	**0.7635**

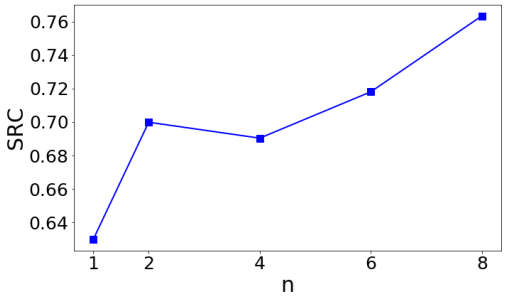

Figure 5.1: Temporal augmentation with n different start frames ($n = 1$ means no augmentation).

Unsurprisingly, the performance increased with an increasing number of shifted versions. Notably, this little trick improved the performance significantly despite being very easy to implement. For all following experiments, I set the number of temporally shifted versions to 8.

As explained earlier, this augmentation technique basically gives us more training data for free, because from the perspective of the feature extractor, a video that is shifted by one or more frames is a different input and thus yields a different description of the action. But of course, the shifted video resembles the same athlete performance apart from a few frames at the beginning and at the end that are barely relevant for the action quality. So in the end, the LSTM module can learn from more data. Moreover, the actions in the test videos are likely not aligned, so for example the first trick of the routine will happen at different times. Therefore, the LSTM module most likely performs better after it has seen a greater variety of training videos where specific actions happen at different times.

5.5.2 Overlap Between Clips

While I had used a stride of 16 in the previous experiments, which means no overlap between clips, I now tested a stride of 8, which means an overlap of 8 frames between two consecutive clips. I used 8 shifted versions and trained the model for 30 epochs. The result is shown in table 5.3 (LSTM column).

Unfortunately, the smaller stride also led to worse results with the trampoline videos, as already observed in [23] with diving videos. So apparently, the additional information packed into the feature vector sequence did not help the LSTM module to learn a better representation for action quality.

5.6 LSTM Comparison

Next, I tested different LSTMs for feature aggregation. At first, I wanted to see if the stride of 8 in experiment 5.5.2 produced such bad results because the single LSTM layer could not handle the long input sequence well. Therefore, I repeated the experiment, but this time with a two-layer LSTM. I set the number of hidden nodes to 128 for each layer, such that the total number of hidden nodes remained the same. Thus, the fully connected layer had an input dimension of 128. I also repeated the experiment with stride 16 and the two-layer LSTM. As you can see in table 5.3, the two-layer LSTM did not make a significant difference with stride 8. Therefore, I set the stride to 16 in all following experiments.

With stride 16, the two-layer LSTM even decreased the performance, so I tested further LSTM variations. All the results are shown in table 5.4. First, I took the two-layer LSTM and doubled the number of hidden units in each layer. To counteract the increased risk of overfitting, I employed dropout between both layers with a keep-probability of 0.5 (C). The results were similar to the ones from the two-layer LSTM, so I discarded this approach.

	LSTM	2-LSTM
Stride 16	**0.7635**	0.7238
Stride 8	0.6755	0.6780

Table 5.3: Comparison of stride 8 and 16 combined with a one-layer LSTM (LSTM) and a two-layer LSTM (2-LSTM).

Next, I experimented with a bidirectional LSTM. At first, I tested 128 hidden units per direction without dropout (D), which performed similarly to the standard LSTM. After that, I added dropout to the fully connected layer after the LSTM with a keep-probability of 0.5 (E). This did not increase performance either, so I doubled the number of hidden units in each direction (F). Note that the outputs of both directions in a Bi-LSTM are combined to form the final output, so the output passed on to the fully connected layer was twice as large as in the two-layer LSTM case. All Bi-LSTM setups performed approximately on par with the standard LSTM. But since they did not yield a significant improvement, I decided to keep the most simple setup and use the standard LSTM for further experiments.

