Artificial Neural Networks for Prediction in 5G Networks

Jan Beckschewe
Technische Universität Berlin
Berlin, Germany

***Abstract*—Wireless networks fill an important need in our society in numerous applications from business to entertainment. With mobile network traffic volumes doubling approximately every two years [10], numerous technological improvements will have to be implemented to solve the challenges of ever higher data rates and new applications. Machine learning algorithms may play an important role in this quest, as they promise optimizations in the fields of network slicing [13], performance predicting, scheduling and more.**

I. Introduction

In this paper, the feasibility of applying neural networks to predict 5G performance parameters will be examined. First, a dataset will be generated from a simulated 5G environment. This will be used to design and then train an artificial neural network. After inputting certain performance parameters, other performance parameters will be inferred. Finally, a user interface for improved usability will be built. This work will address the following question:
How can artificial neural networks be used for performance predictions in 5G networks?

II. Related Work

The idea of incorporating neural networks into mobile networks, specifically 5G and future 6G networks has been researched for several components of the system. Thantharate et al. have built a system to improve the network slicing to make the network capable of better dealing with different reliability, bandwidth and latency demands on the same physical infrastructure [13]. Wang et al. apply neural networks to the physical radio layer for improvements in spectrum utilizations [14]. And Kaur et al. hint at using supervised learning for predicting numerical target values [3] but do not propose an implementation, which is exactly the gap that this work tries to fill.

A. Used tools

To generate the dataset, the 5G-LENA simulator [7] is used. The Vienna 5G simulator [8] was also considered but not chosen because the need was identified to work on different network layers, i.e. the physical layer and transport layer, which is only possible in 5G-LENA because of its full-stack capabilities inherited from its base simulator ns-3 [11]. Vienna only supports the link level.

As the main framework for the network training, PyTorch [6] was chosen over Tensorflow [1] because of its perception as more beginner-friendly [2].

To optimize the hyperparameters, Ray Tune [4] is used. The library supports automated multithreading for the individual training runs and contains support for advanced training run scheduling algorithms and early stopping. TensorBoard[1] integrates with Ray Tune and enables the visualization of different metrics during training runs.

The API for the communication between the frontend and the backend uses FastAPI[2]. For the web frontend application, the JavaScript library React [9] was utilized.

All deliverable executables are contained within Docker containers[3] and described by Dockerfiles. The connection between the individual containers as well as host network port usages are defined using Docker Compose[4].

III. Approach

The work can be split roughly into three stages:

- Familiarizing with 5G-LENA and generating the dataset in a format understandable by the later stages

[1]https://www.tensorflow.org/tensorboard
[2]https://fastapi.tiangolo.com/
[3]https://www.docker.com/
[4]https://docs.docker.com/compose/

- Defining the neural network architecture and training the neural network
- Inferring the projected result of the neural network and building an API and a web frontend around inference

A. 5G-LENA and Dataset Generation

The very first task of the project was to find a set of suitable parameters that could be used as input and output of a neural network. The 5G-LENA simulator contains a number of code examples, some of which appeared promising, because they require a few input parameters and then run a simulation, whose output is send to the command line interface. The initial idea was to modify an existing example, so as to be able to input a few physical radio parameters, such as RSSI and S(I)NR and get as output the troughput and latency. This idea was later modified because none of the existing examples were straightforward enough to modify to process new input parameters. Additionally, there were some doubts, if these parameters would even suffice for an accurate prediction. The solution chosen to circumvent these issues was using the `cttc-nr-mimo-demo.cc` unmodified. This demo was mostly designed to simulate the performance of different MIMO configurations on both sides of the transmission, so the user equipment as well as the gNodeB, the stationary antenna in 5G networks. This means the new input values are the number of MIMO antenna columns and rows of both sides of the transmission, the transmission power of both sides, the frequency in GHz, the bandwidth in MHz and finally the physical distance between both sides in meters. The outputs are still the throughput in MBit/s and the delay in milliseconds.

Next, the simulation had to be automated to run thousands of times, so that the neural network would have enough data to train on. A python script was created for this purpose. Random parameters had to be generated in the correct range and of the correct type for each of the nine total parameters.

