Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Distributed Neural Networks for Continuous Simulations on Mobile Devices

Thomas Hubatscheck

Course of Study:	Informatik / Computer Science
Examiner:	Prof. Dr. rer. nat. Kurt Rothermel
Supervisor:	M.Sc. Johannes Kässinger
Commenced:	November 5, 2020
Completed:	May 5, 2021

Abstract

Due to an increasing complexity of numerical simulations, calculating the results usually takes place on a server with access to large computational resources. To allow for a real-time visualization to users in an AR setting, these simulations shall run on the mobile device itself. Therefore, a way to enable the execution on a resource-constrained device is necessary. The goal is to compute the results of the simulation with a surrogate model in the form of a NN. The model has to comply to latency and quality requirements for an accurate visualization of results.

This thesis proposes the use of a distributed network architecture. Hence, the interaction of a NN on the local device with a NN on a nearby server was simulated. LSTM layers and their ability in a continuous setting was studied to choose the type of network to replace the simulation. The mobile device was able to request accurate updates from the server during execution. Two operators were derived by analyzing the behavior of received updates in crucial input areas for the mobile device. A decision operator determined the frequency of update requests. The merging operator handled the combination of outputs with respect to a predicted quality and the current delay of received updates. For the latter, the local results are decoupled from the execution and serve as a way to adjust the received update. Different approaches to continue delayed updates with the corresponding local changes to fit the current local step are proposed and evaluated. For this, different artificial connection delay and offloading settings are considered.

Using LSTM NNs increased the accuracy and showed a more stable execution compared to NNs without these layers. The proposed methods to merge results decreased the overall MAE from 5% of the local NN down to 2% with the help of updates every 10 steps, if a delay of 10 steps was assumed. This is an improvement of 60% compared to the local execution without updates. The quality-sensitive merging operator was also able to prevent a decrease in quality for bad connection settings by switching to a local-only execution when detecting that the quality of updates decreased. The average time elapsed to produce a single output on the mobile device with the ability to request updates decreased by 63.5% compared to the average inference time of the LSTM NN.

Contents

List of Figures

List of Listings

List of Algorithms

Acronyms

AIT average inference time. 21

AR augmented reality. 15

CNN Convolutional Neural Network. 20

DNN Dense Neural Network. 16

LSTM long short-term memory. 15

MAE mean absolute error. 21

ML machine learing. 15

NN Neural Network. 15

RNN Recurrent Neural Network. 20

1 Introduction

Augmented and virtual reality applications visualizing the results of a simulation, such as the muscle activation of a human arm, require complex numerical simulations as a basis to compute accurate results. Mobility and flexibility of these applications are decisive aspects for a good user experience. Especially in augmented reality (AR) the user should not be bound to one place for the simple reason that the AR applications are built to extend the reality, wherever they are needed. Therefore, the deployment of these complex simulations to mobile devices is inevitable. Mobile devices such as smartphones or mixed-reality devices are resource-constrained on multiple variables. These portable devices come with inherent constraints affecting performance and execution of heavy computational operations. Constraints include the lack of computing resources and storage space, which are necessary to stem large simulations as well as running on a finite energy source. This requires operations on the device itself to be efficient.

A way to compute these simulations more efficiently is with the use of a Neural Network (NN). It works as a surrogate model to handle a specific task. In this case the simulation underlying this thesis calculates the muscle activation of a human arm based on input parameters such as the elbow angle, angle velocity and angle acceleration of an arm movement as well as an optional weight added to the hand. The muscle activation of an observed arm is later visualized for a user who is wearing a mixed-reality device such as the *Microsoft HoloLens*. The user experience depends on accurate, low-delay outputs for a real-time presentation. Thus, the NN used to replace the simulation should be able to meet quality and latency requirements to ensure these goals. In contrast to the numerical simulation which outputs highly accurate results the accuracy of NN depends on many factors. They implement a special case of machine learing (ML). Hence, the NN has to learn the behavior of the data with the help of training inputs. A trained NN can outperform other techniques and is therefore used across various applications, such as speech recognition or image interpretation, as a powerful tool to stem large computations. Many types of NNs exist, each able to handle specific tasks better than another type. Thus, the network can be defined particularly for a single task to achieve an accurate result. Since the simulation, considered in this thesis, utilizes continuous data, the NN to replace the simulation shall use a special NN implementation, namely long short-term memory (LSTM), to interpret sequential inputs. These LSTM layers are widely used for speech recognition and text interpretation. In newer cases, applications with similar goals, such as muscle activation during a gait cycle [Dao18], also use LSTM NNs to compute accurate outputs.

To investigate the ability of these networks in muscle activation computation, differently shaped and sized NNs are tested on accuracy and latency. Both of these properties are crucial for a real-time application to visualize the outputs in an AR environment. Results of this experiment show that LSTM NNs are suitable for execution on devices with access to sufficient computational resources and can produce accurate results. Due to the nature of their architecture, running them on a resource-constrained mobile device is not possible without negatively affecting latency of the outputs. Based on that, two NNs with different size and shape are deduced to function as surrogate

models for the simulation. To account for a resource-constrained environment, a less accurate, but faster Dense Neural Network (DNN) for the use on a mobile device is chosen. On the server, a more accurate, but bulkier LSTM NN can be used due to more available computational resources.

For further improvement of the computation results on the mobile device, both of these NNs form a distributed network architecture. The mobile device shall be able to request accurate outputs from the remote network. These updates can be computed with larger resources but still have to fulfill latency demands for the real-time execution. This idea is also used in other works, and overall lowered the execution time by a factor of 131 with energy saving on the mobile device [DHS+18] and improved results in [CVV+16]. In [DHS+18], communication is handled by a wireless network and not explicitly measured, but still has effects on the overall runtime. Both of these works neglect local results for steps where offloading occurs. Here, a method is proposed to merge local outputs with accurate updates from the remote server. Before being able to combine the outputs, received updates, that may be already outdated due to network latency, have to be adjusted to fit to the current step in the local execution. To adjust delayed updates, the proposed merging operator considers the local results on the mobile device to approximate the change of the remote NN between requesting and receiving the update. Methods to combine the continued update with the corresponding local output range from simple averaging to a more sophisticated approach which takes age and delay of updates into account. For the latter, the behavior of continued updates in three different input areas is analyzed. Depending on the observed change in accuracy over multiple steps after receiving an update, different weights can be assigned to the local and remote outputs before merging. These observations do not only matter for combining the two outputs, but also help decide when to request updates. Additionally, a single update shall be continued and considered for merging with the following local results for as long as its quality is predicted to be above the local outputs' accuracy. Therefore, by continuing the update for steps even after it is received by the mobile device, it is possible to make the request of updates less frequent and save communication costs.

Because of the inaccurate outputs of the local NN and the constant change of inputs, a decrease in accuracy of continued updates is inevitable. This means that at a certain step after receiving an update the continued values can get worse than the local results. Based on this behavior, the decision operator can modify the frequency to request updates. With this, accurate updates can be received well before the drop-off point to ensure a precise execution. The thesis covers approaches to derive suitable offloading and merging methods for the given simulation. With these thoughts in mind, the distributed network architecture is evaluated on different parameter settings affecting request of update frequency and the delay of receiving updates. In total, three merging methods are proposed and their ability to improve the local execution is measured with respect to the overall accuracy.

The structure of this thesis is as follows: In the second chapter the most important background information will be introduced, followed by an overview of related work in chapter three. The fourth chapter contains the problem statement which leads to the comparison of LSTM and dense networks in chapter five, followed by the derivation and evaluation of the merging and decision operator for a distributed network architecture in chapter six. The thesis is finally concluded in chapter 7 which also presents an outlook and suggests future continuation of the developed methods.

2 Background

This chapter covers basic knowledge about the challenges of running numeric simulations on resource-constrained devices and the process of applying ML to train algorithms to solve specific tasks. A special type of class, which utilizes ML, is the so called NN. The basic structure and workflow of these networks is described and is also a short introduction to LSTM NNs. Finally, the specific Hardware and Software to implement the experiments is described, followed by the evaluation method to measure the accuracy and inference time of different approaches.

2.1 Simulations on Mobile Devices

Numerical simulations can be of complex nature depending on input parameters and desired accuracy of the result. Usually, servers handle large simulations because they have access to larger resources than mobile devices. In this thesis the simulation should be able to run on a resource-constrained device. Having mobile devices as a part of a system usually comes with several problems and challenges to overcome. Mobile devices have less usable resources for computations than their stationary counterparts like servers or PCs. Also, they are running on some finite energy source, further limiting their runtime and thus require efficient computing [Sat96]. An alternative way to handle simulations is to replace them with NNs. These NNs are capable of computing accurate outputs, while requiring less resources. This can be further exploited by deploying a computationally light NN on the mobile device with results in lower quality. An idea to improve these results is to send frequent updates to the device. These updates should be of better quality and therefore require the computation on the remote infrastructure to be accurate but also quick to ensure a timely transmission. Since the server does not have access to the inputs during execution, the mobile device on its own has to request these updates by transmitting the current input data to the server. The NN on the server then computes the corresponding output and sends the update to the mobile device. Therefore, mobile devices have to be able to connect to other devices which can be located very distant from the end user, hence presuppose a reliable connection. Due to a possible change in quality and availability of connection depending on the location and overall traffic, this cannot always be granted. Thus, the mobile device should be able to work on its own for a period of time before computations get worse [AGH18].

2.2 Machine Learning

Machine learning is about the process of training a machine to perform a specific task. The process first acquires sufficient training data to ensure an accurate training phase. The algorithm to train the underlying machine observes outputs by the machine and compares them to the desired output, defined by the training data. Learning means that the parameters inside the machine change in a

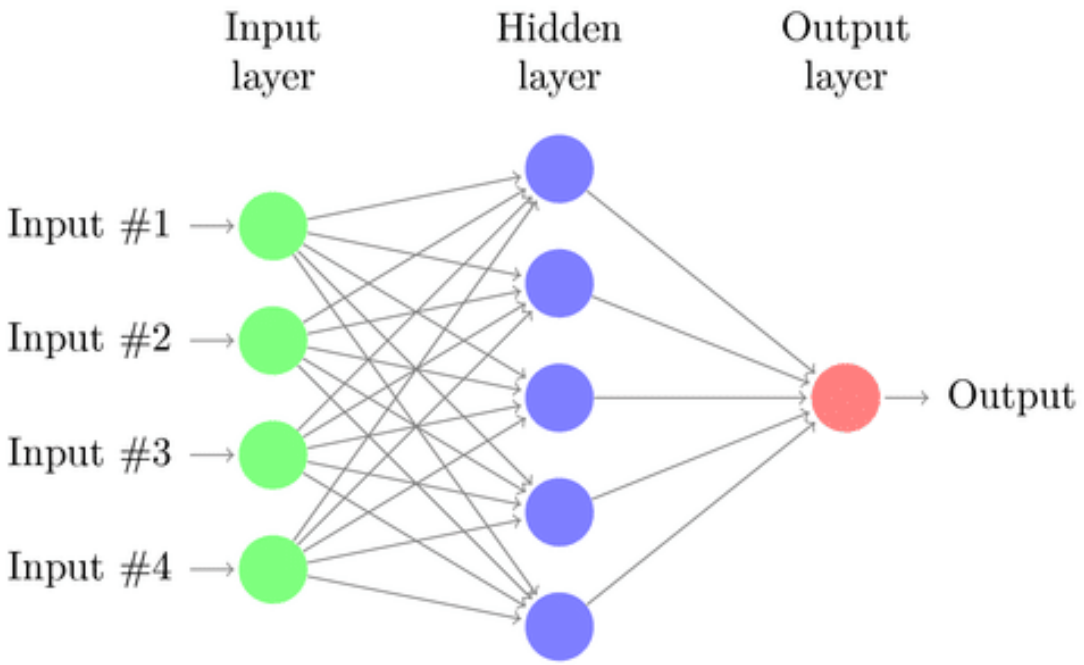


Figure 2.1: Architecture of a Neural Network [Dil21]

way that produces a more accurate result the next time it sees similar data. Stopping the learning procedure at the correct moment is key in preventing the machine to over-fit to the training data. Overfitting can worsen the behavior of the machine, if it is exposed to new data not part of the training procedure. Thus, the machine should not memorize the outputs for inputs from the training data but rather learn characteristics and correlations between input and output for a generalized algorithm. Generalization is a better approach to handle extreme cases in the computation than a hard-coded algorithm. Also, machine learning bypasses the need to cover each exception, which we would have to in a hand-written algorithm. The machine is chosen from a wide spectrum of machine classes and depends on the task it has to fulfill [Sim18].