(A)	LSTM	0.7635
(B)	2-LSTM (128)	0.7238
(C)	2-LSTM-do (256)	0.7271
(D)	Bi-LSTM (128)	**0.7697**
(E)	Bi-LSTM (128) + fc-do	0.7655
(F)	Bi-LSTM (256) + fc-do	0.7610

Table 5.4: Comparison of different LSTM setups (stride 16).

5.7 Spatial Augmentation

Finally, I tried different geometric transformations for spatial augmentation. At first, I introduced random horizontal flipping. This means each input video was randomly flipped horizontally with a probability of 0.5 before it was fed into the network. I used the standard LSTM with 256 hidden nodes for feature aggregation and trained the model for 30 epochs. This modification alone already yielded a decent increase in performance as you can see in table 5.5.

Next, I employed two more transformations, namely rotation and shearing. Rotation rotates each input video by a random angle, which I set to the range [-20, 20]. Shearing shifts the corners of the image along the x-axis by a random amount, so the image is transformed into a parallelogram shape. Here I set the range to [-20, 20] as well. Both transformations keep the centre of the image invariant and do not scale it, so there are black patches in the resulting images and some (unimportant) information in the corners is lost. Figure 5.2 shows an example for each transformation.

In each epoch, I made two copies of the training data to have three times as many videos in total. Then I applied each of the two transformations with such a probability, that on average one third of the videos would not be transformed, and the other two thirds would either be rotated or sheared or both. Concretely, I set the probability for each transformation to 0.42. That way, the probability for no transformation to be applied was 0.33, the probability for exactly one transformation was 0.49, and the probability for both transformations was 0.18. In the latter case, shearing was always applied before rotation. On top of that, the videos were again flipped horizontally with a probability of 0.5. I trained the model for 30 epochs. Note that this time three times as many videos were processed per epoch. The result is shown in table 5.5.

Figure 5.2: Spatial augmentation: Rotation and shearing.

Spatial Augmentation	SRC
none	0.7635
flip	0.7928
flip, shearing, rotation	**0.8485**

Table 5.5: Effect of spatial augmentation.

Adding rotation and shearing significantly improved the performance yet again. But how do geometric transformations like flipping, shearing and rotation contribute to learning a better representation of action quality? First, similar to the effect of temporal augmentation, we can drastically increase the number of training samples with little effort. This alone leads to better results, as deep learning models are reliant on large amounts of data to tune all the parameters.

On top of that, augmentation not only increases the number of samples, but also the variety among them. Let us consider this by the example of horizontal flipping. After processing the training data, let us assume that the model has learned to map a certain trampoline trick and its execution to an action quality score. Now if the test data contains a routine where the same trick occurs, but the athlete faces the opposite direction during the trick, the model might not be able to utilize the learned information about that trick, simply because it is horizontally flipped. But if the model had already seen a flipped version of the training video and thus the specific trick, it could potentially exploit its knowledge about it to assess the quality of the trick in the test video.

The same principle applies to rotation and shearing. For example, by rotating and/or shearing a trick that occurs filmed from the side view in the training videos, it could be transformed in such a way, that it looks more similar to the same trick filmed from the bottom view in the test videos.

Of course, all the transformations in my experiment were applied randomly, so probably not every one of them produced a helpful variation as described in the two latter examples. But since in every epoch, I applied a combination of up to three possible transformations on the tripled training dataset, the model sees different variations in every epoch. While not all of them may be helpful, this evidently still benefits the model in total.

5.8 Overview & Discussion

Table 5.8 shows an overview of all my experiments. With my final setup, I3D-LSTM with 8 shifted versions, a stride of 16 and horizontal flipping, shearing and rotation I achieved a Spearman's rank correlation coefficient of 0.8485. In table 5.7 I compare this final result to the state-of-the-art results on the AQA-7 dataset by Tang et al. [33]. Since they excluded the trampoline videos from their experiments, I can only compare it to the other six disciplines in the dataset and their average correlation. For a fair comparison, one would need to test their Uncertainty-aware Score Distribution Learning (USDL) Approach on the trampoline videos. However, the comparison still shows that my model achieved competitive performance on the trampoline videos and on average performed better than USDL on the other disciplines.

