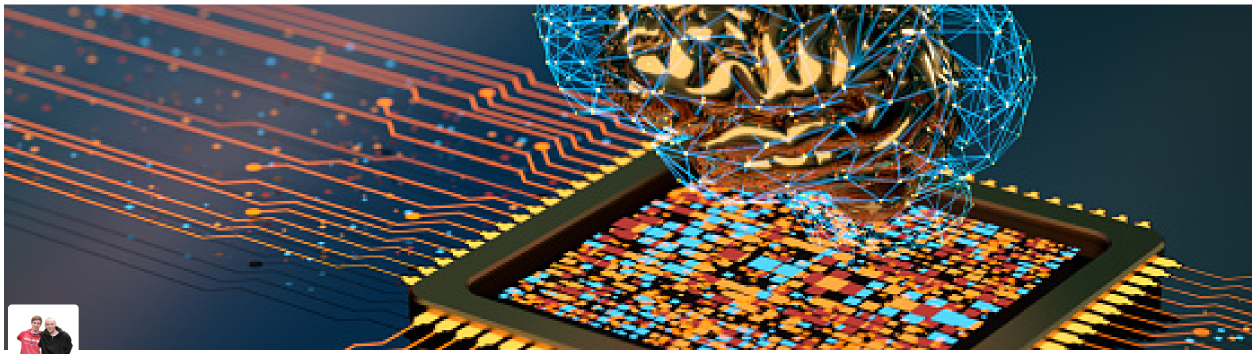
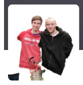



Deep Learning: DeltaCNN Paper Reproducibility Project

24/04/2023

Lucas Hofman	4833287	l.b.hofman@student.tudelft.nl
Ruben Overwater	4832205	r.a.a.overwater@student.tudelft.nl

Preface

Dear reader,

This blog post is written for the course CS4240, Deep Learning at Delft University of Technology. The blog post describes the process used to reproduce the paper "**DeltaCNN: End-to-End CNN Inference of Sparse Frame Differences in Videos",** as can be found via the link below. The codebase behind the paper can be found on GitHub or via the link below.

The general structure of the blog post is as follows. First, a brief overview of the benefits of DeltaCNN is provided. Next, the plan to tackle reproduction is laid out. During the reproduction, some problems were encountered. These are discussed in the following sections. In the final section, our results are presented.

▼ Reference Links

DeltaCNN: End-to-End CNN Inference of Sparse Frame Differences in Videos

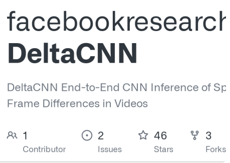
Convolutional neural network inference on video data requires powerful hardware for real-time processing. Given the inherent coherence across consecutive frames, large parts of a video typically...

https://arxiv.org/abs/2203.03996

GitHub - facebookresearch/DeltaCNN: DeltaCNN End-to-End CNN Inference of Sparse Frame Differences in Videos

DeltaCNN End-to-End CNN Inference of Sparse Frame Differences in Videos - GitHub - facebookresearch/DeltaCNN: DeltaCNN End-to-End CNN Inference of Sparse Frame Differences in Videos

https://github.com/facebookresearch/DeltaCNN


facebookresearch
DeltaCNN
DeltaCNN End-to-End CNN Inference of Sp
Frame Differences in Videos
1 Contributor · 2 Issues · 46 Stars · 3 Forks

Content

- Introduction

- Plan
- Problems
- Results
- Conclusion

Introduction

When processing static images with neural networks, lots of data is unused. Think of pedestrian detection from CCTV footage to give an example. The background will hardly change over time. This property is what the DeltaCNN paper is aiming to exploit. Instead of passing the image to the neural network, the difference between consecutive images is passed through the network. This would reduce computation and allow for speedup. To utilize this property, the authors created a package called “DeltaCNN”, based on the CUDA backend and easily implementable in Pytorch by “simply replacing the PyTorch layers with the DeltaCNN equivalent”. This package was intensively tested and this blog post was written.