Listing 1: Random Input Parameter Generation

```
Ns3Parameter("numColumnsUe",
 ↪ LinearDistribution(1, 8, False)),
Ns3Parameter("gnbTxPower",
 ↪ LinearDistribution(1, 50, True)),
Ns3Parameter("centralFrequency",
 ↪ LinearDistribution(0.7e9, 3.5e9, True)),
Ns3Parameter("gnbUeDistance",
 ↪ LogDistribution(5, 2))
```

In Listing 1, some of the parameters can be seen. The number of columns of the user equipment is an integer between 1 and 8. The transmission power in dBm is a real number set between 1 and 50 dBm on the gNodeB. The frequency in this example can be between 700 MHz and 3.5 GHz and finally the Distance between the two transmission sides is distributed not uniformly within the possibility space, but rather lognormally, so that very large distances are less likely, but still appear occasionally within the dataset. Some of these values are chosen somewhat arbitrarily, e.g. there may be edge cases when the transmission power falls outside of this range but the values were chosen to work well within a large space of reasonably likely scenarios, so as to give a good compromise between likelihood and simulation time, as one simulation takes around 5 to 10 seconds based on the hardware used and the resource of computational time is finite.

The random parameters are then used as arguments for the ns-3 command that is being generated (5G-LENA is an extension of ns-3 and the commands are executed by the base program). The python script then executes the full ns-3 command with its subprocess library in order to capture the standard output of the ns-3 process. Because the ns-3 simulation results are written to the stdout, they can be captured. The python script then uses a regular expression to only grab the specific parts of interest, which in this case are the throughput and delay and then finally writes the data including the random input parameters into a new line of a CSV file.

One more problem to address is that the 5G-LENA simulations only fully utilize one core at a time, so using the python built-in multiprocessing library, all compute resources of the system can be utilized by running multiple simulations in a pool with the size of the system thread count.

Because the simulation runs for such extended periods of time, it becomes difficult to run on desktop hardware, which is needed for different tasks throughout the day. The simulations were therefore executed on a cloud server, which was able to make extensive use of the multi core optimizations. This server mandated the use of Docker containers for isolating the environment from the rest of the system. A Dockerfile was created, which orchestrates the installation of ns-3, 5G-LENA, Python and more utilities needed, as well as a docker-compose file to describe the network ports and the build directories. Both were later extended for the additional demands of the other stages of the work.

B. Neural Network Architecture and Training

A basic neural network was designed using PyTorch. While the input and output layers are already set to be the nine 5G-LENA input parameters and two outputs throughput and delay, the hidden layers and all hyperparameters require more experimentation. Listing 2 shows that the network was essentially just a feedforward neural network with two hidden layers.

Listing 2: Early code of the model itself

```python
model = nn.Sequential(nn.Linear(n_input,
 ↪  n_hidden),
            nn.ReLU(),
            nn.Linear(n_hidden, n_hidden2),
            nn.ReLU(),
            nn.Linear(n_hidden2, n_out),
            nn.Sigmoid())
```

The first iteration used common literature values for all parameters, such as 0.01 for the learning rate and 15 hidden neurons in the first hidden layers. The intention was to test the general feasibility of neural networks. There were serious doubts if the data would even allow any accurate prediction of the output, as a subjective test by humans found some predictions to be very difficult to make. The first results were underwhelming, as they inferred exactly either 0 or 1 for any input set. Extensive investigations concluded that normalizing or scaling may improve results, as neural networks may not performing well, if some values are orders of magnitude larger than others, such is the case for the frequency in hertz, around 10^{10} compared to antenna count, around 10^0 to 10^1. With scaling enabled, the first training iterations concluded with losses of approximately 0.05 using the Huber loss function.

1) Accuracy Metric: As the task at hand is essentially a regression task, there is no default accuracy measure. In comparison, classification tasks can use the share of correctly classified items divided by the total classified items. In regression, the output of the error function itself is usually used as the performance metric. This means that results obtained by different error functions are not comparable. Inspired by the binary accuracy measure in classification, this work defines a sample to be accurate, if it falls within a certain share of the true value, with 95% being chosen. There are two different accuracy values for both of the output parameters. This was only done late into the project, so most tests still only use the loss value as a metric.

2) Optimizing the Hyperparameters: The main addition to PyTorch for the hyperparameter tuning is the library Ray Tune. The initial approach to using it was to randomly go through the hyperparameter space and hope for values with low loss. Listing 3 lists some of the parameters. This approach however does not scale well, as the computing time grows exponentially with hyperparameter count.