Feature Extractor	Shifted Versions	Stride	Aggregation	Spatial Augmentation	SRC
C3D	1	16	LSTM (256)	-	0.5463
				flip	0.5773
R(2+1)D	1	16	LSTM (256)	-	0.6099
I3D	1	16	LSTM (256)	-	0.6297
	2				0.6999
	4				0.6904
	6				0.7181
	8				0.7635
	8	8	LSTM (256)	-	0.6755
			2-LSTM (128)		0.6780
		16	2-LSTM (128)	-	0.7238
			2-LSTM-do (256)		0.7271
			Bi-LSTM (128)		0.7697
			Bi-LSTM (128) + fc-do		0.7655
			Bi-LSTM (256) + fc-do		0.7610
			LSTM (256)	flip	0.7928
				flip, shearing, rotation	**0.8485**

Table 5.6: Experiment overview.

The order of the USDL results is the same as the one of the C3D-LSTM single-action results reported in [24] (cf. table 5.1). So apparently, some disciplines are consistently harder to assess. The results for gym vault, skiing and snowboard are most likely worse due to the high view variation in the videos, even though all three disciplines have at least 175 samples. The 10m single diving ("Diving") results are behind the synchronous diving disciplines, which is surprising since the former has 370 samples while the latter two only have 88 and 97 samples and the view variation is very similar among all three. So the results for the synchronous diving disciplines are surprisingly good considering the small number of samples, which shows that there is still room for improvement in my model. On the other hand, compared to the diving videos, trampoline has a greater view variation as I described in section 5.1, which presumably makes it harder to assess.

	Diving	Gym Vault	Skiing	Snowboard	Sync. 3m	Sync. 10m	Trampoline	Avg. Corr.
Mine	-	-	-	-	-	-	0.8485	**0.8485**
USDL [33]	0.8099	0.7570	0.6538	0.7109	0.9166	0.8878	-	0.8102

Table 5.7: Comparison of my final model with state-of-the-art results [33] on the AQA-7 dataset.

6 Conclusion & Future Work

In this work, I have examined various methods to build a trampoline action quality assessment framework. I introduced the C3D-LSTM model and its components and explained that 3D convolution is better suited for video analysis than 2D convolution because it can capture spatio-temporal instead of only spatial features.

In my experiments, I found that the C3D-LSTM model provides a good basis. Inspired by some of the presented literature I tested certain modifications to the baseline model. These included two other feature extractors and the use of pose features, spatial and temporal augmentation and several LSTM variations.

The main challenge was the small dataset with only 83 videos. However, I have shown that with the right techniques, even a model trained on such a small dataset can achieve competitive performance. The key to this was spatial and temporal augmentation, which boosted the performance significantly. While most of previous work did not fully exhaust the possibilities of augmentation, I increased the number of training data in every epoch 24-fold. I achieved this by using 8 temporally shifted versions of each video and copying the training data twice while randomly applying horizontal flipping, rotation and shearing with a given probability in every epoch.

Moreover, I exchanged C3D with the I3D and R(2+1)D feature extractors. Both provided a significant improvement over C3D. This was due to their superior architecture with new concepts like the Inception architecture, batch normalization and residual connections as well as better pre-training and smaller feature vectors. I3D was the only feature extractor to process full resolution inputs and produced the best results.

Although I tested various variations of the LSTM Module for feature aggregation, the baseline one-layer LSTM was not outperformed significantly. Neither two-layer nor bidirectional LSTMs could produce better predictions. Dropout between the LSTM layers or in the fully connected layer was not helpful either, nor was the combination of dropout and an increased number of hidden nodes.