2.3 Neural Networks

A way of implementing machine learning is by the use of NNs. They consist of layers, which on their own contain nodes. Nodes of consecutive layers are connected by weights and represent single values during execution. The first layer of a NN is the input layer and marks the starting point for propagating the values through the nodes until it reaches the final layer. The input layer has the size of the amount of input values, while the last layer contains as many nodes as the specified task requires to represent its output. A simple NN structure can be seen in Figure 2.1. There, the input layer consists of four nodes. Each of these nodes is connected to every node of the following layer, labeled as hidden layer. These nodes are then all connected by one weight to the single output node forming the output layer. For example, in a classification task depending on two classes, the last layer would consist of two nodes. At inference, the values in each node change by forward-propagating with the help of outgoing weights. In the simplest way, the following node

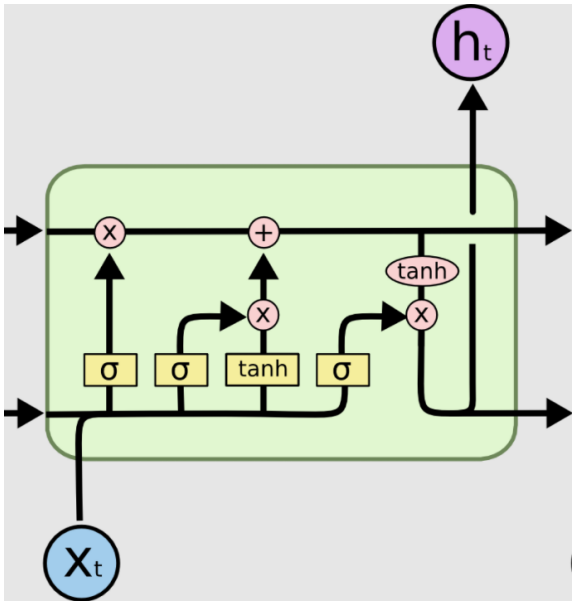


Figure 2.2: Structure inside a LSTM-Cell [Ola15]

multiplies the value of the previous node by the weight and is able to add a bias to the result. The exact weight is determined prior to execution and is part of the training procedure. The training of these parameters inside a NN is done by processing data through the network. Data consists of inputs and the corresponding true outputs. The difference of the results are compared to the ground truth and a so-called *loss* is computed by the training algorithm. This loss is back-propagated through the network to adjust the weights inside the NN. If done correctly, new computations better represent the desired output. Overfitting to the training data can happen by continuing to train the NN, even if the quality is not improving any more [Nie19]. This can cause the accuracy of the NN to decrease, when feeding in new data that is not part of the training process.

NNs of different types perform better or worse depending on the task they have to fulfill. Common layers include fully connected, dense layers in which case a node of layer x is connected by weights to all nodes of layer $x+1$. Dense layers usually perform operations based on linear activation functions. A convolutional layer implements the use of filters, which originate from image processing tasks and can enable the NN to assign a certain class to an input image [KSH17]. In a recurrent layer, nodes not only have connections to nodes in the subsequent layer but are also connected to themselves. This adds a memory to the NN and can be utilized in tasks which work with continuous data, such as sentence interpretation or speech recognition. A special kind of recurrent layer is the LSTM layer. It is a more complex implementation of the standard recurrent type and enables the network to remember more than one time step. Thus, it can detect and recognize long-term dependencies. This layer has two paths inside a single LSTM cell which can be seen in Figure 2.2. The top path represents the cell state and represents the memory. The bottom path handles new input values and applies operations to decide what information to add to or remove from the memory. The second path also computes the output of the current step by considering cell state, the memory, old output and the processed new input to this node. Paths inside the cell by default utilize *sigmoid* and *tanh* functions to process new data, while gates are part of merging data from the two paths [RPNU19]. Standard NNs usually have one time step as input variable. When using LSTM layers, the amount

of dimensions in the input layer increases by one. This additional dimension can be used to have multiple time steps as input variable to the LSTM layer. With this it is possible to for example consider the previous three input values for a single output of the NN. This additional parameter setting depends on the nature of the underlying data and has to be carefully configured to fit the task.

The combination of multiple layers then forms the whole NN, resulting in various characteristics. If it only consists of the most basic dense layers it is called a DNN. A network utilizing convolutional layers is called Convolutional Neural Network (CNN) and operates in image or video processing tasks by applying filters to extract information from different parts of the input. Usage of recurrent layers forms a Recurrent Neural Network (RNN), which is applied when dealing with continuous data. A special form of RNN is the LSTM-NN. It consists of LSTM layers with the ability to detect dependencies across multiple time steps. The different types of layers can be arranged in arbitrary order with each formation leading to new properties of the NN. It is up to the user to define a desirable NN that is suitable for the task it has to perform.

2.4 Hardware and Software

TensorFlow and *Keras* are the main components for the development and implementation of the proposed methods. *TensorFlow* is a Google Brain project that originated in the year 2011. A first working framework was called DistBelief and allowed for a distributed training and inference system. Due to the success of this framework in various tasks, such as image or video classification, speech recognition, sequence prediction or reinforcement learning, they started developing *TensorFlow*. It implements its computations as data flow graphs, allowing for a more flexible execution compared to their first framework. These graphs consist of nodes which are connected to each other and implement the use of operations, such as *add*, *matrix multiply* or *Sigmoid*. The latter operation is widely used in NN environments. *Tensors* are multi-dimensional arrays which are passed along the edges and contain the outputs of a specific node. A *Kernel* is the implementation of an operation, able to be executed on the CPU or GPU. The programming interface communicates with the system by using a *Session*. This can be used to add nodes and edges to the graph. Also, this *Session* implements a method to start the computations inside the graph [AAB+16].

On top of this architecture, *Tensorflow* includes the *Keras* API. This enables the user to build NNs more easily due to the available layer architecture. In *Keras*, it is possible to design and train NNs. A *Keras* model consists of combinable modules, such as neural layers, activation functions for these layers or optimizers. There are two types of models available to use. The sequential model is suitable for stacking layers where each layer has exactly one input tensor and one output tensor. The second type is the functional API. It can handle models with multiple inputs, shared layers or even a directed acyclic graph of layers. *Keras* also implements the training and inference of models with the built-in functions *fit*, *evaluate* and *predict*. Trained models, ready for execution, can be saved and loaded with *Keras*. With that, the model can be deployed on different devices without the need to retrain the network [Ten21].

Following are the concrete specifications of the involved hardware and software to implement and evaluate the experiments. To allow for an accurate comparison of the used NNs the evaluation takes place on a single device. The machine underlying all development and measurement is a Personal Computer with an *Intel Core i7-4790* quad-core CPU with a base frequency of 3.6GH and

16GB DDR3 RAM. The operating system of the Personal Computer is *Windows 10 Home Version 20H2* and *PyCharm 2020.3.3* is the IDE to implement the code of this thesis. For the programming of the methods and experiments, *Python 3.8* of the Miniconda installation is used. Miniconda also functions as the package manager for the python environment, with which *TensorFlow 2.3.0* and other miscellaneous libraries, such as *Numpy 1.19.2* and *Matplotlib 3.3.4*, can be installed to evaluate the experiments.

2.5 Evaluation Method

For the comparison of LSTM-NN to Dense-NN, two different metrics are considered. The first one is the mean absolute error (MAE), which measures the difference between the output of the NN and the output considered to be true according to the validation data. The equation to compute it is

$$\text{MAE} = \frac{1}{n}\sum_{i=0}^{n-1} |\hat{Y}_i - Y_i| \tag{2.1}$$

with parameters n, the total number of outputs, $\hat{Y}_i$, the predicted output for input i and Y_i, the true output for input i. The MAE in the evaluation of different merging techniques represents the average error across the whole validation data which consists of 10 000 input tuples. To better understand the meaning of the resulting MAE values it is necessary to clarify what the value range of the outputs in the validation data is. Each of the five different muscle activation computed by the NN the values are between zero and one. A low MAE means that the NN can accurately predict outputs for a given input. Secondly, the evaluation method measures the average inference time (AIT) of the NNs by computing the time elapsed for the NN to predict the whole data set, then dividing by the amount of samples in the data set. This returns an approximate value for how fast the NN can compute a single output. These two values indicate the overall performance by having a trade-off between accurate results, measured by the MAE, and fast computation of the outputs, measured by the AIT. In the perfect case, the NN has low error outputs, which it can compute quickly. The second experiment is studying the distributed network architecture and the improvements in accuracy, made possible by receiving accurate results which are then merged with local results. Therefore, it only uses the MAE as a metric. Also, since no actual communication is part of this experiment the distributed setup utilizes a static test setting. This means that the evaluation of different merging techniques takes place after the NNs have already computed all outputs for the data set. The data is then sequentially handled within the evaluation method to simulate a real execution of the two involved networks. Different offloading and merging settings and methods, mentioned in detail in Sections 6.1 and 6.2, are measured and compared to learn more about the ability to improve the accuracy of the local execution.

3 Related Work

The following sections cover related work in the field of distributed network architecture, offloading methods, utilization of LSTM networks to compute simulations and techniques to improve the accuracy of NNs. Various relevant articles are presented and put into context with respect to this thesis, showing similarities and differences in the approach to solve existing problems.

3.1 Distributed Network Architecture

There have been many attempts to distribute computation across multiple devices. These approaches also consider the mobile device to not be able to do heavy computations reliably. Thus, the calculations are offloaded to the cloud or edge cloud. Dibak et. al. [DHS+18] use the *HoloLens* as a resource constrained device for running an interactive mobile simulation. In a basic approach, a framework for distribution to a remote server based on the reduced basis method is proposed to improve latency and energy consumption. The heavy computational part is calculating the reduced basis. This is done once prior to execution for known input variables. If the input variables change during execution, the mobile device can request updates to the reduced basis. In all approaches, the received reduced basis is stored on the mobile device to easily access it while execution, and thus allows the simulation to meet quality requirements. They managed to achieve a 131 times faster execution and consumed 73 times less energy compared to offloading everything to the server.

In other approaches, not only a cloud server is used to offload computations, but also edge cloud devices are used to help distribute. These edge cloud devices are usually closely located to the mobile device and can thus benefit from a smaller latency considering communication, while still having considerable more available resources. This property is used for image classification by Teerapittayanon et al. [TMK17]. If the mobile device realizes that it can not reach a certain accuracy, the mobile device forwards the output to the next higher level. Results of this work show that it is able to meet requirements with higher fault tolerance, but having 20 times less communication, compared to standard offloading methods.

In [MCN+17], Mao et al. even go one step further and utilize multiple mobile devices to partition the Deep Neural Network. With this, they reduced computation cost and memory usage for a single device. They were able to accelerate computation by 2.2-4.3 times when going from two to four worker nodes and also achieved a reduced data delivery time. On the lowest level, the NN can be configured to support detecting inaccuracy in its computation and decide that offloading is beneficial. Multiple exit points are implemented in-between layers of the NN, which can be used to offload the computation to the server [SZMZ20]. This resulted in a latency reduction in the overall interpretation of a point cloud.

The work that is most similar to this bachelor thesis is the so called 'Big-Little approach' by Coninck et al. [CVV+16]. It proposes the usage of a big and a small network for a classification process. While the small network is trained on some output classes on the device, the big network can be used to determine the rest of the classes with high accuracy on the server. The goal is to decide, based on the output of the small network, whether it should offload to the big network and then use the big network to classify images that are part of a high-priority class. They tested this on a Raspberry Pi and an Intel Edison and found out that using this principle of prioritizing yields good results on the classification task on the little network. While this thesis also uses a small, worse network and a big, better network for the task at hand, it is still different in not having an output that can be prioritized. Another thing to mention is that the results of the big network are considered to be true and the computations on the small network are discarded. This is also the case for all the works mentioned above. Merging of results from the cloud server is not a considered factor when using distributed architecture as it is good enough already for the goal of reducing computation on the device and improving latency and no use case for time series data is mentioned in the related works. In general, it is a rare case of a distributed setting to work with numerical simulations. Thus, this thesis is trying to combine the benefits of distributed setting by not only offloading computation, but also utilizing the results with higher accuracy.

Another challenge is that a lot of works focus on the training process when talking about a distributed NN. There are various works on how to use distributed training. One of them is [DCM+12] by Dean et al. and proposes a framework called *DistBelief*, which utilizes a cluster of machines to train DNNs and found multiple effective strategies. Typical bottlenecks are communication overhead, parallelization of matrix operations and training data distribution [KP16]. The training phase of NN should not be neglected but is not further considered. The focus of this thesis is to determine a way of utilizing the results received from the remote NN to improve the overall performance of the distributed architecture.

3.2 Offloading

There are several approaches on how to offload efficiently and how to figure out the best decision when and what to offload. This ranges from using an inequality that relates computation offloading system parameters to arithmetic intensity of a computation to determine which computations benefit from offloading [MM16] over using Deep Reinforcement Learning [HBZ18] and Feed Forward Networks [YCB+20]. Another approach is to predict the energy saved on the mobile device by offloading a certain task or to offload tasks which require the least data rate [WHY+17]. Dibak et al. [DDR15] take advantage of the ability to offload to a server and managed to minimize the usage of the mobile device. They improved the energy consumption, while still fulfilling deadlines to receive an accurate result in time. This work particularly is in direct contrast to the goals of this bachelor thesis. The main aim is to minimize the server communication, while also maintaining an accurate computation of the simulation on the mobile device by using the remote NN to request frequent high-quality updates.

3.3 LSTM-Networks for Motion and Simulations

LSTM-Networks are widely used for applications with continuous data. In most cases, they function as a model to handle speech recognition and text interpretation. Also, modeling turbulent flow LSTM-NNs can be used because they can interpret temporal dynamics of turbulence [MG18]. Cecchini et al. [CLS+14] trained a NN with data from solving the inverse problem of evaluating a kinematic model while cycling and the force distribution along the limb to gather information about the applied muscle forces. NNs are used to compute angles and forces based on the movements of knee, hip and ankle. The resulting NN is not a LSTM-Network, but still achieves an accuracy above 99%. In another case an actual LSTM-Network is used to predict skeletal muscle forces from joint kinematics data during a gait cycle where the input data is evaluated separately for three muscles [Dao18]. A relative root-mean-square error of less than 10% is achieved by this method. This is in direct correlation to this thesis because a LSTM NN functions as a mean of predicting muscle forces based on similar input variables. The difference is that for this thesis the LSTM based NN runs on the remote server because a resource-constrained mobile device is not able to run these computationally heavy networks. Therefore, this thesis uses a DNN planned for future deployment on the mobile device, while a more powerful LSTM NN for execution on the remote server is used to compute accurate updates.