Plan

The aim was to utilize the DeltaCNN package on the Human3.6M dataset. The Human3.6M dataset is an annotated dataset of humans, which can be used for pose estimation. Since the pictures are taken from a static camera, this dataset is perfect to show the performance gains of DeltaCNN. Next, utilizing google cloud and Google Colab, it was proposed to test different GPUs. In the paper, the most speedup was reached on lower-level devices and this result was to be checked by reproducing the difference in speedup across devices. Next, a study on a different dataset was proposed. For this dataset, the Virat video dataset was selected. This is an annotated dataset for detecting civilians from CCTV footage. Since the capture device of this dataset is also static DeltaCNN can be used.

Problems - Working with DeltaCNN

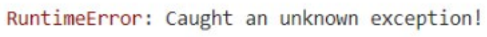
The first step in the reproducibility of the paper was to install and load the DeltaCNN extension on the PyTorch package. This is where the team encountered the first problems in the implementation. Instructions to install the package were provided through the read.me file on the GitHub repository. However, these instructions were not sufficient to get to a working install. Attempts were made on a local desktop (Windows & Linux, multiple laptops from TU Delft laptop project), google cloud services through the Linux command line, and finally on Google Colab. Most likely this is due to an improper install of a specific deprecated version of the CUDA toolkit which needs to communicate with a specific version of Pytorch and DeltaCNN. Luckily, after tweaking the installation commands, a working install was acquired on Google Colab. The used installation commands, only working in Google Colab in GPU runtime, are shown below.

```
!git clone https://github.com/facebookresearch/DeltaCNN.git
!pip install /content/DeltaCNN
```

> *Note: the pip install takes ~7 minutes*

Now that a working install was acquired, a series of attempts were made to run a ResNet-18 using the DeltaCNN layers. Unfortunately, many errors were encountered and therefore the team resorted to a much simpler implementation to gain confidence in working with the DeltaCNN package. The goal was to replace layers with DeltaCNN on a regular CNN. Therefore the CNN setup for the MNIST dataset was used and adapted. In the following subsections, the errors encountered during this implementation are discussed.

- Unknown Exception: Most of the time the team ran into the very frustrating error, “Caught an unknown exception”. This error message which is part of the DeltaCNN module made it hard to debug since very little to no information was given on the cause of the problem. After consulting with other groups who are reproducing the same paper, solutions to this problem were found.



```
RuntimeError: Caught an unknown exception!
```

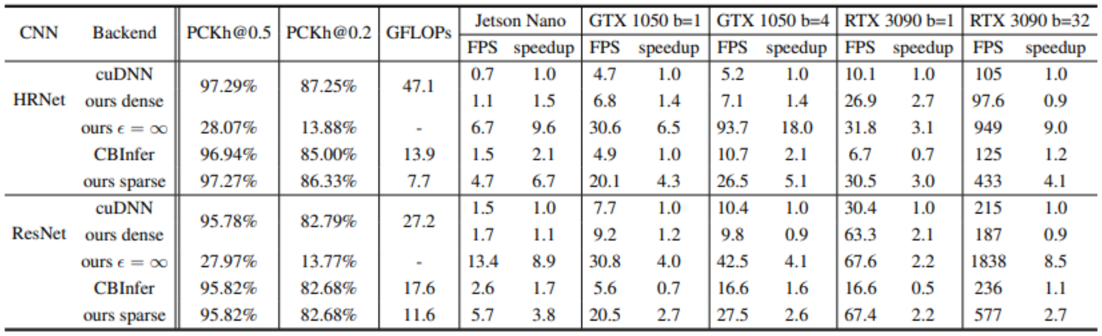
Error message encountered while setting up CUDA layers in DeltaCNN

- Shape Error: Shape errors were given when one or more DeltaCNN layers were implemented in the regular CNN. The only solution the team found to resolve this error was to constrain the batch size of the train and test set of the input to a size of 64. By canceling all other sizes, for example, the final batch which has an odd dimension, results could be produced. This had the inevitable consequence of the loss of train and test data. Therefore the full potential of the dataset is not used.