Listing 3: Random Input Parameter Generation

```python
space = {
    "n_hidden":
     ↪  scope.int(hp.quniform("n_hidden", 5,
     ↪  100, q=1)),
    "n_hidden2":
     ↪  scope.int(hp.quniform("n_hidden2", 1,
     ↪  15, q=1)),
    "batch_size":
     ↪  scope.int(hp.quniform("batch_size",
     ↪  8, 128, q=1)),
    "learning_rate":
     ↪  hp.loguniform("learning_rate", -10,
     ↪  -1),
    "momentum": hp.uniform("momentum", 0.1,
     ↪  1.0),
    "dropout": hp.uniform("dropout", 0, 1),
}
tuner = tune.Tuner(do_training,
 ↪  param_space=space)
results = tuner.fit()
```

Instead, a more manual approach was opted for. One parameter is selected to be investigated with different potential values. The training runs are executed and once finished, the one with the lowest loss is set as a constant for the next runs where other hyperparameters are investigated. The visualization tool Tensorboard can give additional insight into the training runs in some occasions. figure 1 shows the epoch on the X-axis and the loss on the Y-axis. In this particular case the batch size is investigated, which is why the different colors represent the different batch sizes, e.g. yellow represents batch size 8. Tensorboard also gives information on the stability of the value by proving an interpolated as well as the raw plot. Additionally, it can inform about overfitting, because when the network does overfit, the graph will go back up from its lowest point on the loss axis, which appears to be what is happening with the pink line in figure 1, which represents batch_size=4, however some case needs to be taken because this can be misleading, as in this particular case it turned out to be caused by random noise and the value starts falling again after more epochs.

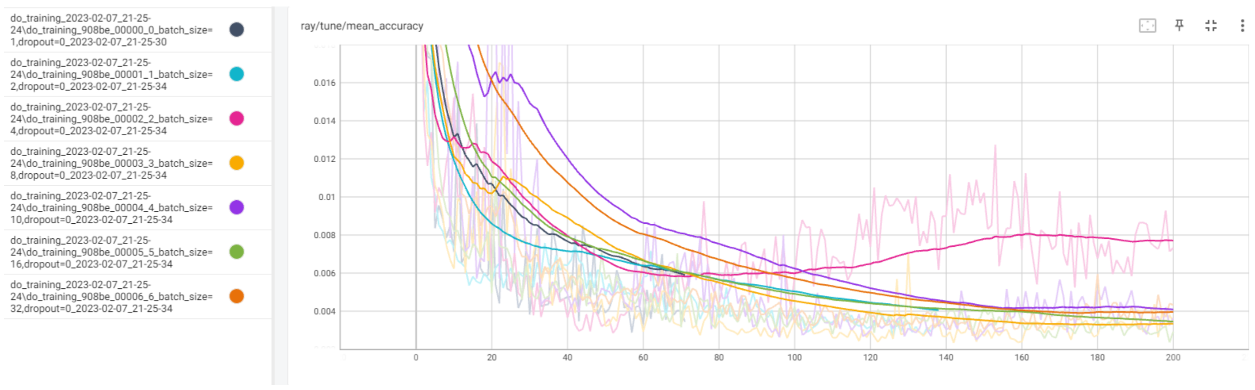


Fig. 1: Loss value by batch size. X-Axis: Epoch, Y-Axis: Huber Loss

Some parameters proved to be more influential than others. Runs with different number of hidden layers are seen in figure 2 with the same axes as before. While most hyperparameters only affect the speed of convergence, this parameter can be one deciding factor, whether the loss rate ever converges at a low value. The results demonstrate that the first hidden layer needs at least around 20 neurons to perform optimally. Another interesting observation is that a reduced batch size, increases the computing time of the training. This makes intuitive sense, because the gradient descent is executed more frequently and increases computational demands. Most other values were found to be best in close proximity to common literature values, such as learning rate around 0.01 and momentum around 0.95. To avoid overfitting, dropout [12] was introduced, however even at large epoch counts, no overfitting was encountered and dropout even increased losses, so it was removed again.