I also tried to incorporate human pose information but unfortunately the results of the human detection systems YOLOv5 and Detectron2 were not sufficient due to distractions in the background and overall bad performance on the trampoline videos. Furthermore, decreasing the stride to have overlapping clips did not help, either.

Overall, I have built a model which is competitive with state-of-the-art results on the AQA-7 dataset. While it cannot outperform human expert judges, it still shows that research in that direction is promising. On top of that, the result was achieved on a very small dataset, which demonstrates the effectiveness of pre-training and augmentation.

Future Work In future work, it might be interesting to revisit the pose feature approach, since the poses of the athlete are partially what determine the execution score. Therefore, they could provide additional information when combined with the visual features. One idea might be to only use the uppermost detected human per frame and only if the bounding box size exceeds a certain threshold.

It might also be interesting to examine other aggregation mechanisms than LSTMs. Such a mechanism should work in a way, that it assigns different weights to the segments as they are not all equally important. One idea would be to use an attention model, as it was done in [37, 39]. Another possibility is a dedicated network like the Weight-Decider in [6]. In general, the idea behind that is that the relevance of each segment to the final score is learned during the training process.

One could also argue that a single score label is not sufficient to describe a complex action sequence like the 20 jumps in a trampoline routine. Therefore, it might be helpful to provide separate labels for the difficulty and execution score. [25] went even further and incorporated automatically generated captions based on the TV commentary, which contained helpful information about the

execution. Of course, such an approach requires additional annotation work, but the additional information could lead to better learning abilities. To save annotation effort, one could also leave the task of determining the difficulty to an action recognition model and incorporate it with the AQA model, as it was done in [25].

References

[1] Joao Carreira and Andrew Zisserman. Quo vadis, action recognition? a new model and the kinetics dataset. May 2017.

[2] Guillaume Chevalier. Larnn: Linear attention recurrent neural network. https://en.wikipedia.org/wiki/Long_short-term_memory#/media/File:The_LSTM_Cell.svg, 2018. [Online; accessed May 3, 2021].

[3] Paul W. Connolly, Guenole C. Silvestre, and Chris J. Bleakley. Automated identification of trampoline skills using computer vision extracted pose estimation. September 2017.

[4] Ekin D. Cubuk, Barret Zoph, Jonathon Shlens, and Quoc V. Le. Randaugment: Practical automated data augmentation with a reduced search space. September 2019.

[5] Jeff Donahue, Lisa Anne Hendricks, Marcus Rohrbach, Subhashini Venugopalan, Sergio Guadarrama, Kate Saenko, and Trevor Darrell. Long-term recurrent convolutional networks for visual recognition and description. November 2014.

[6] Shafkat Farabi, Hasibul Haque Himel, Fakhruddin Gazzali, Bakhtiar Hasan, Md. Hasanul Kabir, and Moshiur Farazi. Improving action quality assessment using resnets and weighted aggregation. February 2021.

[7] Deepti Ghadiyaram, Matt Feiszli, Du Tran, Xueting Yan, Heng Wang, and Dhruv Mahajan. Large-scale weakly-supervised pre-training for video action recognition. May 2019.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. December 2015.

[9] Sandro Heiniger and Hugues Mercier. National bias of international gymnastics judges during the 2013-2016 olympic cycle. July 2018.

[10] Thomas Helten, Heike Brock, Meinard Müller, and Hans-Peter Seidel. Classification of trampoline jumps using inertial sensors. *Sports Engineering*, 14(2-4):155–164, November 2011.

[11] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. July 2012.

[12] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, November 1997.

[13] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. February 2015.

[14] Hiteshi Jain, Gaurav Harit, and Avinash Sharma. Action quality assessment using siamese network-based deep metric learning. February 2020.

[15] Glenn Jocher. ultralytics/yolov5: v3.1 - Bug Fixes and Performance Improvements. https://github.com/ultralytics/yolov5, October 2020.

[16] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. Large-scale video classification with convolutional neural networks. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, June 2014.