Other works focus on classifying and predicting movement sequences with the use of LSTM-Layers. Bao et al. [BZX+19] use a CNN-LSTM-Framework for wrist kinematics estimation based on surface electromyography to solve sequence regression. This approach outperformed other machine learning methods that do not utilize LSTM properties. Carrara et al. [CESZ19] also focus on predicting movement and annotate actions like walking or doing a cartwheel to sequences of frames. The whole body is analyzed and input into several LSTM NNs, each having distinct properties and architecture. Improvements in quality and inference time have been achieved with this method. This shows another possible benefit of LSTM NNs that is not part of this thesis. The tasks in these works is different to computing outputs of the underlying simulation to this thesis. For prediction tasks, outputs are of the same nature as inputs and LSTM NNs can more easily model dependencies across multiple steps. Contrary to that, the simulation at hand uses observations of a human arm and maps these to the muscle activation in the current position.

3.4 Improving Results of Neural Networks

Improving and combining different results to increase the overall performance is the main goal of this thesis. When working with Neural Networks, there are multiple approaches available on how to combine results of different networks. Hansen and Salamon [HS90] propose the use of a majority voting scheme, where the result that most networks agree on is considered to be an accurate output. A higher accuracy than using only one of the given networks has been achieved by this work. Another method is using a simple average over five different networks. This resulted in a better generalization and a higher fault tolerance [LS89]. There are also more advanced techniques available like the theory of evidence, with which statistical information about the relative classification accuracy of several networks is gathered. Based on that, different weights are applied when merging the results of these networks [Rog]. Similar to that, Perrone and Cooper [PC95] proposed a method called *Generalized Ensemble Method*, which is a weighted combination of

multiple results to minimize the mean square error of these networks. They found out that it yields better results than averaging outputs or only taking the best individual result. This thesis implements similar approaches and tries to better understand the nature of combining results from individual NNs with different prediction strength. The proposed merging operator contains simple methods like averaging, but also more sophisticated methods that account for delay and quality of outputs. Received updates additionally shall be continued with the help of the local behavior and thus allow for a consideration across multiple steps.

4 Problem Statement

The problem underlying this thesis is to enable the execution of a surrogate model of an interactive and complex numerical simulation on a resource-constrained mobile device. The outputs, produced by the simulation, are then visualized in real-time for a user in an AR environment. Therefore, the results have to be computed efficiently and accurately. A short computation time ensures a high rate of results, while outputs also have to be accurate for a correct presentation to the user. Both properties are necessary for a smooth and exact visualization of results.

Currently, mobile devices do not have the required computational resources to properly execute these numerical simulations, without impacting the accuracy of results. Hence, a different way of computing the outputs of the simulation has to be considered. A more efficient approach is the use of NNs, functioning as surrogate models to replace the simulation on a mobile device. This is combined with a server-assisted execution of two NNs to form a distributed neural network architecture. Here, a less complex, resource-efficient NN is proposed for the use on the mobile device to form the local model, while a more complex NN with access to more resources is thought to be deployed on a server to form the remote model. Results, computed on the server, are expected to be of higher quality, and therefore shall be used to improve the accuracy of the local model. With this working properly, the mobile device will be able to produce precise results for the visualization, while also bypassing the need to have complex simulations running on its own resources.

To achieve this goal, the thesis tackles the following problems of having an accurate distributed execution. First, different implementations of NNs, particularly the use of LSTM layers compared to the use of only dense layers, are analyzed with respect to quality and latency. This is investigated by changing the number of layers and number of nodes for both types of NNs and then measuring the accuracy and inference time of resulting models. Secondly, designing a method to combine local and remote results is key for an improved performance. A substantial problem of using remote updates is, that due to the inherent communication cost, these updates arrive with a delay on the local device. Since the computation of the local model continues while waiting for the update, the quality of the requested update decreases with an increasing delay. This is because of continuously changing input values. Therefore, upon receiving the update, it has to be adjusted to fit to the current step in the execution before merging.

A method considering local output changes is proposed for continuing the remote update. The merging operator handles both the adjusting and the final combination of the local and remote outputs. Merging happens, in the simplest way, by taking the mean of local and remote output, or, for a more complex method, by assigning weights to each output with respect to the behavior of continued updates in different input areas. A single update is not discarded after the use in the step upon arrival, but rather continued for the following steps to continuously benefit from the high quality and decrease the need for communication between local and remote architecture. The counterpart to the merging technique is the decision operator. It is responsible for the frequency of requesting updates. This, like the merging operator, runs on the mobile device simultaneously. The

proposed offloading strategy uses fixed update frequencies across the available data for evaluation, but also introduces a way of dynamically determining a required minimum request frequency to ensure an improved execution.

5 Comparison of LSTM and Dense Networks

This experiment sheds light on the capabilities of LSTM-Networks in the use case of continuous simulations. The goal is to train and evaluate NNs with different amount of layers and nodes present in each layer. The thought is to compare the NNs based on AIT and MAE by looking at NNs with similar architecture. The following subsections show the implementation and results of the creation and evaluation of 500 distinct NNs.

5.1 Experiment Setup

Training and evaluating NNs takes a lot of time. To cover all possible unique NNs is mere impossible to do because of all the different parameters that can be changed during the creation of a NN. Some of these parameters, including amount of layers, amount of nodes, layer types and corresponding activation function, are covered in Section 2.3. Since there is no formula to create the perfect NN for a specific task, this comes down to applying knowledge about the task and finally trial and error. In this experiment, NNs with three to seven layers are considered to produce accurate results. The structure of these networks follows a specific pattern. Early layers contain fewer nodes than middle layers, which on their own contain more nodes than layers close to the output. This is no claim that it is the best structure but rather what has worked in previous tests and is now examined, given data from a continuous simulation.

The next part describes the method to generate different network architectures. A simple loop creates the aforementioned NN structures and saves them in arrays for further processing. For instance, a NN with five layers has i, $\lceil 1.5 \cdot i \rceil$, $2 \cdot i$, i, 5 nodes respectively for each layer and uses $i \in \{1, ..., 50\}$ to produce the fifty different NN shapes. Notice how the last layer is a fixed number because the output in the data set consists of five values. Thus, each NN has to have an output layer with five nodes. The NNs consist of l layers, with l ranging from three to seven. This method outputs an array that contains 50 unique structures for NNs for l layers. The array functions as the input into a method to create NNs consisting of l dense layers or l-1 LSTM layers, combined with one dense layer. Dense NNs are build by defining a *TensorFlow* sequential model and adding layers according to the input array. All but the last layer use a rectified linear activation function. The final layer, which is computing the output, utilizes a linear activation.

LSTM NNs follow a similar scheme. Again, a *TensorFlow* sequential model is defined and l-1 LSTM layers are added before adding a dense layer as the output layer. The LSTM layers use the hyperbolic tangent activation function, with the dense layer using a linear activation. For all layers prior to the last LSTM layer the option to return sequences has to be set to *True* so the next LSTM layer receives a correct input shape. The resulting models are compiled with the *TensorFlow Adam* optimizer and the mean squared error is defined as the loss function. The training process of each model uses a maximum of 50 epochs to avoid overfitting. Another measure taken to lessen the

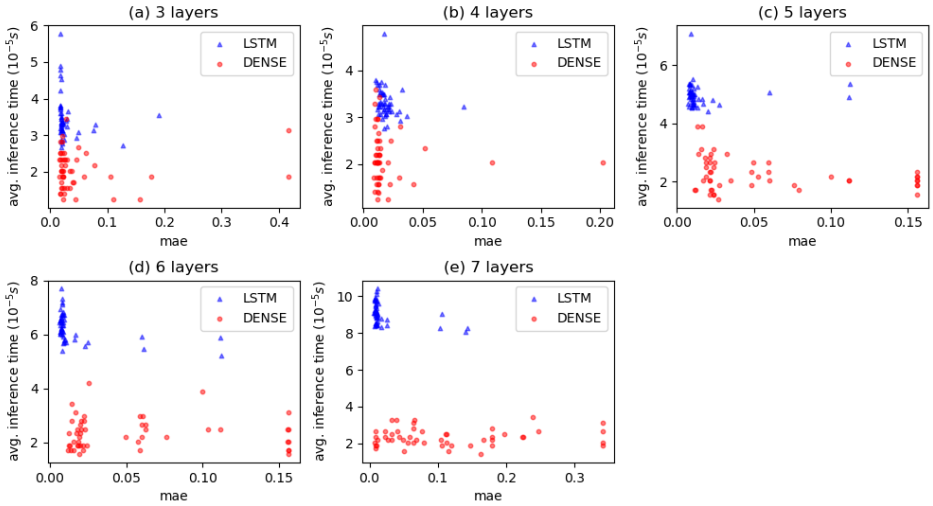


Figure 5.1: LSTM and Dense Networks Comparison

chance of overfitting to the training data is to define a callback inside the *TensorFlow fit* function, which stops the training early, if a user defined criteria eventuates. In this case, the training process stops, if the MAE of the training data has not changed by $1 \cdot 10^{-4}$ over the course of the last two epochs. The early stopping callback prevents the NN to stay in a local minimum of the loss function for longer than necessary. Although it is possible to use multiple time steps as input of a LSTM NN, the networks trained in this experiment do not use this feature. This is due to the fact that in earlier tests, networks that used multiple time steps could not achieve a significant increase in accuracy, while the AIT increased. The behavior of NNs with a different amount of input steps is not further investigated in this thesis. Thus, in this thesis, both dense and LSTM NNs utilize one input frame to compute results. The final function takes an array containing the trained models from the previous step and computes the AIT and the MAE of each model as described by the evaluation method in Section 2.5.

5.2 Results

Evaluation of multiple different network structures provide an overview on the abilities of LSTM NNs and dense NNs in computing outputs for a continuous simulation. For each type, fifty networks with three to seven layers, 500 in total, are trained as described in the previous section and then evaluated on the validation data. Figure 5.1 shows the results of the evaluation. The subplots (a) to (e) display the different outcomes of utilizing three to seven layers respectively. Each subplot consists of fifty dense networks (red) and fifty LSTM networks (blue) for one layer count. The horizontal axis represents the MAE across the validation data. The vertical axis shows the AIT in 10^{-5} seconds.

When looking at the lower layer numbers, concretely plots (a) and (b) of Figure 5.1, no advantage of using LSTM can be seen, as the majority of networks achieve a MAE of 0.020. DNNs have an AIT of 15-25 μs, while the mean value for LSTM networks is between 30-40 μs. Some networks even

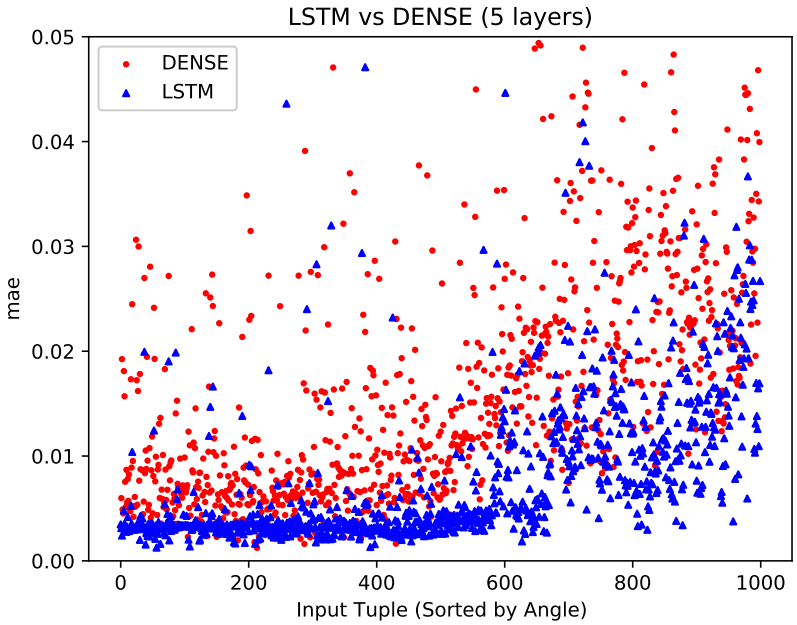


Figure 5.2: MAE Comparison of Networks with Five Layers

drift off to higher numbers. That shows that for a low amount of layers we can expect an increase in inference time with a factor of two. This trend also continues when using five to seven layers. While dense networks carry on having an AIT of 20 μs, their LSTM counterpart has an increase in inference time from 50 μs on average when using five layers (subplot (c) in 5.1) to 90-100 μs in the case of networks with seven layers (subplot (e) in 5.1). That shows a five times higher inference time than dense networks with the same network architecture. On the plus side, the MAE drops below 0.010, with many networks reaching values of 0.007. DNNs reach values of 0.0125, which is minimally worse than the performance of LSTM networks. In the case of seven layers, we can see that many dense networks do not achieve a MAE below 0.10. Contrary to that, the MAE of only four LSTM networks rise above that threshold.

To get a better understanding how well the different types of networks perform when using the same layer and node structure, the following showcases the difference in performance of a single LSTM and dense network with five layers. Each NN contains layers with 20, 30, 40, 20, 5 nodes respectively, starting from the first hidden layer and ending at the output layer. Figure 5.2 displays the variation of results by taking a look at the MAE for single inputs to both NNs. The horizontal axis represents each fourth input tuple of the validation data which is sorted by the input angle. The vertical axis shows the MAE of each step. Red dots and blue triangles correspond to values of the dense and LSTM network, respectively. In the first half of the validation data, both networks have MAE values close to zero. For the DNN, the majority of values are below 0.01, while the MAE values for the LSTM NN are below 0.004. Additionally, the DNN has more outliers compared to the LSTM network. This behavior is acceptable for both NNs as the overall range of output values for the validation data is between zero and one. In the second half of inputs, the differences of both types become explicit. The MAE values of the dense network spike into regions of 0.02-0.04, while most values of the LSTM network stay below 0.02. Both NNs perform worse for higher input

angles but the LSTM outperforms the dense counterpart in the considered areas. This behavior is also represented in the total MAE of the NNs. The DNN used for this comparison has a MAE of 0.0178 while the LSTM network achieves a value of 0.0084 and thus produces more stable and accurate outputs.