3) Computation Offloading: One of the common ways to optimize neural networks, is to offload computations to secondary hardware optimized for matrix multiplication, which is often times a graphics processing unit (GPU). This was tried in this work by utilizing the CUDA extension to PyTorch, which allows copying matrices and models onto NVIDIA graphics cards. The success of this turned out to be very limited. Despite the intuition of performance benefits, the actual performance was approximately half of performing the same calculations on the main processor. The assumed cause for this is, that the matrices used in the neural network are comparitively small. Matrices in the domain image processing are frequently among the thousands $\times$ thousands, whereas in this work, the matrices not larger than batch size $\times$ batch size, with batch size being set to 30. This likely means that more time is spent copying over the data than the time spent performing the multiplications. Other PyTorch users agree with this suspicion[5].

4) Mishaps at Model Design: The fact that in the early stages of the training effort, the model only ever inferred either 0 or 1 as output, was circumvented using the MinMaxScaler, but the true reason was not identified until much later. The reason was the sigmoid function behind the last linear layer of the network, which seems obvious at hindsight. The sigmoid function is usually called at the last stage of a classification task to make the output be a probability. However, removing this erroneously placed function call, against the intuition, did not improve the network. This is likely because it gives the network a sort of anchor, because without the sigmoid function, the network frequently predicts negative throughput and delays. It is however also possible that it is really caused by optimizing the hyperparameters too much, so that it essentially overfits in the hyperparameter space with small changes to the model definition. This is another reminder to understand the effects of every single line of code adopted from the internet.

C. *Inference and User Interface*

In order for the neural network to be useful, it needs to be able to infer outputs based on inputs. The simplest way to achieve this would be simply this code:

```
model.eval()
model([all_nine_input_values])
```

However, for the model in this project, some extra steps are needed. First, the calculated model parameters need to be loaded from the

[5]https://stackoverflow.com/questions/51179133/gpu-performing-slower-than-cpu-for-pytorch-on-google-colaboratory

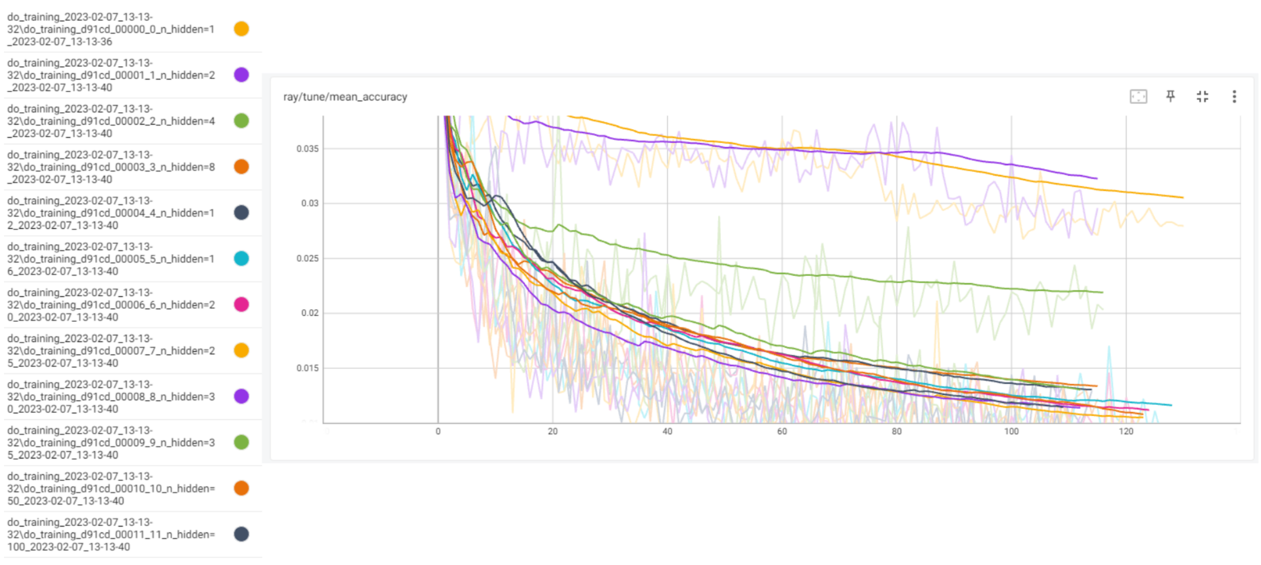


Fig. 2: Loss value by hidden layer count. X-Axis: Epoch, Y-Axis: Huber Loss

file system along with the parameters for the scaler. Then the input vector needs to be scaled to the scaled space and finally, after the aforementioned code is executed, the scaling needs to be inverted back to the original space.

