Predicting the next scheduled task using machine learning

Sampanna Kahu
Virginia Tech

Abstract

In the modern Linux kernel, the scheduling algorithm is responsible for deciding which process gets executed. The current default scheduler in the Linux kernel, the Completely Fair Scheduler (CFS), aims to maximize the overall CPU utilization while also maximizing interactive performance [2, 9]. In this project, we attempt to learn the behaviour of CFS using recent machine learning techniques. The intuition behind doing this is to evaluate the possibility of building a generalized task scheduler for any kind of workload. The important contributions of this project are: 1) Extracting and building a CFS scheduling dataset from a running Linux kernel. 2) Training and evaluating a deep learning model on this obtained dataset.

In this project, we also discuss the possibilities of integrating this model into the Linux kernel and the possible implications of doing so. Finally, we conclude the project with conclusions and future work. The source code of the project will be made available at the following link after the final submission: https://github.com/SampannaKahu/lkp_project.git

1 Introduction (Problem statement)

Currently, the process scheduling in the Linux kernel is done by the CFS. However, CFS does not recognise each process based on it's previous execution history. Basically, it's objective is to best utilize processor time. However, the Linux kernel does not take into account the past execution history of the processes into account when scheduling a process. It also tries to preempt a running process even it is about to finish it's execution which leads to multiple context switches. However, if a scheduler could take into account execution history of a process and if it could also predict a process is going to complete it's execution on the processor, it could let the process finish and then schedule the next process. This could improve the overall performance of the kernel since this could lead to lesser context switches.

The possibilities of this project are numerous. For example, if the scheduler could predict the amount of time a process is going to execute, it could make a well-informed decision about whether to schedule a process at a certain time or not. Further, the prediction of process execution could be extended to any domain where scheduling is used. For example, in Android, we know that opening some apps take significant amount of time because the app's data and resources need to loaded into the memory. However, if the scheduler could predict this in advance, the necessary data could be pre-fetched into the memory to ensure a better user experience when using the device.

2 Background

This section talks about the background about the various concepts and tools used in this project.

Almost all machines these days are multi-processor machines. This means the multiple processors are executing tasks parallely. This brings in the notion of hardware threads. As explained in [10], hardware threads are capable of handling a single process. This means that, the hardware is controlling the threading. Completely Fair Scheduler (CFS) decides which process will be executed next and the execution of which process will be preempted. As mentioned in the official documentation [2], CFS basically models an "ideal, precise multi-tasking CPU" on real hardware. On real hardware, only a single task can be run at any given instant. Thus, it is necessary to introduce the concept of "Virtual Runtime" which specifies when the next time-slice of a task would start execution on an ideal multi-tasking CPU [2].

The CFS is triggered every-time the system-clock/timer is triggered. A scheduling routine runs on each timer shot and checks whether any new process needs to be scheduled. This mechanism uses multiple data structures some of which are task-struct, queue of waiting processes, the rb-tree which signifies which task has spent how much amount of time on

the CPU, and so on. However, this all happens internally in the kernel and no concrete data is available to get better visibility in this scheduling process. Thus, many profiling tools have been built to collect these kind of data. One of such tools is the *perf-sched* [5] tool. This tool uses the internal kernel functions and collects metrics from the using kprobes and kretprobes. Each version of the *perf-sched* tool is specific to a certain kernel because the function signature of the kernel change with each new version. The data returned by this tool can be stored in *.csv* format and sued for further downstream analysis.

NGinx [7] is a popular open-source software for high-performance web-servers. It is also commonly used as a proxy and sometimes as a load-balancer as well. It is one of the fastest server architectures available openly and constantly beats the performance of the Apache LAMP stack. Because of its performance and popularity it is commonly used in academic research papers as part of experiments [14].

Ab [8], also known as ApacheBench, is a popular tool by Apache for simple bench-marking and load-testing of http endpoints. This is a command-line tool and can send numerous parallel requests to any given http endpoint. Due to its easy setup, it is the logical choice to use it in experiments.

LSTMs are a popular tool for modelling time-series data, or any general sequence of data. They were first proposed by Hochreiter and Schmidhuber in 1997 in their famous paper [12]. LSTMs use forget and remember gates for better remembering both the long and short term instances of data they encountered. Further, they are easier to train that a regular recurrent neural network because there is an easy/shorter path for gradients to flow backwards between the LSTM cells.