6 Distributed Neural Networks with Offloading and Merging

Based on the results of the previous experiment, two networks are selected to be used in the distributed network setting. The goal of this chapter is to learn about the ability of frequent, high-quality updates in improving low-quality outputs by merBging the two separate results on the mobile device. For this, the methods described in section 6.1 and 6.2 are evaluated on different parameters. Before evaluating, two NNs are determined to replace the simulation. One for the execution on a mobile device and one to function as the NN to handle update requests. Following the constraints of mobile devices as outlined in section 2.1, this test considers a NN that is quick at producing outputs for the simulation. The downside of having a fast inference time comes with the challenge of achieving accurate outputs.

Therefore, a DNN shall be used as the component on the resource-constrained device. This NN consists of a single hidden layer with ten nodes, followed directly by the output layer with five nodes and forms the local model. The network has an AIT of 12.5 μs and achieves an average MAE of 0.0511 on the validation data. A LSTM NN is trained to function as the remote counterpart. Since it is responsible for producing accurate results, the NN uses a more complex architecture. It consists of seven layers in total, six of them being LSTM layers with nodes distributed across these layers, in a way that follows the same principle as used in the Chapter 5. Here, this network uses 40 nodes in the first hidden layer and has the maximum amount of nodes in the third layer. This layer consists of 100 nodes before declining to five nodes in the seventh layer, forming the output. The network has an AIT of 100 μs, and therefore is 8 times slower than the counterpart planned for the mobile device. The overall MAE is at 0.0107 which is close to five times better than what the dense network can achieve on the validation data.

One thing to mention is that the goal was to define a small and fast NN for the execution on the mobile device to benefit from remote updates. Although it is possible to define a NN that produces more accurate results, this would also mean an increased resource usage. Therefore, a simple NN shall benefit from accurate updates to have an accuracy that is close to the performance of a bigger NN running on more resources. The overall difference in the amount of trainable parameters of both NN correlates to the resulting accuracy of the two NNs in this distributed setting. The DNN only has 105 trainable parameters, while the LSTM NN has 142 000 parameters, adjustable during training.

Taking a look at the communication between the devices in a distributed setting, an inherent delay exists between requesting and receiving updates. A delay of zero steps would mean that if the mobile device requests an update, it would be instantly available to use. Generally, this is the best case scenario but in reality unachievable due to communication overhead and inference time of the remote NN. Here, the NN functioning as the part on the constrained device is eight times faster which equals to a delay of minimum eight steps. This holds if we want to compute as many outputs

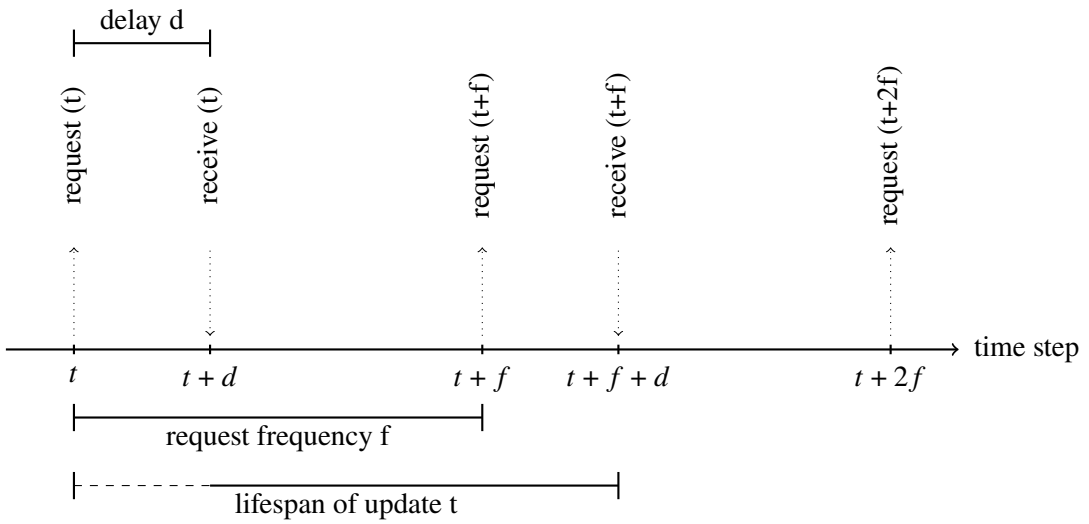


Figure 6.1: Offloading and Merging Workflow

as possible per second, but still neglects the communication cost. If we require lower frame rates, the delay can be lowered by slowing down the computation on the mobile device, because fewer steps per second have to be computed. In this experiment, we take a look at the case of achieving a high frame rate. Thus, it considers the situation that inputs are handled immediately after one another.

In this chapter, the setup and technique for merging is described and different methods to combine outputs are derived by analyzing the behavior of outputs for three input angle areas. After that, a quick take on the decision operator discusses the effects of connection availability on the requesting of updates. The final sections present the results of evaluating the distributed architecture with a delayed arrival of updates and the derived merging methods.

6.1 Merging Operator

In the future, the merging operator shall run on the mobile device and then listens for incoming updates. It is responsible for combining local with remote results in a way that improves the accuracy of the local outputs for multiple steps after arriving at the mobile device. This is done by not only adjusting the update to fit to the current step, but also continuing it with the help of local changes. This section is about the general workflow of the merge process and its different components. Also, three different merging methods are derived and described for the use in the evaluation later in this chapter. To start things off, consider the application running in local-only mode with no updates currently available on the device. In Figure 6.1, the workflow of requesting updates is shown for a setting that assumes that we are currently at time step t of the computation and have an update frequency of f. At this time step t, the decision is made that we want to request an update from the remote NN. The mobile device sends a request, and we temporarily save the local result of time step t, in the following called l_t. We then receive the update for step t, called u_t,

with a delay d at time step $t + d$. This update corresponds to step t and therefore has to be adjusted to be applicable in the current step.

$$\text{change}_x = l_{t+x} - l_t \tag{6.1}$$

The idea is to use the change of the local network over the past d steps which is computed with Equation 6.1 by substituting x with d. The next step is to continue the update from step t by using the change we just calculated.

$$\text{continued}_{t+x} = u_t + \text{change}_x. \tag{6.2}$$

The continued update gets computed by adding the change to the update as shown in Equation 6.2. Now, both the continued update and the local result of step $t + d$ can be merged.

$$\text{merged}_{t+x} = \alpha \cdot \text{continued}_{t+x} + (1 - \alpha) \cdot l_{t+x} \tag{6.3}$$

The merging formula is shown in Equation 6.3. It uses the parameter α as a mean of determining to which extent the continued update should contribute to the merged result. The parameter α can take values between zero and one. Setting α to zero means that only the local result is considered, while a value of one corresponds to taking only the continued update as the final output. More on determining the value of alpha is presented later in this section. For now, we get back to the merging workflow as shown in Figure 6.1. Step $t + d$ marks the point in lifespan of the remote update u_t, where it is possible to consider it for merging with the local output. This is the interval in which the update is part of the merging operation. In the case of a fixed request frequency f and a fixed delay d, this interval is exactly f steps long and reaches from receiving one update to receiving the next update. For the next f steps, the merging operator calculates the local change for each of the following time steps back to step t. This means that the next merged result in the sequence can be achieved by replacing the index x with $d + 1$ in Equations 6.1 to 6.3. As already mentioned, this continues for f steps until step $t + f + d - 1$ is reached. Before that, at step $t + f$, the decision operator requests a new update which will be received at step $t + f + d$. At this point it will start replacing the current update u_t and begins the active phase in its own lifespan. The cycle then continues for the following f steps until the next update arrives at the local device.

Algorithm 6.1 Merging algorithm

Input: local result at request: l_t, current local result: l_{t+x}, remote update: u_t
Output: merged output: merged_{t+x}

```
procedure MERGING(l_t, l_{t+x}, u_t)
    localChange ← l_{t+x} − l_t            // local difference between request step and current step
    continuedUpdate ← u_t + localChange    // adjusted remote update to fit for the current step
    α ← COMPUTEALPHA( )
    mergedOutput ← α · continuedUpdate + (1 − α) · l_{t+x}        // merging with parameter α
    return mergedOutput
end procedure

function COMPUTEALPHA( )
    return α                                // α is calculated based on the merging method
end function
```

The general procedure is the same for all following merging methods and can be seen in Algorithm 6.1. It shows the procedure for of merging local and remote output for one step, and utilizes the aforementioned equations. These procedures are called for each step in the computation. The additional calculations per step consist of a fixed number of operations to compute the local change, here this is five subtractions due to the network having five output values, see Equation 6.1. To get the continued update, calculated in Equation 6.2, the local change is added to the remote update resulting in a constant five additions per step. Calculating the weight parameter α is left out for the runtime analysis as it depends on the merging method and can range from a simple table lookup to a more complex calculation, depending on other parameters, such as delay or total age of the continued update. The additional runtime of adjusting the update was evaluated and showed an average time consumption of 10 μs. Compared to the AIT of 12.5 μs by the local model this means that we close to double the cost for each step by adjusting the received update. After computing the continued update, both outputs are merged together to form the final output. This consists of ten multiplications, one subtraction and five additions. Merging the outputs was also evaluated on additional runtime for each step. Results showed that it took 14 μs on average. In total, we have six subtractions, ten additions and ten multiplications for one step. This stays constant for the whole data because the output size does not change. With the resulting extra runtime per step, the average processing time increases from 12.5 μs to approximately 36.5 μs.

For a real distributed setting containing NNs on different devices, additional communication costs emerge when requesting updates from the server and receiving them on the mobile device. Since the evaluation takes place on a single endpoint in this thesis, thus no external communication is necessary and not further investigated to account for additional overall cost. Comparing this value to the runtime of the LSTM NN shows that this approach is three times faster than running the LSTM NN on the underlying device for evaluation. When evaluating in a real distributed setting, the execution on the mobile device is expected to have an increasing AIT for the base computation, but also an increased additional time consumption to merge the outputs. With a changing approach to calculate the weight parameter α, the runtime improvement can decrease. This has to be noticed as the weight value decides how much the update contributes to the merging process. In the following, we take a look at three different merging methods and their approach to determine the parameter α.

6.1.1 Mean Merging

The first method implements the combination of local and remote results as *mean merging*. This is a naive way of combining the outputs. It takes both the local output and the continued output and adds them together before dividing by two, hence the name mean merge. The parameter α is set to a constant value of 0.5. This method does not account for any quality of the continued update and marks the starting point for the next approaches. Hence, this and the following approaches should consider a parameter that represents the maximal lifespan before the continued update gets worse than the local output. This behavior is further investigated in Section 6.1.3. There, a way to derive a possible dynamic setting to determine the lifetime is proposed. For a well-developed merging operator the goal should be to derive a parameter that represents the maximal lifespan before the continued update gets worse than a local output. The next approach tries to further exploit the

increased accuracy of continued updates by assigning a smaller weight to the local output when merging. This can increase the overall accuracy but still has the same problem as the *mean merging* approach by not being able to detect a decreasing quality.

6.1.2 Decoupled Execution

If we think again about the proposed method of a highly accurate NN that provides the local execution with outputs, we can question why we do not use the update with a higher value of α. This second method implements this thought by setting the α to a value of 1.0 in Equation 6.3. This way, the factor of utilizing the local result for the merged output turns zero. The continued update immediately turns into the final output of the current step. But as for all merge methods, we still need the local result to compute the local change. The outcome of this approach represents a *decoupled execution*. In general, we would expect the remote update to be of higher accuracy than the local computations. Thus, taking the update with a higher percentage is desired.

The problem lies in continuing the remote output with the help of local changes. By continuing the update, it is possible for the accuracy to decrease over time. Following that, if we still consider the value of parameter α to be one, the final outputs for the *decoupled execution* can be worse than the local outputs, while still being considered for combination. This observation is thoroughly described in Section 6.1.3. Based on the results there, the *decoupled execution* would also produce worse results than the *mean merging*. In earlier stages, the method of only taking the continued update yields better results, but after the intersection between local and continued results the averaging of results would decrease the negative effects. The next merging method tackles this problem of considering inaccurate values by analyzing the properties of continued updates. It proposes a possible solution to detect and avoid the situation of considering less accurate continued updates for the final output.

6.1.3 Quality-Sensitive Merging

The third and most refined approach uses a quality-sensitive merging method with a dynamic value for α, according to the current delay and properties of the current inputs to the NN. Figure 6.2 shows the behavior of the local NN for one input weight in the data set. The steps on the horizontal axis are sorted by the angle of the input. Each blue dot represents the MAE of a single input to the local NN. We can see that the network performs well on inputs zero to 750 and 1500 to 2400 with a MAE of below 2% and below 5% for the first and final section of the inputs, respectively. These are areas with inputs having small and large input angles. For the medium angles, the NN struggles to produce accurate outputs with the MAE fluctuating from below 5% up to 15%. This way, we can divide the inputs into three differently performing sections depending on the angle of the current step. In this case, the first 25%, the middle 50% and the final 25% of inputs for one input weight use different methods to determine the parameter α. It is necessary to again clarify that the value range of the outputs in the validation data is from zero to one. Having MAE values of 2 to 5% shows that this is already in the lower bound for possible MAE values.