[17] Will Kay, Joao Carreira, Karen Simonyan, Brian Zhang, Chloe Hillier, Sudheendra Vijayanarasimhan, Fabio Viola, Tim Green, Trevor Back, Paul Natsev, Mustafa Suleyman, and Andrew Zisserman. The kinetics human action video dataset. May 2017.

[18] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. December 2014.

[19] Yongjun Li, Xiujuan Chai, and Xilin Chen. End-to-end learning for action quality assessment. In *Advances in Multimedia Information Processing – PCM 2018*, pages 125–134. Springer International Publishing, September 2018.

[20] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft coco: Common objects in context. May 2014.

[21] Joe Yue-Hei Ng, Matthew Hausknecht, Sudheendra Vijayanarasimhan, Oriol Vinyals, Rajat Monga, and George Toderici. Beyond short snippets: Deep networks for video classification. March 2015.

[22] Jia-Hui Pan, Jibin Gao, and Wei-Shi Zheng. Action assessment by joint relation graphs. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. IEEE, October 2019.

[23] Paritosh Parmar and Brendan Tran Morris. Learning to score olympic events. November 2016.

[24] Paritosh Parmar and Brendan Tran Morris. Action quality assessment across multiple actions. April 2019.

[25] Paritosh Parmar and Brendan Tran Morris. What and how well you performed? a multitask learning approach to action quality assessment. June 2019.

[26] Hamed Pirsiavash, Carl Vondrick, and Antonio Torralba. Assessing the quality of actions. In *Computer Vision – ECCV 2014*, pages 556–571. Springer International Publishing, October 2014.

[27] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. April 2018.

[28] Kurt W. Rotthoff. Revisiting difficulty bias, and other forms of bias, in elite level gymnastics. *Journal of Sports Analytics*, 6:1–11, 2020.

[29] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge. September 2014.

[30] Khurram Soomro, Amir Roshan Zamir, and Mubarak Shah. Ucf101: A dataset of 101 human actions classes from videos in the wild. December 2012.

[31] Josef Steppan. Sample images from mnist test dataset. `https://en.wikipedia.org/wiki/MNIST_database#/media/File:MnistExamples.png`, 2017. [Online; accessed April 26, 2021].

[32] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. September 2014.

[33] Yansong Tang, Zanlin Ni, Jiahuan Zhou, Danyang Zhang, Jiwen Lu, Ying Wu, and Jie Zhou. Uncertainty-aware score distribution learning for action quality assessment. June 2020.

[34] Du Tran, Lubomir Bourdev, Rob Fergus, Lorenzo Torresani, and Manohar Paluri. Learning spatiotemporal features with 3d convolutional networks. December 2014.

[35] Du Tran, Heng Wang, Lorenzo Torresani, Jamie Ray, Yann LeCun, and Manohar Paluri. A closer look at spatiotemporal convolutions for action recognition. November 2017.

[36] Yuxin Wu, Alexander Kirillov, Francisco Massa, Wan-Yen Lo, and Ross Girshick. Detectron2. `https://github.com/facebookresearch/detectron2`, 2019.

[37] Chengming Xu, Yanwei Fu, Bing Zhang, Zitian Chen, Yu-Gang Jiang, and Xiangyang Xue. Learning to score the figure skating sports videos. February 2018.

[38] Yuancheng Ye and Yingli Tian. Embedding sequential information into spatiotemporal features for action recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. IEEE, June 2016.

[39] Ling-An Zeng, Fa-Ting Hong, Wei-Shi Zheng, Qi-Zhi Yu, Wei Zeng, Yao-Wei Wang, and Jian-Huang Lai. Hybrid dynamic-static context-aware attention network for action assessment in long videos. August 2020.

[40] Barret Zoph, Ekin D. Cubuk, Golnaz Ghiasi, Tsung-Yi Lin, Jonathon Shlens, and Quoc V. Le. Learning data augmentation strategies for object detection. June 2019.