3   Related work

In 2005, Negi. et. al, proposed a new method for predicting the next process to be scheduled. They used machine learning techniques to learn the CPU time-slice utilization behaviour of known processes with the goal to minimize the TAT (turn around time) of a process. After collecting the data, they used decision trees to predict the next task to be scheduled. However, our work differs from their because we use deep learning techniques (LSTMs) to model the CPU time-slice utilization behaviour instead of a decision tree. Although LSTMs might be computationally heavier than decision trees, the aim of our work is to evaluate their performance for this particular application.

An interesting application of machine learning towards Linux kernel was demonstrated by Sasha Levin and Julia Lawall in 2018 [6]. They used machine learning to identify patches that fix bugs from the ones that don't. As input to

their machine learning model, they used the source code of the patch. Then, based on the past history of which patches were applied to the stable tree, they created the ground truth labels and trained their machine learning model. As a result, their model was able to classify patches with an accuracy of more than 80%. However, this is not a direct application of machine learning inside the running Linux kernel and hence is different that our work here.

Some work on predicting schedulers was done by Shin et. al. in 2012 [16]. As a part of this work, they predicted the usage patterns of applications running on a mobile phone. Their model predicted the usage of 9 candidate apps with an accuracy of over 87%.

4   Motivation

As mentioned in the previous section, the model developed in this project could be used to enhance the Linux scheduler. For example, if the model could predict the amount of time it is going to spend on the processor, the CFS scheduler could take a better informed decision about when to schedule this task. Further, if the model could predict how much more time a process is going to spend on the scheduler, it could preempt the process a little later or a little earlier depending upon the requirement. This could save a lot of time spent in context switching. This could speed up the Linux kernel considerably.

Further, this type of model could be potentially applied to other types of scheduling problems too. For example, in case of the Android mobile operating system, a number of applications can be installed and executed. However, it takes considerable time for some applications to load it's resources in the memory and start executing. An analogy could be drawn to the context switching between two processes which consumes vital CPU resources. A model which could predict the next task that is going to be executed could pre-load it's resources into the memory to save time and computation [11].

5   Methodology

5.1   Dataset generation:

For data generation, we used the *perf* tool available in Linux. Via this tool, it is possible to get detailed metrics around scheduling of the task. This tool needs to be run in root(*sudo*) mode. This tool generates a *perf.data* file which is a compressed version of all the collected data during the session. This file can be opened using the tool itself.

Following command is used to record the task scheduler metrics of the current kernel:
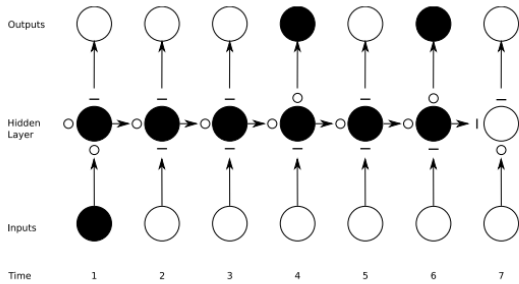
Figure 1: High-level structure of Recurrent Neural Networks [1].

Figure 2: Example of recording scheduler metrics using the *perf* command. This example test was run for 50 seconds. A total of 49.915 MB of data was collected which consisted of 422084 samples.

```
sudo perf sched record -- sleep 50
```

Listing 1: Command to record the scheduling data from the current running kernel for a total of 50 seconds. The output will be stored in a file in the same folder with the name *perf.data*.

This command shown in Figure 2 will record the scheduling metrics for all the cores and all the processes in the current running kernel. As shown in the figure, a total of about 50 MB of data was collected in 50 seconds which consisted of about 422084 samples. However, it is worth noting that the amount of data collected depends the load of the system and the number of processes being scheduled. For example, if a lot of processes are getting scheduled on the system, the number of samples in the collected profiler data using *perf* will be higher.