To derive a formula for α, we take a look at the accuracy of continued updates, depicted in Figure 6.3. It consists of small sections inside the three partitions, corresponding to small, medium and large input angles. Each figure shows the steps after the arrival of an update on the horizontal

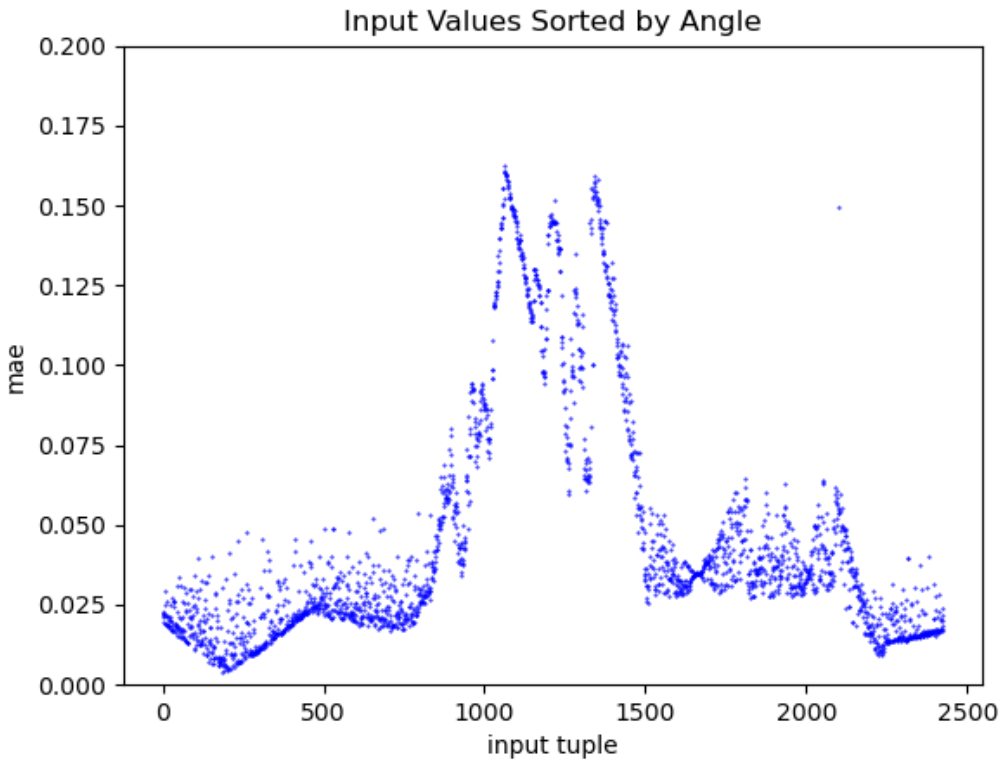
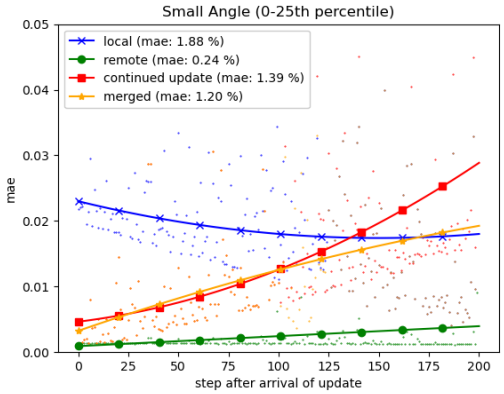


Figure 6.2: Evaluation of Inputs on the Local Network

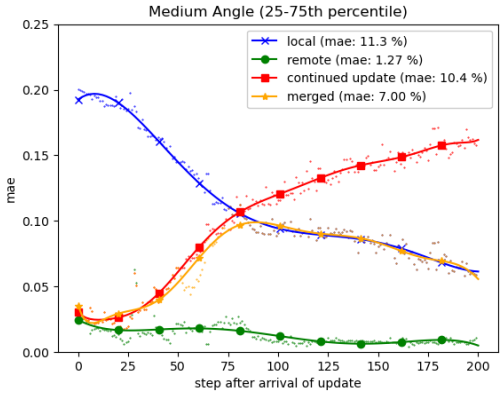
axis. The update is therefore received at step zero. The vertical axis represents the MAE of the different computations. Shown in this figure are the performance of the local NN (blue), remote NN (green), continued update (red) using the decoupled merge method as previously described and the merged output (orange). The method for calculating the merged output is derived in the following by looking at the behavior of the continued update.

When inspecting Figure 6.3a, we can see that the continued output is accurate in the beginning steps but then drops off over time, intersecting with the local performance at step 125. Taking a look at the properties of the red line reveals a non-linear decrease in accuracy. Therefore, when merging in the first 100 steps, the continued update is considered with a weight of one. At 100 steps after the request, the value of α linearly decreases and α turns zero at 125 steps after receiving the update. This means that if no update is received after 125 steps, the merging operator forces the application to switch to a local-only execution.

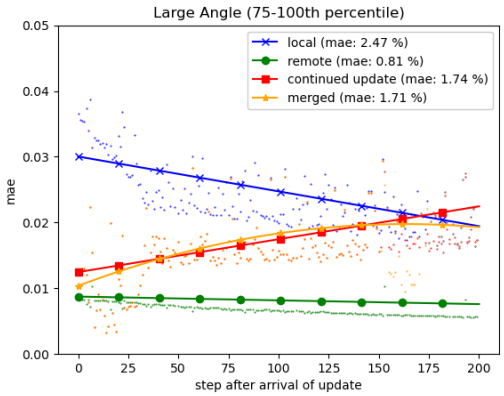
For small angles, the function in Listing 6.1 calculates α with the previously described properties. For angles in the lower 25 percentiles, the lifetime is set to 100. This marks the start of a decreasing weight being applied to the continued update. The orange line in this figure shows the MAE of the continued update merged with the local result using this exact formula to determine the value of α for each step. This lowers the MAE in this snippet from 0.0139 to 0.0120. The main difference can



(a) Small Angle Area



(b) Medium Angle Area



(c) Large Angle Area

Figure 6.3: Characteristic of Continued Updates

be seen in steps greater than 125 where the usage of the continued output is worse than the local execution. There, the switching to local-only execution of the merge method shows its impact by disregarding the continued update in these steps.

A similar behavior can be seen in Figure 6.3b. The only difference is that in the medium angle area the quality of the continued updates decreases faster. For this specific section of inputs, the intersection between local and continued update occurs at step 75. Based on that, the function in Listing 6.1 calculates the value of α in a way that 50 steps after request, the continued update linearly decreases its contribution to the merging process and turns zero at 75 steps after request. This is done by setting the lifetime to 50 steps for this part of the input. The orange line in Figure 6.3b shows the MAE of outputs merged using the proposed function to calculate α. The MAE of merged outputs over the 200 steps explored here is 0.070, while only using the continued output results in a MAE of 0.104. This is due to the decrease in accuracy of the continued output after step 75 where the merge method takes the local outputs and ignores the continued update.

The final 25% of data for each weight uses a different approach to calculate α. As we can see in Figure 6.3c, the continued update follows a linear decrease in accuracy before intersecting with the local results at step 175. For this area, we again use the function in Listing 6.1 as the

Listing 6.1 Weight Computation for Small, Medium and Large Angles

```
def calculate_alpha(request_step, current_step, angle):

  if angle <= 0.25:  # small angle
    lifetime = 100
  if 0.25 < angle <= 0.75:  # medium angle
    lifetime = 50
  else:  # large angle
    lifetime = 150

  if current_step - request_step < lifetime:
    alpha = 1.0
  else:
    alpha = 1 - (1 / 25 * (current_step - request_step - lifetime))

  return max(alpha, 0)
```

formula to calculate α. This time, for the first 150 steps, only the continued update is considered. Parameter α then decreases linearly over the next 25 steps. This way, we get an α of zero for merging computations 175 or more steps after requesting the update. For the special case of the 200 steps shown in this figure, this method decreased the MAE from 0.0247 down to 0.0171% when comparing to the local results. It also preserved a similar MAE compared to the continued update method. This again shows the desired behavior as it utilizes high accuracy outputs in earlier stages and then switches to local outputs for areas where the continued update would mean a decrease in accuracy.

These equations can be further refined and by no mean fit these sections perfectly but rather serve as a starting point and show an early approach to determine the merge parameter α. They are approximations to the behavior across multiple input values which can be improved by dividing the input set into more than three parts. Each part then has an own equation to calculate the value of α. The derived equation can be tweaked to better fit the properties of the continued output in a smaller section of input values. That way, the merging operator can be optimized for better detection of when the continued output is no longer worth considering for the merged result. This method can be utilized in cases where either the connection speed is slow or only long request frequencies can be achieved due to the lack of a stable connection. In those cases, the ability of the merging operator to decide when to switch to local execution should have an advantage over ordinary methods. The next section is about the decision operator, which is responsible for deciding when to offload.

6.2 Decision Operator

While the merging operator adjusts the received update to fit to the current step and then combines local result with the update, the decision operator answers the question of how often an update is requested by the mobile device. Again, this correlates to the quality of the continued update compared to the local performance. This is thoroughly characterized in Section 6.1 where we looked at different areas of input data and how well the continued output performed for 200 steps

after receiving the update (Figure 6.3). There, we saw that for the small NN, functioning as the network for later evaluation, three areas with different properties can be defined. Following that, we decided that for these specific cases a threshold can be set to indicate that the continued update no longer outperforms the local results. Hence, the offload decision can be dynamically derived in the same way that the merging parameter for the combination of results can be determined.

Crucial for the offload decision is again the intersection between the accuracy of the local execution and the continued update. A close investigation on the behavior of continued updates depends on the way of adjusting remote updates on the local machine. For the derived methods in the previous section, the performance compared to the local results was taken into account to derive a merging technique. This way, a method was proposed that exploits accurate updates in early stages after arrival and discards low-quality continued updates for steps after intersection of both performances. In Figure 6.3 the behavior for three different angle input areas and the improvements with the *quality-sensitive merging* approach is shown. For small angles, an update every 125 steps would be sufficient to achieve an improvement, that is assuming a delay of 0 steps. In medium and large angle areas, an update every 75 and 175 steps respectively is necessary to ensure an increase in quality of the merging technique. Since a delay of 0 steps does not exist in a real application, we also have to think about the current delay of the updates. A late request to the server can result in the update not arriving in time to stop the decrease in accuracy of the previous update on the local machine. This means that the request frequency of updates should also change depending on the properties of the current input data and the current delay. For areas where the NN struggles to produce accurate results or where the accuracy fluctuates, a higher update frequency stops the effect of deteriorating continued updates. When the input values correspond to an area where the NN is accurate, a received update can be used for a longer time, before getting worse than the local outputs.

Based on these observations, a function to determine whether an update should be requested is derived. This function takes angle area and velocity, current connection delay and the steps after requesting the latest update into account. It can be seen in Listing 6.2. The current angle area is responsible to decide which base function to choose. Angles are separated in the same way as for the third merging approach. Therefore, the *angle* input to the function is between zero and one to determine in what section of area the local computation currently is. This is decided in the first part of the offloading function. For example, when looking at small angles in Figure 6.3a the continued update intersects the local performance at step 125 after arrival of the update. The same way that we determined for the *quality-sensitive merging* in Section 6.1.3 we decide that the request of an update should already happen at step 100. This results in a base function to model the decision of requesting an update which reaches the value 1.0 at step 100. Depending on the input angle the maximum active lifetime of an update is specified. For small angles the decision operator sets the lifetime to 125. With different angle areas come different functions to model the quality decay and lifetime of updates. In the medium angle area a value of 75 is chosen as the lifetime and the base function turns 1.0 at 50 steps after requesting the latest update. Based on Figure 6.3c a lifetime of 175 is determined, and the applied function turns 1.0 at step 150. All three cases use a quadratic function to determine the base value to decide when to request an update.

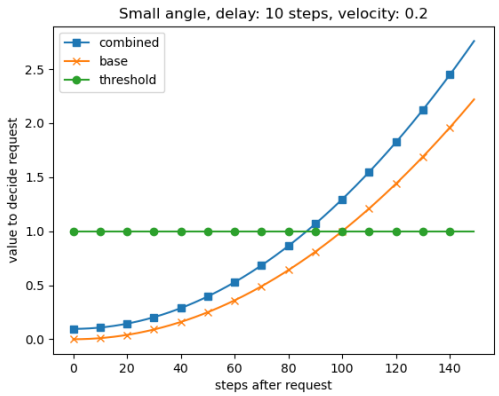
The current delay of updates also plays an important role in requesting an update. This factor is calculated in the second part of the function in Listing 6.2. Big delays require updates to be requested in time to prevent the unintended behavior of considering continued updates, although they are not beneficial to the merging anymore. In contrary, small delays do not require immediate

Listing 6.2 Function to Determine the Offloading Decision

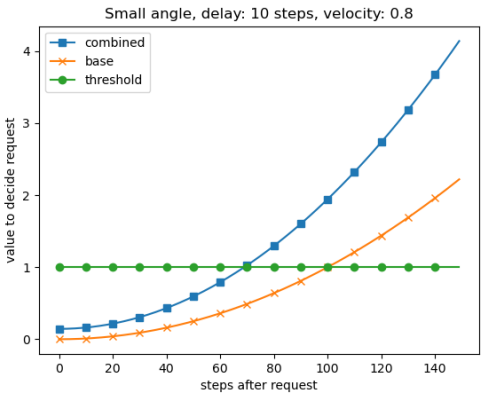
```
def determine_request(angle, angle_velocity, steps_to_last_request, delay, cooldown):

  # early stopping due to existing cooldown
  if cooldown:
    break
  # determine base value to request an update
  if angle <= 0.25:  # small angle
    max_lifetime = 125
    value_last_request = (1/10000) * pow(steps_to_last_request, 2)
  if 0.25 < angle <= 0.75:  # medium angle
    max_lifetime = 75
    value_last_request = (1/2500) * pow(steps_to_last_request, 2)
  else:  # large angle
    max_lifetime = 175
    value_last_request = (1/22500) * pow(steps_to_last_request, 2)

  # determine additional value based on the current delay
  value_delay = delay / max_lifetime

  # determine additional value based on the angle velocity
  value_velocity = 1 + abs(angle_velocity)

  # decide whether to request an update
  if (value_last_request + value_delay) * value_velocity >= 1.0:
    request_update()
```

requests and can be used to lessen the communication need. With additional knowledge about the lifetime of an update in the current angle area a way to model the need for requests can be defined. The method proposed here divides the current delay by the maximal lifetime. This way, for long delays an earlier step to request updates can be achieved, while short delays do not significantly affect the offloading decision. The resulting value gets added to the base value. In a more advanced setting, the delay changes depending on the available connection quality and availability. This means that the decision operator can try to predict the current delay based on the delay for previous updates.