- Max Pooling Layers: To implement a max pooling layer through DeltaCNN an extension of DeltaCNN had to be used. This extension unfortunately suffered a series of default values that were not documented. Therefore it was difficult to find the problems in shape errors as a result of incorrect stride, padding, and dilution. After consulting the regular Torch documentation and diving into the DeltaCNN source code the team figured out what changes had to be made to the preset parameters in stride, padding, and dilution. This resolved the final shape errors encountered.

Results

The goal of the reproducibility part in this project is to verify the results presented in the DeltaCNN paper. In the table below the results from the paper can be found. The project is focused on the implementation of DeltaCNN on a ResNet. In the table the performance on various devices is given. The original idea was to replicate these results for different devices by using Google Colab and Google Cloud Services. Some challenges where encountered when setting up the Google Cloud Services and unfortunately due to time constraints it wasn't possible to set it up for this project. This resulted in an alternative approach, which could still reveal the possible speedup for the implementation of DeltaCNN layers. This will be further discussed in the following sections.



CNN	Backend	PCKh@0.5	PCKh@0.2	GFLOPs	Jetson Nano		GTX 1050 b=1		GTX 1050 b=4		RTX 3090 b=1		RTX 3090 b=32	
					FPS	speedup	FPS	speedup	FPS	speedup	FPS	speedup	FPS	speedup
HRNet	cuDNN	97.29%	87.25%	47.1	0.7	1.0	4.7	1.0	5.2	1.0	10.1	1.0	105	1.0
	ours dense				1.1	1.5	6.8	1.4	7.1	1.4	26.9	2.7	97.6	0.9
	ours $\epsilon = \infty$	28.07%	13.88%	-	6.7	9.6	30.6	6.5	93.7	18.0	31.8	3.1	949	9.0
	CBInfer	96.94%	85.00%	13.9	1.5	2.1	4.9	1.0	10.7	2.1	6.7	0.7	125	1.2
	ours sparse	97.27%	86.33%	7.7	4.7	6.7	20.1	4.3	26.5	5.1	30.5	3.0	433	4.1
ResNet	cuDNN	95.78%	82.79%	27.2	1.5	1.0	7.7	1.0	10.4	1.0	30.4	1.0	215	1.0
	ours dense				1.7	1.1	9.2	1.2	9.8	0.9	63.3	2.1	187	0.9
	ours $\epsilon = \infty$	27.97%	13.77%	-	13.4	8.9	30.8	4.0	42.5	4.1	67.6	2.2	1838	8.5
	CBInfer	95.82%	82.68%	17.6	2.6	1.7	5.6	0.7	16.6	1.6	16.6	0.5	236	1.1
	ours sparse	95.82%	82.68%	11.6	5.7	3.8	20.5	2.7	27.5	2.6	67.4	2.2	577	2.7

Results produced using the DeltaCNN layers, presented in the paper. [1]

DeltaCNN implementation in ResNet

In the following figure the implementation of DeltaCNN on various layer types is shown. The primary goal of the second and third level is to sparsify (increased number of zeros) the throughput such that the forward pass over the layers can be computed faster. In order to do this the difference in sequential input images is evaluated such that static features (for example background features) will only be evaluated once.