Some of the steps in running the simulation can be difficult for the untrained user. Installing ns-3, which is only available on Linux, is difficult because numerous packages need to installed. Installing 5G-LENA is difficult because it needs to be compiled from source, and requires the user to choose the correct version. Executing the Python code within the Jupyter Notebook requires some programming knowledge and installing the dependencies.

All of these issues are solved with the web-based frontend application combined with the containerization with Docker.

A REST API was created using FastAPI for the communication between the web frontend and the predicting backend. In listing 4, the code for the prediction resource, which is the only resource of the API, is given. A POST request can be sent to /predict/{method}, with method being either ”nn” for neural network or ”ns3” for the simulation. In the body of the request, the nine parameters are expected. This design allows getting both the inferred results from the neural network as well as the simulated results from 5G-LENA asynchronously, which is useful because the simulation takes significantly more time.

Listing 4: A FastAPI resource for the prediction

```python
@app.post("/predict/{method}")
def predict(api_params: InputParams, method:
 ↪  str):
    throughput, delay, _ =
     ↪  run_nn(api_params.params) if method
     ↪  == "nn" else
     ↪  get_ns3_sim_result(api_params.params)
    return {"input": api_params.params,
     ↪  "output": {"throughput [MBit/s]":
     ↪  throughput, "delay [ms]": delay}}
```

1) Frontend: The frontend itself is using React and bootstrapped using Vite. It is kept to a single page where some default parameters are predefined for the user to change and a single button labelled ”Run Simulation”, which, when pressed, causes the two requests to get sent to the backend. Once the responses arrives, the two divisions below will be populated from the results generated using the two different methods.

The visual components are written in TSX, which is a TypeScript superset of JSX, which itself is an extension to JavaScript with the additional ability to write HTML within it. In listing 5 the table of outputs for either the neural network or 5G-LENA output is defined. The outer blocks of the excerpt are made up of HTML whereas the ”{” marks the beginning of a TypeScript block, which in this case maps an array of the parameters and corresponding metadata to the HTML table entries.

Listing 5: Displaying the table of predicted values

```tsx
<table>
    <tbody>
    {e.arr.map(param => <tr key={param.name}>
        <td style={{textAlign:
         ↪  "right"}}>{param.name}:</td>
        <td>{param.value?.toFixed(2)}
         ↪  {param.value ? param.unit :
         ↪  null}</td>
    </tr>)}
    </tbody>
```

```
</table>
```

D. *System Architecture*

The schematic of the system architecture is shown in figure 3. The decision to place all system components inside Docker containers was made because for the generation stage, it was already explained to be necessary and continuing with the rest of the system required reduced effort. Which parts are contained within which container is not a straightforward decision. Placing only ns-3 and 5G-LENA within its own container would improve modularity, however doing so would require some sort of communication with the other containers. The only two ways of communicating between Docker containers are either through the file system or through networking[6]. Using the file system is infeasible in this case because a direct request/response pattern is needed and using the network would work, but it would require an additional server application within the ns-3/5G-LENA container. Therefore, the trade-off to improve code cohesion at the cost of reduced modularity was made.

IV. Evaluation

The model is evaluated based on the loss value of the loss function as well as the two accuracy functions defined in this work. The results can only be compared with different iterations of this same work, because the dataset generation is part of this work, so there are no reference figures. Additionally, the computational performance of the neural network approach is compared to the original simulation.

All used metrics were identified to be flawed in some way. The loss is already known to be incomparable between loss functions. The defined accuracy metric is comparable, however a larger value does not necessitate a better performance overall. Manual spot checks identified that using the Mean Square Error function (MSE), the results are highly mixed between very accurate samples and grossly miscalculated samples. The Huber Loss function performs worse on the accuracy metric, yet delivers a more consistently mediocre result.

It can also be noted in table 1 that the accuracy of the throughput is much higher than that of the delay in both cases. The assumed reason for this is that in conditions of weak reception, the delay will be very high on the order of several hundred milliseconds but then plummet to exactly zero once none of the packages arrive at the destination.