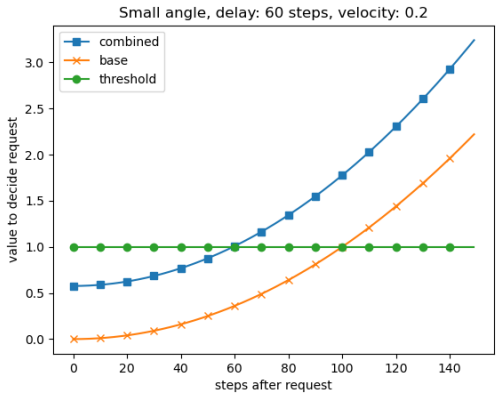
As a fourth factor, the angle velocity is considered. Due to the quick change in input values when having a high angle velocity, updates have to be more frequent to account for possible rapid changes in the continued update and therefore its accuracy. When having a low velocity, new updates are not immediately requested since continuing remote updates yields better results due to the more stable nature of local changes. Because angle values in the input data are normalized to be between -1 and 1, the absolute value of the velocity is considered. This also means that both directions of the movement are treated equally. Here, the angle velocity functions as a factor. For low velocities, the factor is close to 1.0 while for high velocities it can reach values close to 2.0 and therefore doubles the value used to decide a possible request.



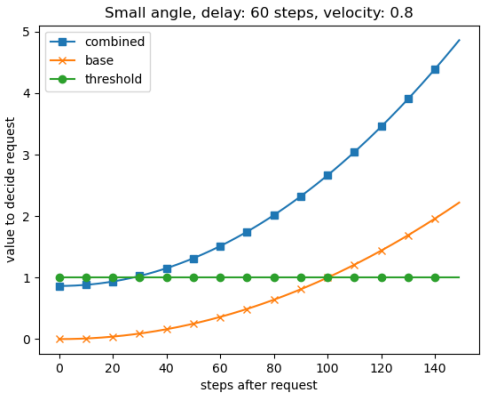
(a) Low Delay and Low Velocity



(b) Low Delay and High Velocity



(c) Medium Delay and Low Velocity



(d) Medium Delay and High Velocity

Figure 6.4: Characteristic of Continued Updates

The last part decides whether the decision operator should request an update or not. This final decision is made by treating the resulting value as a probability and if it is equal or greater than 1.0 an update is requested. Since the final value can be greater than 1.0, an update request could be triggered for each step in the execution. This would result in an abundance of updates and could overload the communication channel to the server with further negative effects. To counteract this problem a request cooldown of multiple steps should be added to the decision operator. This would relieve pressure on the total communication and lessen the effects of additional costs coupled with handling received updates.

An example calculation for a fixed delay and velocity setting is given in Figure **??**. Each plot contains the base value for deciding whether to request an update in orange. The green constant line shows the threshold at which the request gets triggered. In blue, we can see the combined method considering base, delay and velocity values as described previously. Figures 6.4a and 6.4b show the values for a delay of 10 steps and a velocity of 0.2 and 0.8, respectively. For this setting, the figures show that a higher velocity steepens the curve and forces the decision operator to request updates faster than for a low velocity. For a low delay the request step gets shifted to the left by 15 to 25 steps in these cases. The Figures 6.4c and **??** represent the change of values for a medium delay setting of 60 steps. Here, the delay value plays a bigger role as it increases the values for

the base functions by $\frac{60}{125}$. For the setting with a low velocity, we can again see that this does only slightly affect the steepness of the curve. In contrast, the high velocity setting shows the situation in which an update would be requested at step 30 already. This is 70 steps earlier than the base function suggests without considering delay and velocity.

Since the primary focus of this thesis is the merging operator, the exact performance of this dynamic decision operator is not used for evaluation. It serves as a starting point to derive a dynamic request frequency dependent on different factors during the execution including delay, input values and step of the last update. In an advanced approach to evaluate a distributed network architecture the method derived here can be used as a starting point to allow for a dynamic Here, the request frequency, used for the evaluation, follows a static scheme. For an update rate of k, an update is requested every k steps, regardless of the current input properties. This way, the focus can be put on the performance of previously derived merging methods. The exact values are used for the static approach to model the offloading decision are described in the following section on evaluating the three proposed merging methods.

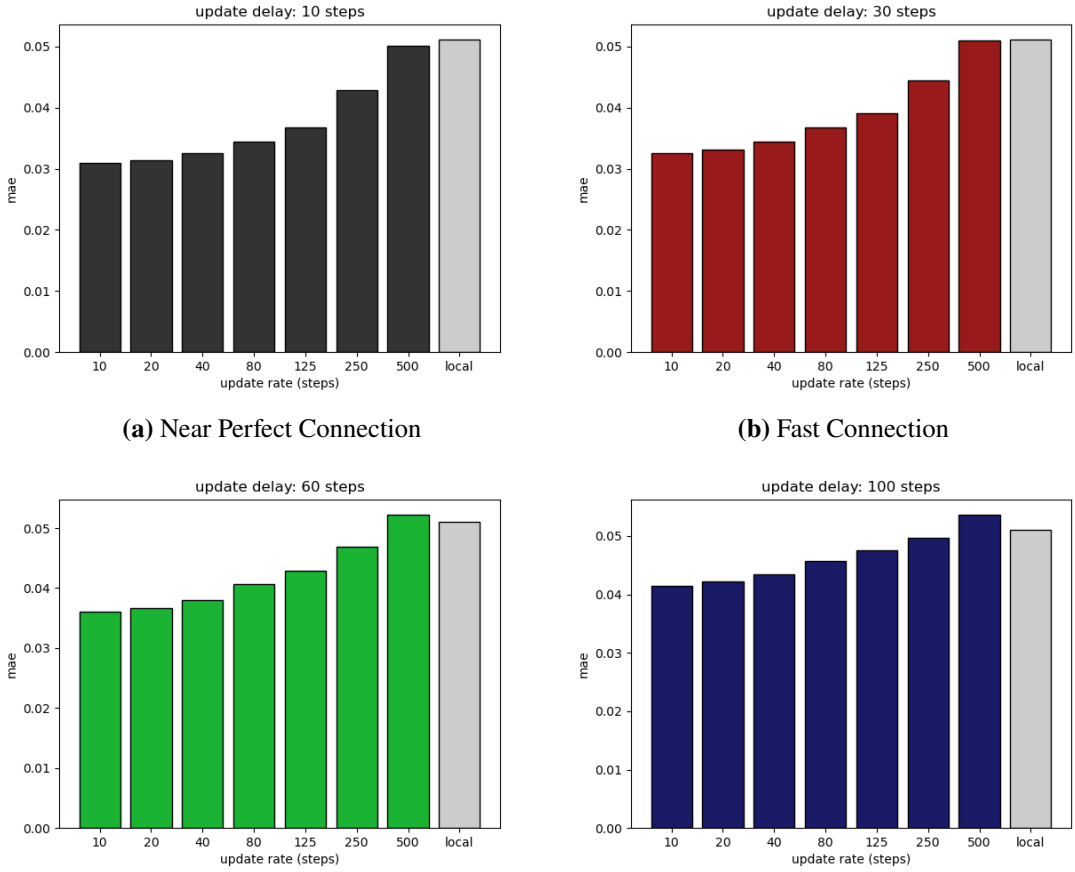
6.3 Evaluation

This section covers the evaluation of the previously derived merging and offloading settings. It considers different connection quality parameters and computes the MAE across the whole validation data as described in Section 2.5. For the update frequency, static values of 10, 20, 40, 80, 125, 250 and 500 are chosen to cover possible offload rates by the decision operator. The test environment uses a single frequency for the whole validation data and is not dynamically calculated, see Section 6.2 for more details on the decision process. A delay of ten steps is used across all update rates to show the results of a near-perfect communication. Accounting for a fast and medium transmission quality is a delay of 30 and 60 steps, respectively. An arrival of the update 100 steps after request investigates the possible improvements, if the connection quality is slow. This setting is also used to determine if late updates can still improve the performance of the local NN. These delays are arbitrary values and can differ in real life application. In this case, they are solely used to represent different connection qualities for the proposed update rates.

6.3.1 Mean Merging

The first merging method to be tested on different parameter settings for delay and update rate is *mean merging*, described in Section 6.1.1. This is a naive way of combining a continued update with the local result as it is using the same weight for both outputs without any assumptions about the quality of the both. Figure **??** shows the increase in accuracy for the averaged output between local and remote computation. Each subplot shows a different delay setting, precisely 10, 30, 60 and 100 steps, which are used for different connection qualities. The bars represent the MAE with respect to each unique update rate. The MAE is calculated for the whole validation data and shows the average error across the 10000 steps of validation. For comparison, the performance of the local network is represented by the gray bar and shows a MAE of 0.051.

Evaluating different connection qualities on the mentioned delay settings resulted in the following behavior. First, we have to mention that although the average MAE is shown to be below the local value in most cases, for combinations of update rate and delay settings that add to a value higher



(a) Near Perfect Connection

(b) Fast Connection

(c) Medium Connection

(d) Slow Connection

Figure 6.5: Results of Mean Merging

than the intersection step in Figure 6.3 result in an unwanted behavior. This is due to the decreasing accuracy of the continued update if it gets considered for 75, 125 or 175 steps depending on the input area. The characteristic seen there showed that for the first steps after receiving an update, the accuracy of continued updates is far better than the local outputs. Contrary to that, in later stages, the accuracy of continued outputs decreases below local outputs. Thus, it can happen that the overall MAE is still better, although for some steps a negative behavior can be noted. Therefore, even before the bars represent a higher MAE to the local execution an undesired behavior occurs for the aforementioned settings, where a single update not only contains parts with an improved accuracy, but also results in parts with a worse accuracy than the local outputs. For the *mean merging*, a near-perfect connection results in the biggest improvement over the local computation, shown in Figure **??**. This decreased the MAE to 0.031 when receiving an update every 10 and 20 steps. This is 0.02 less than only using local results and is an improvement of 39%. The effects of receiving accurate outputs decrease when updates arrive every 80 steps or more. A fast connection still achieves a MAE of 0.033 at the highest update frequency. Only sparsely receiving updates does not substantially improve the overall accuracy, which can be seen for an update rate of 500 steps. For a medium connection, the best improvement shows a MAE of 0.036. Here, the phenomena of the fast connection solidifies, and less frequent updates decrease the improvements. In case of

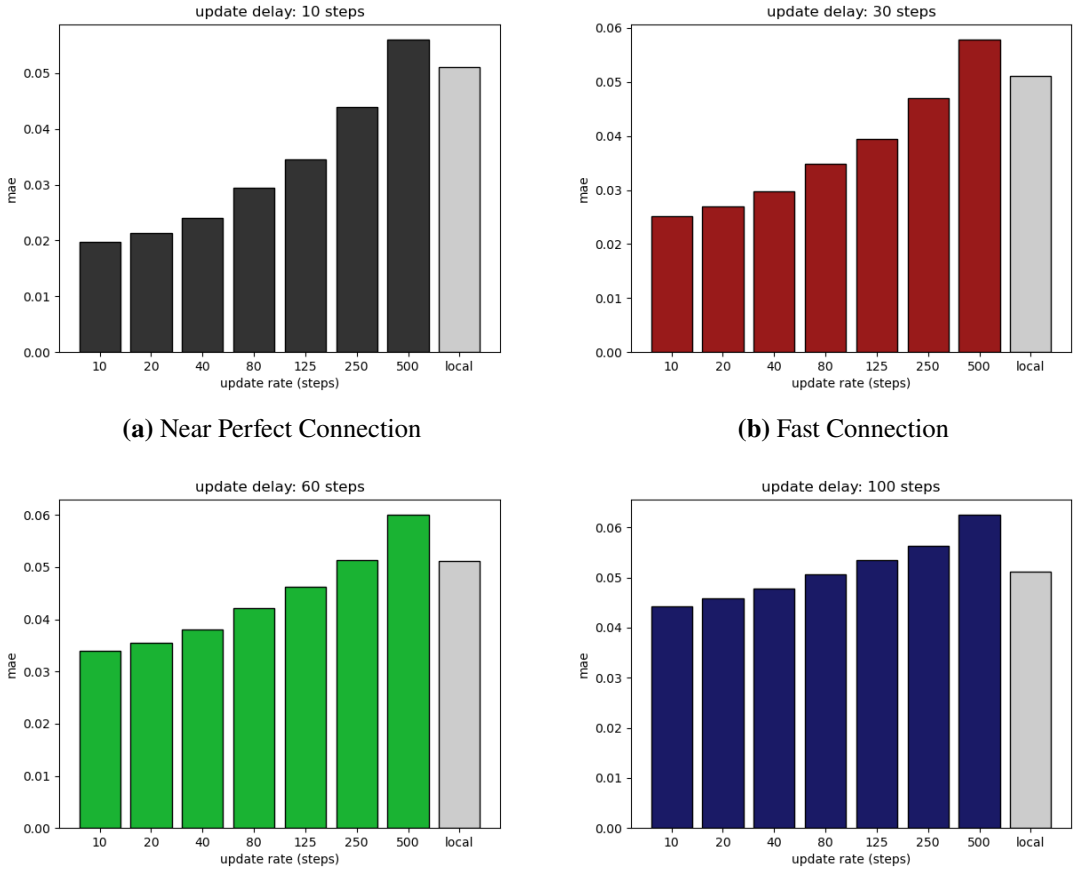
an update rate of 500, the merged results are averagely worse than the local execution. While this behavior first shows at an update rate of 500, even for smaller values it is possible for multiple steps to use less accurate continued updates as previously mentioned. A slow connection improved the MAE in the best case to a value of 0.041 which is still an improvement of 20% over the local execution. Sparse updates increase the MAE as already seen across other connection qualities.