Further, the data collected using the above command (Figure 2) can be visualized using the following command:

```
sudo perf sched map
```

Listing 2: Command to visualize the scheduling data recorded using the record command. This command expects a file named *perf.data* in the current directory.
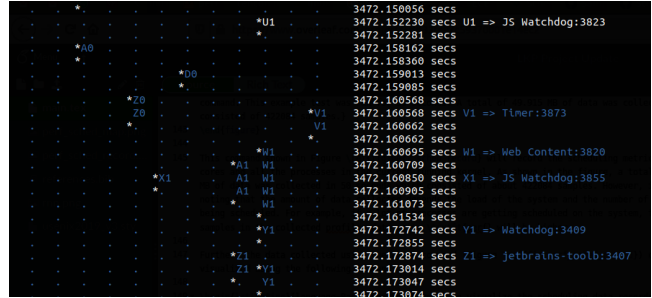
The output of the above command can be seen in Figure 3.

Figure 3: Visualization of the 'perf sched' command.

5.2 Experimental setup

5.2.1 Hardware parameters and operating system

As evident in Figure 3, the visualized data is for a multi-CPU machine. That machine can support 12 threads in total. Each CPU is executing a single process in parallel with the other 11 CPUs. However, in this project, predicting the scheduling performance of multiple CPU will complicate things. Thus, for this project, we will limit the scope to only a single CPU machine.

To accommodate this constraint, we run our experiments on a single CPU machine (i.e. a machine which supports only a single hard-ware thread). We accomplish this using a virtual machine which is configured to use only a single CPU (i.e. a single hard-ware thread). We use Ubuntu 18.04 Bionic as the operating system running in the virtual machine. This operating system is running the Linux Kernel 4.15.0-58-generic. The host operating system is Ubuntu 18.04 Bionic as well which is running the Linux Kernel 5.0.0-36-generic. The host hardware is a Dell XPS 15 laptop with a total of 12 processors of type Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz.

5.2.2 Load generation

We tried to record the data on this virtual machine using the command shown in Figure 2. However, the number of samples in this recording were quite low. Also, we observed that the CPU was idle for considerable amount of time. Upon further investigation, we found out that this was happening because the kernel was not under much load. In other words, not many applications were running on the operating system. This was causing the processor to sit idle intermittently. Thus, to mitigate this issue, we induced artificial load on the kernel.

Taking inspiration from one of the experiments in [14], we started the *nginx* server with the default settings. However, simply starting the nginx server would not induce load on kernel. However, we tackled this by keeping the nginx server

busy. Since nginx is an http web server, it will become busy if it is serving requests. Thus, it will be sufficient to keep sending requests to the nginx server to keep it busy.

Sending http requests to nginx was achieved by a tool called *ab*. This is the acronym for Apache Bench. This tool comes as a part of the the apache2-utils package. It sends dummy http requests to a given IP address. It stops sending requests when it has finished sending the configured number of requests. We configure this tool to send http requests to nginx which is running on the same virtual machine. As a result, both nginx and ab will generate load for the processor. We start this tool using the following command:

```
ab -n 1000000 -c 5 http://10.0.2.15/
```

Listing 3: Command to send one million requests to the given IP address using ab (Apache Bench).

With the given configuration of hardware (single processor i7 virtual machine), it takes more than 100 seconds for these 1 million http requests to complete. Thus, we record the scheduler profiler data using the command mentioned in Figure 2 during this time-frame.

5.2.3   Data pre-processing

The data collected by the the *perf* tool is not suitable to be given as an input to the machine learning model. In it's raw format, it encodes the task names into alpha-numeric format based on occurrence (see Figure 3). Thus, we need to convert these alphanumeric code into their real task names. To do so, we write a simple python script to read the mapping between each alphanumeric code with it's corresponding task name and replace those codes with their real task names. The initial statistics of the collected dataset can be seen in Figure 4.

Further, these task names are categorical data. They are currently stored in string format because they are names. We convert them into vectors by applying one-hot encoding on these names. In the dataset that we collected, we found a total of 28 task names during the entire recording process. Thus, the dimension-size of these one-hot vectors would be 28 after applying the one-hot encoding.