[6]https://www.tutorialworks.com/container-networking/

Table 1: Metrics based on Loss Function

	Huber Loss	**MSE Loss**
Loss	0.00299	0.00598
Accuracy Throughput	0.442	0.486
Accuracy Delay	0.088	0.102

A. *Time Performance Measurements*

This work performs very well on the metric of computational demand. In table 2, the computing time is compared between the simulation and the neural network. The neural network approach is on average around 400 times faster than 5G-LENA. Added to that, the time is more consistent because the simulation consumes more time when setting the antenna count higher, whereas the neural network time is unaffected by this parameter. All computations for this comparison were performed on a machine with an AMD Ryzen 5 3600 CPU and 16 GB of RAM within a Docker container.

Table 2: Runtime Performance compared between Simulation and Neural Network

	5G-LENA	**NN**
Average Runtime [ms]	4231	11.1
Standard Deviation Runtime [ms]	2895	5.3

B. *User Interface Demo*

A screenshot of the user interface can be seen in figure 5. The upper section contains the input fields and the button to start the prediction and the two lower sections give the output of the neural network and the simulation.

V. Conclusion and Future Work

The viability of using neural networks to predict performance parameters in 5G networks has been demonstrated in this work. The 5G-LENA network simulator was used to generate a dataset. This dataset was used train and test the a neural network with the goal to predict the performance parameters of the 5G user equipment. Additionally, a user interface for a reduced entry barrier was created.

Future work is needed for applying nonlinear neural network architectures, such as recurrent neural networks [5], to the problem, which may possibly increase the accuracy but weren't implemented in this work due to time constraints.