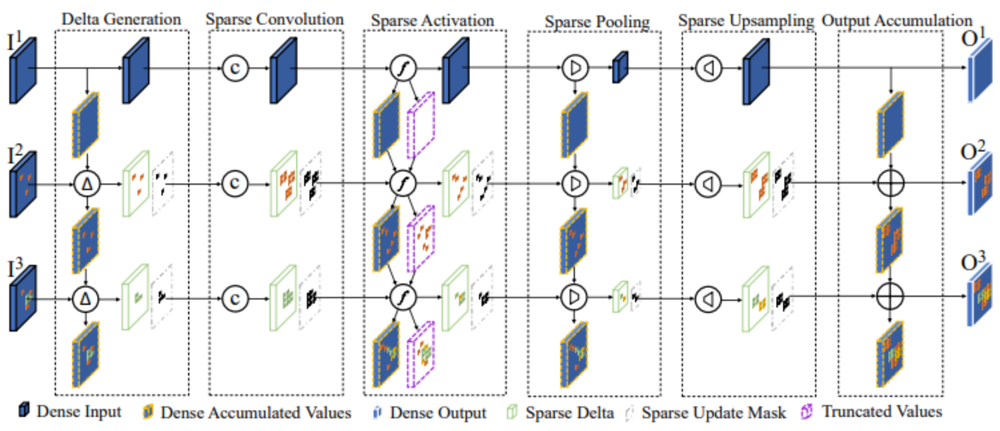
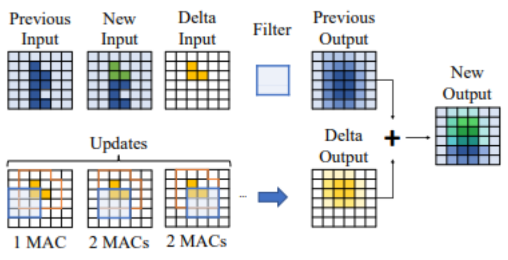


Illustration of the layer by layer implementation of DeltaCNN. [1]

After the passing the sparse layers through the full network the outputs are accumulated again. The process of accumulation is shown in the following figure. The output from the previous input is given. Then, the difference between the new and previous input is computed and is passed forward through the convolution layer. Finally, the previous output and delta output are summed to obtain the new output.



Summation of sparse layers onto a dense layer for convolutions. [1]

Simplification

As explained above, due to time constraints, the ResNet architecture could not be implemented. To still test the performance difference of DeltaCNN compared to regular CNN, a simpler architecture was chosen. It consists of the following layers:

- Convolution layer
- Max pooling layer
- Convolution layer
- Max pooling layer
- Fully connected layer

With the two convolution layers, it is expected to still be able to observe the performance gain of DeltaCNN compared to regular CNN.

Dataset switch

With this simplified architecture, the training set also needs to be simplified. Human pose estimation with this simple model is not just possible. It would therefore be very difficult to compare the (poor) performance of DeltaCNN to the (poor) performance of regular CNN. It was decided to do image classification on the MNIST dataset. This is a simple dataset, consisting of 60,000 training images and 10,000 testing images of handwritten numbers. The above-mentioned architecture was trained with both regular and DeltaCNN layers to compare the accuracy and the speed.

Dataset modification

DeltaCNN shines when sequential data is presented. To accommodate and simulate this, the MNIST dataset was ordered. As can be observed in the image below, sorting the dataset results in minor changes between consecutive images. This minor change will simulate static image conditions where DeltaCNN was designed to operate in.

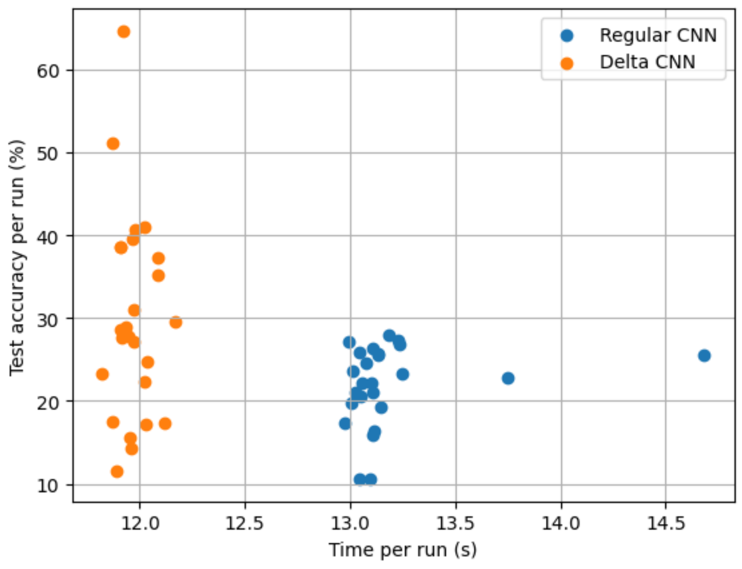
Sorted MNIST from [2].