The data recorded by *perf* also contains the time-stamp at which a process was scheduled. The significant/meaning of the absolute value of this time-stamp is not clear. However, all we are interested for the scope of this project is the time duration between two consecutive schedules of processes. This data can be easily obtained from the time-stamps by differentiating them. In essence, we take the differences of consecutive time-stamps to differentiate them. We drop the last record. Further, we scale and shift this data to get zero

Figure 4: The number of schedules of a given process. This was captured when 'ab' was generating load on 'nginx' by sending a total of 1 million http requests.

mean and unit variance.

One unexpected issue encountered during the pre-processing of the data was that one of the differentiated time values was abnormally higher than the others. Its value was more than one complete second. This meant that the corresponding process took more than 1 second to be scheduled, execute its time-slice and get preempted. However, that time duration is unusually high for any process in the kernel. Hence, we got rid of that entry from the dataset assuming that it was a one-off issue.

5.3   Model:

The collected dataset is a series of readings collected over time. Which means that this is a time-series dataset. According to the latest research in the field, Deep Learning models are the best fit of handling large amounts of data. Specifically, the Long-Short-Term memory (LSTMs) [12, 13] (a type of Recurrent Neural Networks (RNNs) [15] See figure 1) are best suited to handle time-series data.
We train an LSTM model using this data to predict the next task that is going to be scheduled or preempted. It is important to note that the model is trained on the difference of the consecutive time intervals between two schedules. In other words, we record the time intervals between two consecutive schedules and differentiate them. Further, we select sequences of appropriate length for training the model. The model is trained until the test loss reasonably stabilizes. Further, we use the LGFBS optimizer for 30 epochs or until the loss converges. See figure 5 for the computational graph of our LSTM model generated using PyTorchViz [17].
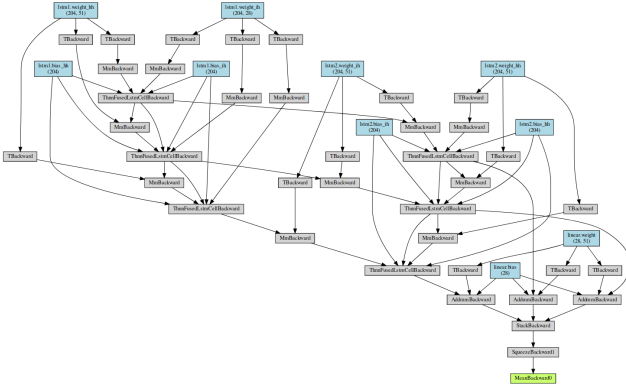
4

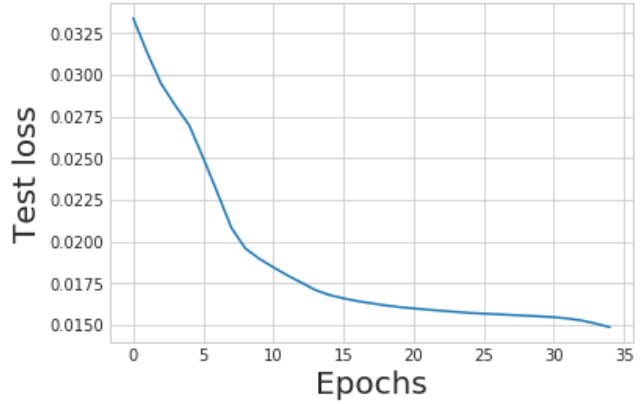Figure 5: The computational graph of our model.

Figure 6: Plot of test loss vs. the epochs.

6 Results

We computed and recorded the test loss for each epoch during the training process. Figure 6 shows the test loss as the training process progresses. From this figure, we can observe that initially the test loss was about 0.0325. However, as the training progresses, the loss started to reduce and stabilized after 30 epochs. Beyond this point, training further won't lead to the further decrease in the loss and would simply lead to over-fitting. For our evaluations, we use the model saved at 30 epochs because it has the least test loss.

After the training process was complete, we provide sample input sequences to the trained model. We make the model generate predictions after the sample input sequence is over. In other words, once the model sees the entire sample input sequence, we run the model further for 1000 steps to make it predict the next schedules. Also, to get a better idea of how the model learns to predict the sequences, we make the model generate predictions after each epoch.