The general trend is that more frequent updates can still outperform the local network, even if the update arrives with a high delay. A decreasing connection quality with longer delays reduces the positive effects of frequent updates. For sparse updates, the accuracy cannot be improved significantly. This is due to the continued update being adjusted with the local changes over many steps, therefore slowly decreasing its accuracy. Following that, the continued update can get worse than the local execution, while still being considered for the merging of the outputs. The next section covers the *decoupled execution* which assigns a bigger weight to the continued update, but still does not account for a decreasing accuracy the longer an update is used.

6.3.2 Decoupled Execution

The *decoupled execution* implements the usage of continued updates, whenever they are available as shown in Section 6.1.2. This means that the local outputs are disregarded completely for the merged results and only serve to compute the local change to adjust the remote update. Results of the *decoupled execution* are shown in Figure **??**. The seven bars of the same color represent the overall MAE of different rates to request updates, ranging from 10 to 500 steps. The local execution without any updates is shown as the light gray bar on the right-hand side. Figure **??** displays the measurements for a delay of 10 steps. Having frequent requests yields the best result for this technique. It lowers the MAE to 0.020 and is an improvement of 61% over the local execution. With more sparse updates, the error increases. For an update rate of 500 steps, the error increases to 0.056. This is an overall higher value than the local network without offloading. This again shows the problem of only using continued updates as seen in Figure 6.3. For higher values of the update rate the continued update with decreasing quality gets considered for more steps. Hence, a cutoff point would be needed to stop the low-quality output from being used as the final result. Depending on the area, even for an update rate of 125 it is possible to have this unwanted behavior when closely looking at one update request cycle. Although the bar shows an overall improved MAE it might be necessary to change the update rate to not only improve the overall accuracy, but also decrease the probability of considering low-quality continued outputs. A fast and medium connection quality with delays of 30 and 60 steps show a similar behavior for an increasing update rate. Requesting updates every ten to 125 steps with a fast connection increased the MAE by 0.005, with the best performance showing a value of 0.025, compared to the near-perfect connection. A medium connection still improved the overall error to 0.034, although the arrival of a requested update is delayed by 60 steps. With more sparse updates, the MAE using this technique again gets worse than for a local-only execution. For a delay of 100 steps, the MAE increases to values higher than the local MAE starting at an update rate of 125. Therefore, in this case, frequent updates are necessary to achieve improvements.

When looking at better connection properties, we might think that requesting an update every 250 steps is sufficient to improve the accuracy of the local NN. This is true for the underlying validation data if considering the overall performance, but does not consider the decreasing quality of continued updates over many steps, which is covered in Section 6.1.3. There, we saw that the



(a) Near Perfect Connection

(b) Fast Connection

(c) Medium Connection

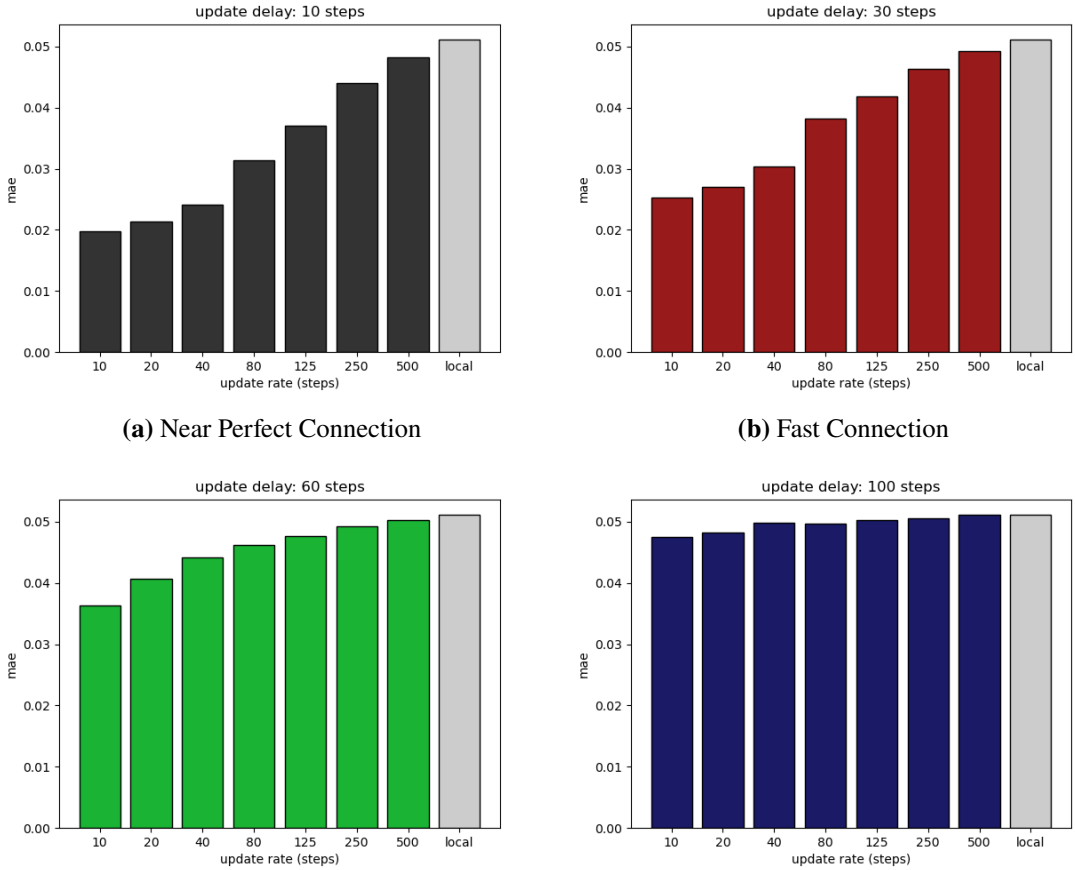
(d) Slow Connection

Figure 6.6: Results of Decoupled Execution

longer an update is continued, the less accurate it gets and eventually gets worse than the local outputs. Depending on the area, this intersection occurs well before 250 steps. Thus, the continued outputs are more accurate for one part of the lifespan of the update and less accurate for the later steps of its usage, compared to the local results. This lowers the MAE also for sparse requests, but is not the desired behavior in a real-time execution. Therefore, a cutoff point is added to the technique, after which the merging operator switches to a local-only computation. This progression is implemented and forms the *quality-sensitive merging*, evaluated in the next section.

6.3.3 Quality-Sensitive Merging

This merging method implements a first approach to minimize the probability of considering low-quality continued updates for merging. Weights of the adjusted update are based on a quality measurement considering the delay and predicted accuracy of received updates. According to three different input angle areas, varying techniques to calculate the weight are proposed to fit the behavior for each specific area. An in-depth description of *quality-sensitive merging* can be found in Section 6.1.3.



(a) Near Perfect Connection

(b) Fast Connection

(c) Medium Connection

(d) Slow Connection

Figure 6.7: Results of Quality-Sensitive Merging

Figure **??** shows the evaluation of the same delay and update rate settings as used for the *mean merging* and *decoupled execution*. The subplots again represent four connection qualities, near-perfect, fast, medium and slow, and show the overall MAE of different update rates applied as bars. The gray, most right bar stands for the MAE of the local execution without updates across the whole validation data. When looking at a near-perfect connection, we can see that updates every ten and 20 steps yields the best result with MAEs close to 0.020. This is a decrease in error of 60% compared to the local-only execution. In Section **??**, we saw that considering only the continued update, the best setting achieved an improvement of 61%. Therefore, this method performs similarly on frequent updates and a low delay. For an increasing amount of steps between updates sent to the device, accuracy drops off to 0.038 for an update rate of 125 steps. This trend continues until we can only see an overall improvement of 2% for requesting updates every 500 steps. The overall MAE, considering all update rates, shows worse results than the *decoupled execution*. This is because we have changed the greedy nature of the previous method to switch to a local execution to lower the possibility of considering less accurate continued updates for merging. Hence, for the *quality-sensitive merging*, the exploitation of accurate continued updates in earlier stages decreases. But in contrast, it results in the benefit of being able to switch to consider local outputs when expecting a decreased quality for continued outputs. As seen in Figure 6.3, the

accuracy of continued updates decreases after 75 to 175 steps after requesting an update, depending on the current input angle area. Therefore, for update rates higher than 125 a switch to the local execution is applied in the majority of update cycles.

The fast connection behaves the same as the near-perfect setting, but with values of the MAE shifted upwards. For new updates every ten steps, this method achieves a MAE of 0.025, which is an overall improvement of 51% over the local execution. Sparse updates on the other hand again decrease the benefits and make the MAE of this merging technique approach the same level as the local network. This happens because the applied method, which, depending on the input angle, switches to a local-only execution for 75 to 175 steps or more after requesting the update. When comparing the results of the good connection qualities to the *mean merging* on the same settings, a lower overall MAE for frequent update rates can be seen. This means that the method of having a weight above 50% for the continued update is beneficial, if receiving many updates. Also, for sparse offloading requests, a slower drop-off in accuracy is the outcome of using this method with a lowered weight, if the current step is already many steps after arrival. When looking at medium offloading decisions, mainly update rates of 80 to 125 steps, *mean merging* outperforms *quality-sensitive merging*. For these values, it achieves a lower overall MAE, hinting towards a too late cutoff for considering the continued output in the merging process.

When looking at Figures **??** and **??**, we can see the improvements for a medium and slow connection setting with delays of 60 and 100 steps. In earlier sections, we saw that for the two previous merging methods, longer delays, combined with fewer updates caused the MAE to get worse than the local execution. Since there is a cutoff specified for the *quality-sensitive merging*, after which we do not consider the update for merging, this behavior does not appear here. While this is good on the one hand, we can also see that for frequent updates with high delays, the application does not benefit as much as it did for better connections. This is because the implemented switch to a local execution stops an update with high delay to be used for many steps shortly after arriving at the local device. Here, the overall MAE of the medium connection is lowered to 0.036 in the best case. For a slow connection the most frequent update rate showed an improvement of 7% over the local execution, to a MAE of 0.048. As we can see in the figure covering the slow connection, no significant benefit has been achieved for any update rate. On the plus side, no update rate caused the overall MAE to rise above the local-only performance.

For the fast and near-perfect connection settings, equal overall MAE values as for the *decoupled execution* are achieved. Again, when looking at the performance over the course of the lifespan of a single update, the *decoupled execution* at some point utilized an output that is worse than local values. Due to the high accuracy in early steps after receiving an update, the negative steps can be overruled by the many good steps in the beginning, when looking at the overall MAE of the decoupled approach. The *quality-sensitive merging* in contrast tried to minimize the possibility of this behavior by defining a cutoff point. This way, for more sparse updates, the performance across a single update is increased and implements a more conservative approach to utilize the high-quality updates. For this, the characteristics shown in Figure 6.3 are applied to consider the predicted quality of continued outputs during a single update lifespan.

7 Conclusion and Outlook

In this thesis, we tackled the problem of running numerical simulations on a resource-constrained mobile device. Outputs of this simulation are supposed to be of high quality and efficiently calculated for the use in visualization in an AR setting. NNs work as a surrogate model to compute accurate outputs more efficiently. Especially LSTM networks, a type of NN, used for continuous data, showed a decrease in error of over 40% compared to standard DNNs. This comes at the cost of an AIT five times higher than a similar NN without LSTM layers. Based on this result, the networks for the use in the distributed architecture were determined. They functioned as the local and main calculation of outputs on the mobile device, and for the server, used to compute high quality updates. A sophisticated method for merging local and remote results was derived by investigating the behavior of continued updates, remote outputs adjusted to fit the current step of the computation, across different areas of input parameters. With the help of these characteristics, the method also implements a switch to local-only execution for cases, where no further updates should be considered. This stops old, and possibly inaccurate, updates to be considered in the merging process. For the decision operator, a similar way was proposed to dynamically determine whether the mobile device shall request an update for the current step. The proposed deduction considered input properties such as angle and angle velocity, delay and age of the current update. Evaluation of the presented merging methods on static offloading decisions showed that, with updates every ten steps, the overall MAE can be decreased from 0.0511 to 0.020 for a simulated delay of ten steps between requesting and receiving an update. The effects of the methods decrease with higher delay and less frequent updates. Overall, this means that the proposed methods of integrating remote outputs in the local execution can increase the accuracy of results for the underlying data, if updates can be frequently requested. For longer distances between requests or a worsening connection speed, the benefits of merging local and remote results decrease, up to the point of not positively effecting the local execution at all. A first method was proposed to combine outputs depending on delay and the change in quality of updates over time. The *quality-sensitive merging* with updates every ten steps saved 63.5% of time elapsed per step compared to the LSTM NN, and improved the overall MAE by 60% compared to the local NN without updates. The combined method with updates planned for the execution on the local device averagely took 36.5μs, while the LSTM NN had an AIT of 100μs per step. The consequence is an overall two times higher MAE compared to the MAE of 0.0107 by the LSTM NN. Due to the already low overall MAE of the LSTM NN the deterioration to a value of 0.020 still represents the lower error bound. Therefore, the MAE of the combined approach is closer to the LSTM NN than it is to the DNN without the option to request updates.