Test Results

Data gathered using DeltaCNN on the MNIST dataset is presented below. Two networks with the same architecture were compared against each other. One of the networks utilized the “regular” CUDA backend, while the other utilized the CUDA DeltaCNN backend. First, the speedup of the DeltaCNN backend is evaluated. This is followed by some words on hyperparameter tuning

Speedup evaluation

For the speedup evaluation, both networks were trained and evaluated on the MNIST dataset for 25 runs each, both with 6 epochs. The MNIST dataset was sorted such that the difference between inputs changed as little as possible and DeltaCNN could utilize its concept. The time per run and final test accuracy is plotted for each network. The following observations can be made. DeltaCNN is faster than the regular CNN network. Although the MNIST dataset is used sorting still allowed DeltaCNN to achieve a speed-up compared to the regular CNN network. The accuracy across the DeltaCNN runs is less consistent compared to the regular CNN network. This could be explained due to the relatively low amount of epochs and runs.



Plot, top left is desired

In the following table the average runtime for both CNN's are shown. Comparing the speedup to the results in the DeltaCNN paper it becomes apparent that the reproduced speedup is on the lower bound of the speedups found in the previously mentioned table. Since the complexity of the CNN used on the MNIST dataset is far simpler than a ResNet a less significant speedup is expected. Next, the MNIST dataset is not made for this test case which will also significantly decrease performance on the speedup. However, as can be seen in the graph above, a significant speedup can be clearly seen and therefore the implementation of DeltaCNN did show its potential for more complex CNN's.

MNIST Dataset	**Delta CNN**	**Regular CNN**	**Delta CNN speedup factor**
Average time	11.9 (s)	13.2 (s)	1.11 (-)

Hyperparameter tuning

It was attempted to do a hyperparameter variation study to test if further speedup could be achieved. As many errors were encountered when adjusting the batch size, it was not possible to perform this study and therefore this is left as a recommendation for further studies. The results presented above are likely to be far from optimal and so the true speedup potential of DeltaCNN even for this small CNN remains yet to be uncovered.

Discussion and Conclusion

For this reproducibility project the goal was to reproduce the results presented in the DeltaCNN paper. Unfortunately, the team encountered challenges with the activation and implementation of the methods presented in the paper. Within the given timeframe and resources, it was not possible to reproduce the table and therefore we had to divert to a different approach. The team managed to get the DeltaCNN package to run on a simple CNN and so the results of the speedup are evaluated. By sorting the MNIST dataset for training and testing a significant speedup became apparent between regular CNN and using the CUDA backend and so the theory presented in the DeltaCNN paper holds. The results are very promising and further evaluation is recommended under the condition that more elaborate documentation of the DeltaCNN package is provided.

References

[1] M. Parger, C. Tang, C. D. Twigg, C. Keskin, R. Wang, and M. Steinberger, “Deltacnn: End-to-end cnn inference of sparse frame differences in videos,” 2022.

[2] Wikipedia.org, URL: https://en.wikipedia.org/wiki/MNIST_database

Chosen project topics:

Reproduced

New data

Task distribution:

Lucas Hofman	Ruben Overwater
Lucas worked on the reproducibility of the results presented in the DeltaCNN paper. In this time Lucas worked through the errors the package produced and focused later on the implementation in a simpler network.	Ruben's expertise was required to setup and install the DeltaCNN packages and net initialization. After that Ruben focused on the implementation of a DeltaCNN layer on a new dataset.