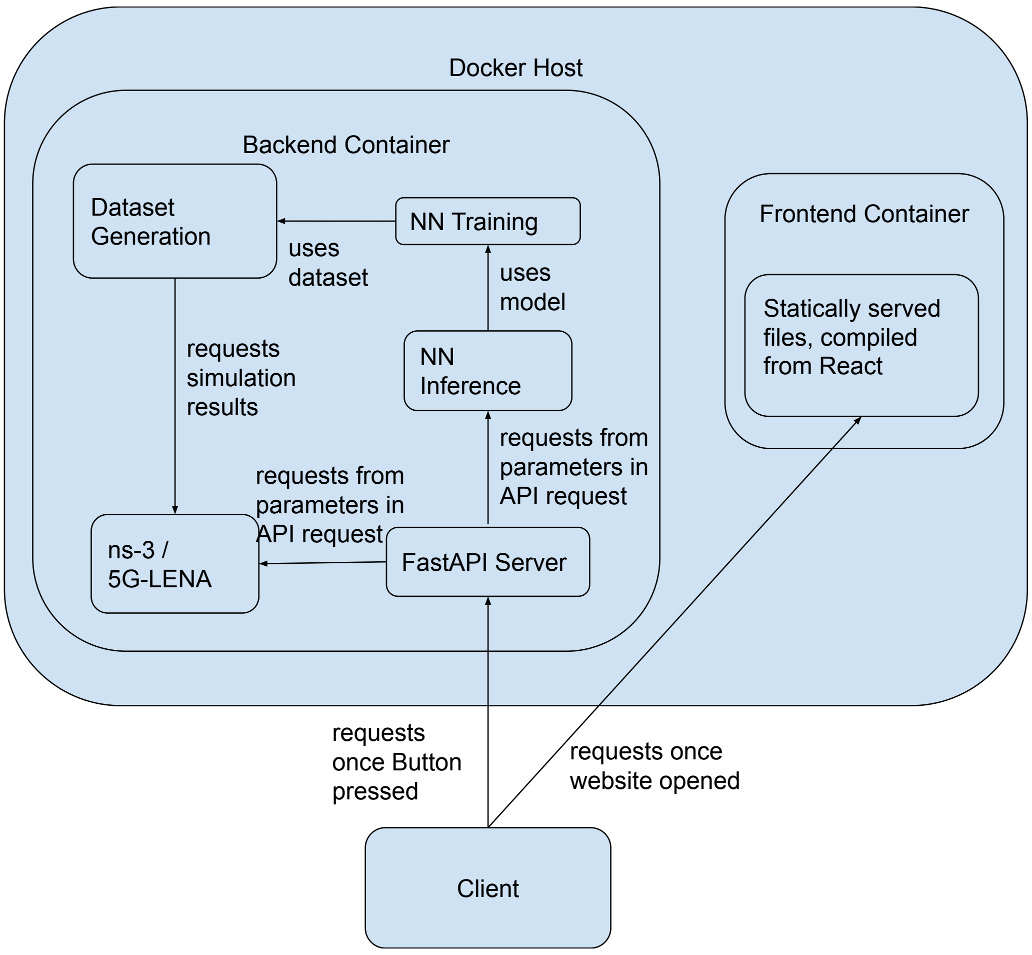

Docker Host
Backend Container
Dataset Generation
NN Training
uses dataset
uses model
requests simulation results
NN Inference
requests from parameters in API request
requests from parameters in API request
ns-3 / 5G-LENA
FastAPI Server
Frontend Container
Statically served files, compiled from React
requests once Button pressed
requests once website opened
Client

Fig. 3: Schematic of the Architecture of the System

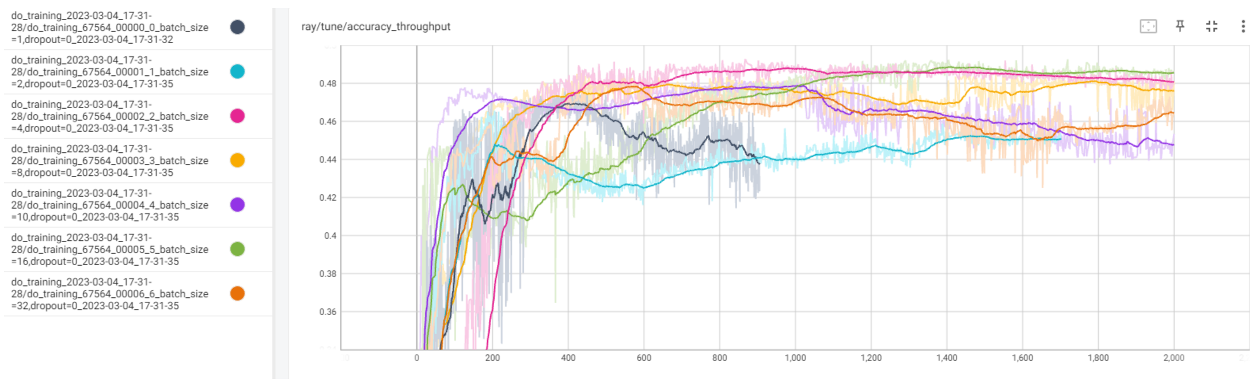
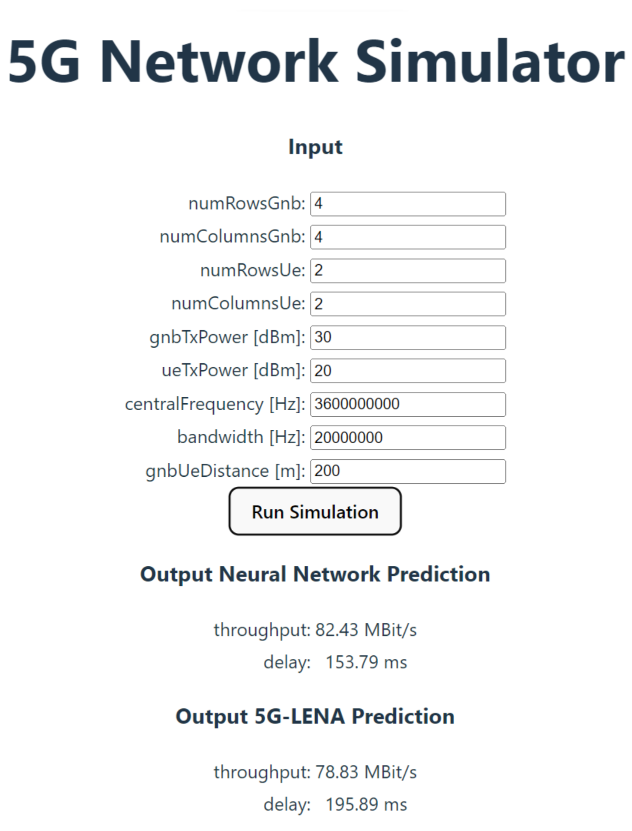