Figure 7 shows the prediction of the model for a sample input sequence after the model was trained for 1 epoch. The values in this figure before the 1200 mark on the x-axis represent the sample input sequence. The value after that represent the values predicted by the model. It is important to note that the model was trained on the difference of the consecutive time intervals between two schedules. In other words, we recorded the time intervals between two consecutive schedules and differentiated them. We can see that the model does not predict anything in figure 7. It is just a flat line.

Figure 8 shows the predictions of the model for the same sample input after 3 epochs of training. W can observe that the model is trying to match the magnitude of the training sequences.
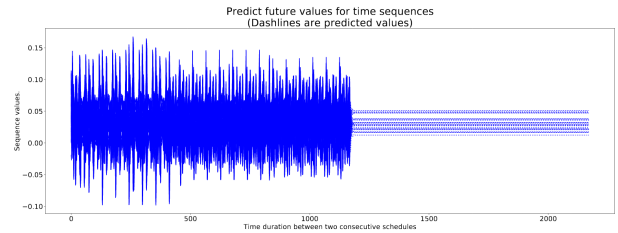

Figure 7: Prediction of the model after training the model for 1 epoch.

Further, in figure 9, the model is trained for 5 epochs. The predictions of the magnitudes is good enough. However, we can see that the model has some bias in the predictions. After a few more epochs, the model looses that bias and now the predictions look much better in figure 10.

Finally, in figure 11, the model is now trained for 30 epochs. The magnitude of the predictions match very closely with the magnitudes of the input sequence. Further, we can observe that even the patterns of values in the predicted sequence resembles the pattern in the input sequence. This indicates that model has now been trained well which is also confirmed by stabilized test loss in figure 6.

7 Discussion

From figures 7, 8, 9, 10, 11, we can see the progression of the predictions of the model as it is trained. Initially, the model was not predicting anything at all. It was simply using predicting a flat line. Further, as the training progressed, we can observe that the model slowly starts to acquire characteristics of the training data. For example, first, it starts to model the magnitude of the input sequences. Then, it starts to model the patterns in the input sequences. After a certain amount of training, it becomes difficult for even a human to
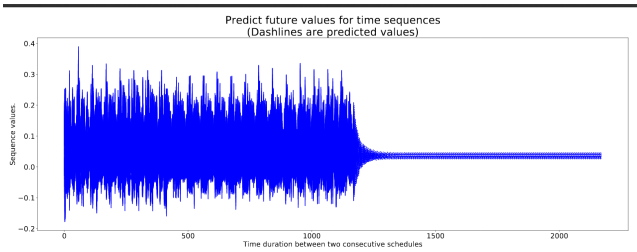
5

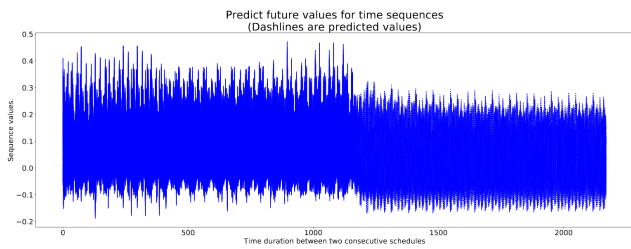Figure 8: Prediction of the model after training the model for 3 epochs.

Figure 9: Prediction of the model after training the model for 5 epochs.
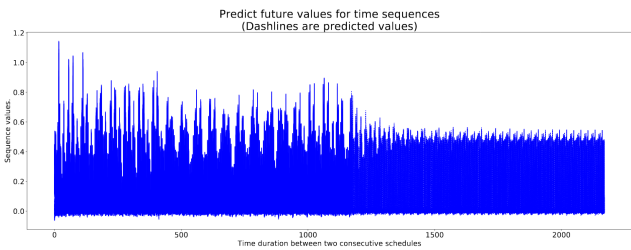

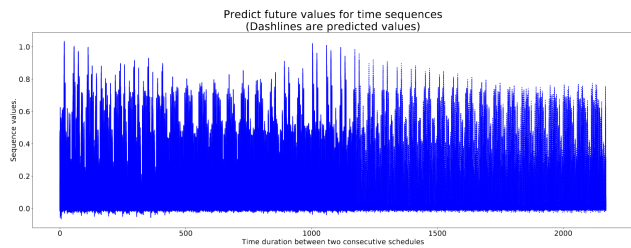Figure 10: Prediction of the model after training the model for 15 epochs.