Other simulations likely use different data to compute outputs. Therefore, the concrete functions, derived for the use in the merging operator, do not necessarily apply to other simulations. Depending on different NNs, used as local and remote NNs, the characteristics may differ from the behavior shown in this thesis. Due to this change, different functions are inevitable to calculate the weights for the local and remote outputs before merging. Also, the number of input areas, which use the same

underlying weight function, can be increased to refine the merging functions specifically for a small input area. Not only does this change the way merging works, but also the decision operator can, based on a similar investigation, dynamically decide, if an update is beneficial to the local execution. While the proposed final parameters inside the merging and decision operator do not work for all simulations, the procedure of deriving a possible solution can be applied on other underlying data. Further investigation, considering the use of LSTM networks for continuous simulations, is possible to optimize NN models on the mobile device and the server. As already mentioned, future research can improve merging and offloading operators by studying the behavior for smaller input areas. Moreover, realizing a generalized approach to determine parameters for weight assignment and request frequency for various continuous simulations can automate the currently manual procedure. Future research can continue this work by implementing the mentioned improvements or by using a real distributed setting, involving a suitable mobile device and server. With that, the impact of communication costs and delays in a real setting can be investigated.

Under consideration of the results by this thesis, in the future, mobile devices are able to efficiently compute simulations for many applications, which require accurate and real-time outputs. With the help of an optimized server communication, the combined decision and merging methods might be able to simulate the execution of the complex remote network on the mobile device, solely with the help of updates by a more accurate NN. This way, high-quality outputs are possible across all inputs, even though the local execution utilizes a less complex NN. Following that, the visualization to a person in an AR setting on a mobile device can fulfill quality criteria to allow for an interactive user experience.

Bibliography

[AAB+16] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems". In: (Mar. 14, 2016). arXiv: 1603.04467 [cs.DC] (cit. on p. 20).

[AGH18] K. Akherfi, M. Gerndt, H. Harroud. "Mobile cloud computing for computation offloading: Issues and challenges". In: *Applied Computing and Informatics* 14.1 (Jan. 2018), pp. 1–16. DOI: 10.1016/j.aci.2016.11.002 (cit. on p. 17).

[BZX+19] T. Bao, S. A. R. Zaidi, S. Xie, P. Yang, Z. Zhang. "A CNN-LSTM Hybrid Framework for Wrist Kinematics Estimation Using Surface Electromyography". In: (Nov. 28, 2019). DOI: 10.1109/TIM.2020.3036654. arXiv: 1912.00799 [eess.SP] (cit. on p. 25).

[CESZ19] F. Carrara, P. Elias, J. Sedmidubsky, P. Zezula. "LSTM-based real-time action detection and prediction in human motion streams". In: *Multimedia Tools and Applications* 78.19 (June 2019), pp. 27309–27331. DOI: 10.1007/s11042-019-07827-3 (cit. on p. 25).

[CLS+14] G. Cecchini, G. Lozito, M. Schmid, S. Conforto, F. Fulginei, D. Bibbo. "Neural Networks for Muscle Forces Prediction in Cycling". In: *Algorithms* 7.4 (Nov. 2014), pp. 621–634. DOI: 10.3390/a7040621 (cit. on p. 25).

[CVV+16] E. D. Coninck, T. Verbelen, B. Vankeirsbilck, S. Bohez, P. Simoens, P. Demeester, B. Dhoedt. "Distributed Neural Networks for Internet of Things: The Big-Little Approach". In: *Internet of Things. IoT Infrastructures*. Springer International Publishing, 2016, pp. 484–492. DOI: 10.1007/978-3-319-47075-7_52 (cit. on pp. 16, 24).

[Dao18] T. T. Dao. "From deep learning to transfer learning for the prediction of skeletal muscle forces". In: *Medical & Biological Engineering & Computing* 57.5 (Dec. 2018), pp. 1049–1058. DOI: 10.1007/s11517-018-1940-y (cit. on pp. 15, 25).

[DCM+12] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, A. Y. Ng. "Large Scale Distributed Deep Networks". In: *NIPS*. 2012 (cit. on p. 24).

[DDR15] C. Dibak, F. Durr, K. Rothermel. "Numerical Analysis of Complex Physical Systems on Networked Mobile Devices". In: *2015 IEEE 12th International Conference on Mobile Ad Hoc and Sensor Systems*. IEEE, Oct. 2015. DOI: 10.1109/mass.2015.12 (cit. on p. 24).

[DHS+18] C. Dibak, B. Haasdonk, A. Schmidt, F. Dürr, K. Rothermel. "Enabling Interactive Mobile Simulations Through Distributed Reduced Models". In: (Feb. 14, 2018). DOI: 10.1016/j.pmcj.2018.02.002. arXiv: 1802.05206 [cs.DC] (cit. on pp. 16, 23).

[Dil21] C. Dilmegani. *Dark side of neural networks explained*. Jan. 1, 2021. URL: https://research.aimultiple.com/how-neural-networks-work/ (cit. on p. 18).

[HBZ18] L. Huang, S. Bi, Y.-J. A. Zhang. "Deep Reinforcement Learning for Online Computation Offloading in Wireless Powered Mobile-Edge Computing Networks". In: (Aug. 6, 2018). DOI: 10.1109/TMC.2019.2928811. arXiv: 1808.01977 [cs.NI] (cit. on p. 24).

[HS90] L. Hansen, P. Salamon. "Neural network ensembles". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 12.10 (1990), pp. 993–1001. DOI: 10.1109/34.58871 (cit. on p. 25).

[KP16] J. Keuper, F.-J. Pfreundt. "Distributed Training of Deep Neural Networks: Theoretical and Practical Limits of Parallel Scalability". In: (Sept. 22, 2016). arXiv: 1609.06870 [cs.CV] (cit. on p. 24).

[KSH17] A. Krizhevsky, I. Sutskever, G. E. Hinton. "ImageNet classification with deep convolutional neural networks". In: *Communications of the ACM* 60.6 (May 2017), pp. 84–90. DOI: 10.1145/3065386 (cit. on p. 19).

[LS89] W. P. Lincoln, J. Skrzypek. "Synergy of Clustering Multiple Back Propagation Networks". In: *Advances in Neural Information Processing Systems 2, [NIPS Conference, Denver, Colorado, USA, November 27-30, 1989]*. Ed. by D. S. Touretzky. Morgan Kaufmann, 1989, pp. 650–657. URL: http://papers.nips.cc/paper/228-synergy-of-clustering-multiple-back-propagation-networks (cit. on p. 25).

[MCN+17] J. Mao, X. Chen, K. W. Nixon, C. Krieger, Y. Chen. "MoDNN: Local distributed mobile computing system for Deep Neural Network". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, Mar. 2017. DOI: 10.23919/date.2017.7927211 (cit. on p. 23).

[MG18] A. T. Mohan, D. V. Gaitonde. "A Deep Learning based Approach to Reduced Order Modeling for Turbulent Flow Control using LSTM Neural Networks". In: (Apr. 24, 2018). arXiv: 1804.09269 [physics.comp-ph] (cit. on p. 25).

[MM16] S. Melendez, M. P. McGarry. "Computation Offloading Decisions for Reducing Completion Time". In: (Aug. 20, 2016). arXiv: 1608.05839 [cs.DC] (cit. on p. 24).

[Nie19] M. Nielson. *Neural Networks and Deep Learning*. 2019. URL: http://neuralnetworksanddeeplearning.com/ (cit. on p. 19).

[Ola15] C. Olah. *Understanding LSTM Networks*. Aug. 27, 2015. URL: https://colah.github.io/posts/2015-08-Understanding-LSTMs/ (cit. on p. 19).

[PC95] M. P. PERRONE, L. N. COOPER. "When networks disagree: Ensemble methods for hybrid neural networks". In: *How We Learn How We Remember: Toward an Understanding of Brain and Neural Systems*. WORLD SCIENTIFIC, Sept. 1995, pp. 342–358. DOI: 10.1142/9789812795885_0025 (cit. on p. 25).

[Rog] G. Rogova. "Combining the Results of Several Neural Network Classifiers". In: *Classic Works of the Dempster-Shafer Theory of Belief Functions*. Springer Berlin Heidelberg, pp. 683–692. DOI: 10.1007/978-3-540-44792-4_27 (cit. on p. 25).

[RPNU19] N. M. Rezk, M. Purnaprajna, T. Nordström, Z. Ul-Abdin. "Recurrent Neural Networks: An Embedded Computing Perspective". In: (July 23, 2019). DOI: 10.1109/ACCESS.2020.2982416. arXiv: 1908.07062 [cs.NE] (cit. on p. 19).

[Sat96] M. Satyanarayanan. "Fundamental Challenges in Mobile Computing". In: *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996*. Ed. by J. E. Burns, Y. Moses. ACM, 1996, pp. 1–7. DOI: 10.1145/248052.248053 (cit. on p. 17).

[Sim18] O. Simeone. "A Very Brief Introduction to Machine Learning With Applications to Communication Systems". In: (Aug. 7, 2018). arXiv: 1808.02342 [cs.IT] (cit. on p. 18).

[SZMZ20] J. Shao, H. Zhang, Y. Mao, J. Zhang. "Branchy-GNN: a Device-Edge Co-Inference Framework for Efficient Point Cloud Processing". In: (Oct. 27, 2020). arXiv: 2011.02422 [cs.DC] (cit. on p. 23).

[Ten21] TensorFlow. *Keras Documentation*. 2021. URL: https://www.tensorflow.org/guide/keras/overview (cit. on p. 20).

[TMK17] S. Teerapittayanon, B. McDanel, H. T. Kung. "Distributed Deep Neural Networks over the Cloud, the Edge and End Devices". In: (Sept. 6, 2017). arXiv: 1709.01921 [cs.CV] (cit. on p. 23).

[WHY+17] K. Wang, P.-Q. Huang, K. Yang, C. Pan, J. Wang. "Unified Offloading Decision Making and Resource Allocation in ME-RAN". In: (May 29, 2017). arXiv: 1705.10384 [cs.NI] (cit. on p. 24).

[YCB+20] B. Yang, X. Cao, J. Bassey, X. Li, T. Kroecker, L. Qian. "Computation Offloading in Multi-Access Edge Computing Networks: A Multi-Task Learning Approach". In: (June 29, 2020). arXiv: 2006.16104 [eess.SP] (cit. on p. 24).

All links were last followed on May 2, 2021.

A Kurzfassung

Aufgrund der zunehmenden Komplexität numerischer Simulationen werden die Ergebnisse normalerweise auf einem Server mit großer Rechenleistung berechnet. Um eine Visualisierung der Ergebnisse in Echtzeit für Benutzer einer AR Umgebung zu ermöglichen, sollten die Simulationen direkt auf dem mobilen Gerät ausgeführt werden. Daher wird ein alternativer Weg benötigt, um die Ausführung auf einem Gerät mit eingeschränkter Rechenleistung zu ermöglichen. Ziel dieser Arbeit ist es, die Simulation mit neuronalen Netzwerken zu ersetzen. Das resultierende NN muss Latenz- und Qualitätsanforderungen erfüllen, um die Ergebnisse in der weiteren Verarbeitung präzise visualisieren zu können.

Für die Auswertung wird das Prinzip einer verteilten Netzwerkarchitektur verwendet. Dort wurde das Zusammenspiel eines Netzes auf dem lokalen Gerät mit einem Netz auf einem nahe gelegenen Server simuliert. LSTM-Schichten und deren Fähigkeit im Falle kontinuierlicher Daten wurden untersucht, um den Netzwerktyp auszuwählen, welcher die Simulation ersetzen soll. Das mobile Gerät konnte wärend der Ausführung genaue Updates vom Server anfordern. Zwei Operatoren zur Anfrage und interner Weiterverarbeitung von Updates wurden hergeleitet. Dies geschah, indem das Verhalten empfangener Updates in verschiedenen Input-Bereichen analysiert wurde. Dabei wurde auf die vorliegende Veränderung der Qualität der Updates geachtet und Methoden zur Kombination lokaler und empfangener Updates entwickelt. Mit Hilfe dieser Beobachtung wurde der Offloading-Operator bestimmt, welcher die Häufigkeit und Zeitpunkte der Updateanfragen ausgewählt hat. Ein Ansatz zur dynamischen Entscheidung des Offloading-Zeitpunkts wurde behandelt. Beim zweiten Operator handelt es sich um den Merging-Operator. Dieser hat vom Server empfangene Updates für die Kombination mit Hilfe der aktuellen lokalen Werten weitergeführt. Abähngig von Latenz und erwarteter Qualität der Updates wurden gewichte für die Kombination der localen und angeforderten Ergebnisse bestimmt. Dadurch wurde eine Methode entwickelt, welche dynamisch die Einberechnung des Updates anpasst.

Die Verwendung von LSTM-Netzwerken hat die Genauigkeit im Vergleich zu Netzwerken ohne diese Schichten erhöht und zu einer stabileren Ausführung geführt. Für Updates alle 10 Schritte und einer Verzögerung von 10 Schritten konnten die erarbeiteten Methoden den MAE des lokalen Netzes von 5% auf 2% senken. Dies ist eine Verbesserung um 60% gegenüber der Ausführung des lokalen Netzes. Bei schlechter Verbindung und hoher Latenz konnte der am weitesten entwickelte Merging-Operator einen Qualitätsverlust über viele Schritte hinweg verhindern. Dabei wurde für Schritte, für welche sich die Qualität der weitergeführten Updates verringert, auf eine lokale Ausführung gewechselt. Dadurch konnte in allen Eingabebereichen eine Ausführung mit den bestmöglichen Werten erreicht werden. Obwohl für das Kombinieren beider Ergebnisse weitere Kosten durch das Weiterführen des Updates entstehen, konnte die Methode die Laufzeit pro Schritt um 63.5% verringern.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Stuttgart, 5.5.2021, Thomas Hübschle

place, date, signature