Fig. 4: Accuracy of Throughput Prediction. X-Axis: Epoch, Y-Axis: Accuracy

Automatic hyperparameter optimization could also improve accuracy and remove the need for tedious manual optimization. Ray Tune features algorithms for early stopping and advanced scheduling algorithms. Additionally, techniques for a more consistent neural network similar to the work by Wang et. al. [15] could be implemented, because of the lack of consistency identified in spot checks. Finally, the enormous performance benefits of neural networks demonstrated in this work mean that it may make sense to use the approach to create throughput maps, as these could be much more detailed at the same generation time as current methods.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *Osdi*, volume 16, pages 265–283. Savannah, GA, USA, 2016.


5G Network Simulator

Input

numRowsGnb: 4
numColumnsGnb: 4
numRowsUe: 2
numColumnsUe: 2
gnbTxPower [dBm]: 30
ueTxPower [dBm]: 20
centralFrequency [Hz]: 3600000000
bandwidth [Hz]: 20000000
gnbUeDistance [m]: 200

Output Neural Network Prediction

throughput: 82.43 MBit/s
delay: 153.79 ms

Output 5G-LENA Prediction

throughput: 78.83 MBit/s
delay: 195.89 ms

Fig. 5: User Frontend Interface

[2] Mihai Cristian Chirodea, Ovidiu Constantin Novac, Cornelia Mihaela Novac, Nicu Bizon, Mihai Oproescu, and Cornelia Emilia Gordan. Comparison of tensorflow and pytorch in convolutional neural network-based applications. In *2021 13th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, pages 1–6. IEEE, 2021.

[3] Jasneet Kaur, M Arif Khan, Mohsin Iftikhar, Muhammad Imran, and Qazi Emad Ul Haq. Machine learning techniques for 5g and beyond. *IEEE Access*, 9:23472–23488, 2021.

[4] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.

[5] Larry R Medsker and LC Jain. Recurrent neural networks. *Design and Applications*, 5:64–67, 2001.

[6] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[7] Natale Patriciello, Sandra Lagen, Biljana Bojovic, and Lorenza Giupponi. An e2e simulator for 5g nr networks. *Simulation Modelling Practice and Theory*, 96:101933, 2019.

[8] Stefan Pratschner, Bashar Tahir, Ljiljana Marijanovic, Mariam Mussbah, Kiril Kirev, Ronald Nissel, Stefan Schwarz, and Markus Rupp. Versatile mobile communications simulation: The vienna 5g link level simulator. *EURASIP Journal on Wireless Communications and Networking*, 2018:1–17, 2018.

[9] Prateek Rawat and Archana N Mahajan. Reactjs: A modern web development framework. *International Journal of Innovative Science and Research Technology*, 5(11):698–702, 2020.

[10] Vasileios P Rekkas, Sotirios Sotiroudis, Panagiotis Sarigiannidis, Shaohua Wan, George K Karagiannidis, and Sotirios K Goudos. Machine learning in beyond 5g/6g networks—state-of-the-art and future trends. *Electronics*, 10(22):2786, 2021.

[11] George F Riley and Thomas R Henderson. The ns-3 network simulator. *Modeling and tools for network simulation*, pages 15–34, 2010.

[12] Nitish Srivastava. Improving neural networks with dropout. *University of Toronto*, 182(566):7, 2013.

[13] Anurag Thantharate, Rahul Paropkari, Vijay Walunj, and Cory Beard. Deepslice: A deep learning approach towards an efficient and reliable network slicing in 5g networks. In *2019 IEEE 10th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, pages 0762–0767. IEEE, 2019.

[14] Dan Wang, Bin Song, Dong Chen, and Xiaojiang Du. Intelligent cognitive radio in 5g: Ai-based hierarchical cognitive cellular networks. *IEEE Wireless Communications*, 26(3):54–61, 2019.

[15] Lijing Wang, Dipanjan Ghosh, Maria Gonzalez Diaz, Ahmed Farahat, Mahbubul Alam, Chetan Gupta, Jiangzhuo Chen, and Madhav Marathe. Wisdom of the ensemble: Improving consistency of deep learning models. *Advances in Neural Information Processing Systems*, 33:19750–19761, 2020.