Figure 11: Prediction of the model after training the model for 30 epochs.

distinguish between the sequences predicted by the model and the input sequence.

This capability of our model to predict the future of the scheduled tasks makes it valuable. At any given time, of the Linux kernel knows the next N tasks to be scheduled, it can take better decisions while scheduling them. For example, it could be possible to club two relatively small CPU time slices allotted for a given process into a a bigger slice. This would help reduce the number of context switches between tasks thereby improving performance.

To practically integrate our work into the Linux kernel, the biggest challenge is to deal with the predictions latencies of the model. Deep learning models are orders of magnitude slow when compared to the latencies of the scheduler. Since the scheduler runs at each context switch, it becomes crucial to bring down the latencies of the deep learning model if it ever has to replace the traditional scheduler.

Many ways have been explored till now to improve the predictions latencies of deep learning models. One of the prominent solutions is to use a better language, like C++. This approach is often used in autonomous cars where prediction latencies are important too. Another approach to dedicated hardware for running the predictions of the model (e.g. GPU, TPU, ASIC, etc). GPUs already provide a huge improvement over pure-cpu prediction latencies. Further, Google's TPUs [3] are more specialized for deep learning. Finally, ASICs can theoretically bring down the prediction latencies by orders of magnitude and seem to be an interesting avenue to explore [4].

8 Development status

8.0.1 Completed tasks

- Literature survey, background and related work is complete.

- The experimental setup for data collection and generation have been figured out.

- The actual data collection and it's processing scripts are complete.

- Building, implementing and training the LSTM model on the collected dataset.

- Exploring/discussion the possibilities of integrating this the Linux kernel along with its implications.

9 Conclusion

In this project, we introduced the problem of predicting the scheduling of processes in the Linux kernel. Further, we discussed our motivation to develop a solution to predict the

6

scheduling behaviour of Linux kernel and the benefits it will provide. We then discussed in detail our methodology to implement our proposed solution. This includes the process of generating the data, the experimental setup, and the details of training the deep learning model. Finally, we show our results, discuss them in detail and conclude.

10 Future work

To improve this further, we could use a more versatile dataset. In other words, we used only Apache Bench and NGinx to generate the load on the kernel. However, in the future, we could use a more versatile and natural data for training (for example, the scheduling data of a public server).

References

[1] A Beginner's Guide to LSTMs and Recurrent Neural Networks. http://skymind.ai/wiki/lstm.

[2] CFS Scheduler — The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html.

[3] Cloud TPU. https://cloud.google.com/tpu/.

[4] Deep Learning on ASIC. https://www.easics.com/products/deep-learning-asic.

[5] Linux Performance. http://www.brendangregg.com/linuxperf.html.

[6] Machine learning and stable kernels [LWN.net]. https://lwn.net/Articles/764647/.

[7] What is NGINX? https://www.nginx.com/resources/glossary/nginx.

[8] ApacheBench. https://en.wikipedia.org/w/index.php?title=ApacheBench&oldid=920436197, October 2019. Page Version ID: 920436197.

[9] Completely Fair Scheduler. https://en.wikipedia.org/w/index.php?title=Completely_Fair_Scheduler&oldid=918940983, October 2019. Page Version ID: 918940983.

[10] Multithreading (computer architecture). https://en.wikipedia.org/w/index.php?title=Multithreading_(computer_architecture)&oldid=895110143, May 2019. Page Version ID: 895110143.

[11] Ricardo Baeza-Yates, Di Jiang, Fabrizio Silvestri, and Beverly Harrison. Predicting the next app that you are going to use. pages 285–294, 02 2015.

[12] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. 1999.

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

[14] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-weight contexts: An {OS} abstraction for safety and performance. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 49–64, 2016.

[15] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Eleventh annual conference of the international speech communication association*, 2010.

[16] Choonsung Shin, Jin-Hyuk Hong, and Anind K. Dey. Understanding and prediction of mobile application usage for smart phones. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, pages 173–182, New York, NY, USA, 2012. ACM.

[17] Sergey Zagoruyko. szagoruyko/pytorchviz. https://github.com/szagoruyko/pytorchviz, December 2019. original-date: 2018-01-30T15:37:55Z.