Impact of image compression on CNN performance metrics for CPS nodes at the Arctic Tundra

Steffen Randrup, Issam Raïs, John Markus Bjørndalen, Phuong Hoai Ha, Otto Anshus
Computer science, UiT Tromso, Tromso, Norway
Email: steffen.r.kristensen@uit.no

***Abstract*—In cyber-physical systems (CPS), data can be collected and pre-processed at the edge nodes before being sent to a back-end server. However, for environments like the Arctic tundra, both energy and data networks are constrained. The nodes must save energy to have long operational lifetime from a single battery charge. Consequently, the size of the data to be sent should be reduced as much as possible without losing the information needed to perform the analysis at the back-end.**

This paper evaluates the effects of data size reduction at edge nodes on the analysis performance at a back-end server. The use-case considers edge nodes deployed in the Arctic tundra to take pictures of animals. These images are then transmitted to a back-end server to determine the species of the animals using a convolutional neural network (CNN). We identify 17 functions for reducing the data size of an image series. We run several combinations on a series of images collected on the tundra. Our experimental results show a possible reduction in the number of bytes required to represent the images between 90.2% and 92.1% with decrease in mean precision and mean recall by less than 0.03. Re-scaling images is required to reach large size reductions on the image series.

***Index Terms*—Cyber-physical system, CPS, CNN, edge computing, energy efficiency, tundra, monitoring;**

I. Introduction

A distributed multi-node cyber-physical system can be deployed into a hostile and resource-constrained environment to observe on-ground events. Each node comprises one or multiple battery-powered microcontrollers and computers with storage, multiple data networks, and sensors. In the DAO project[1], we call such a node an Observation Unit (OU).

In this paper, we focus on OUs deployed to the arctic tundra to take pictures of animals. The pictures are sent to a CNN at a back-end for determining the species of the found animals. However, because the OUs have just a single battery charge and need to stay operational for a year or more, they must aggressively save energy. They do this by sleeping most of the time, and by being frugal when they are awake. Because the OUs primarily send the data they collect from the sensors over a data network, smaller data-size provides for a shorter transmission time and therefore less energy spent. In addition, energy consumption is reduced because the nodes can go to sleep again sooner.

We are not reporting on running the CNN in the OUs, which would let the OUs report only the predictions from the CNN to a back-end. There are multiple reasons for transferring images from the Arctic tundra to a back-end. It enables more frequent monitoring by a back-end of the state of the tundra as well as of the node itself. Also, data that has been transferred will not be lost due to a node failing. Finally, having the data at the back-end allows for improving the CNN and for back-end re-analysis.

The amount of data to transfer can be reduced by compressing and filtering the data. However, this can result in some loss of information. The objective is to reduce the size of the data but avoid having a significant impact on the annotations produced by the CNN compared to analyzing the original full-sized data. This paper reports on several functions and many combinations of functions to reduce the data size of images. It also reports on the change in results from the CNN when giving it full-size vs. reduced-size data. We select a number of filter and compression functions and combine them into a number of variable length pipelines. We run the functions and pipelines on the full-sized data to get reduced-size data. We compute the reduction in size achieved by the pipelines as well as the execution time. We then run the CNN on the full-sized and the reduced-size data. We use the CNN predictions to compute the mean precision and mean recall.

The main contributions of this paper are:

- An evaluation of the performance (in time, size reduction, and change in CNN performance) for individual functions and combinations of them in pipelines.
- An evaluation of the functions most commonly present among pipelines yielding high size reduction and good CNN performance.
- Documenting that a significant reduction in the size of images (in bytes) through multiple functions can have insignificant impact on the CNN performance metrics.
- Documenting that the CNN may get low precision and recall on the output from *combinations* of size-reducing functions, even when the precision and recall are high on the output of each of the functions *separately*.
- Documenting that the order of functions is important for both execution time, and CNN performance.

This paper is structured as follows: Section II presents the motivating use-case. Section III describes the CNN and data sets. Section IV presents how we reduce the size in bytes of the images. Section V presents the experiments we run and the metrics used for evaluation. Section VI present the results of the experiments. In Section VII we discuss the results of the experiments.

[1]https://en.uit.no/project/dao

II. Motivating use-case: The Arctic tundra

The arctic tundra is a vast, remote area with a harsh environment, especially with regards to weather and lack of sunlight during the winter [3]. The lack of infrastructure on the tundra limits the amount of ground-based observation sites that realistically can be visited to retrieve data or replace batteries. In certain areas, it may even be impossible to reach the sites for a prolonged duration of time due to weather and other conditions. To scale an on-ground observation system with regards to the number of nodes and coverage, automation of observations and reporting is needed.

The COAT program (which collaborates with DAO) monitors ecosystems on the arctic tundra, which is one of the areas most sensitive to climate change [3]. Monitoring the ecosystems on the tundra includes capturing images of local wildlife to detect their presence and count occurrences of different species. Results from these observations are input to climate and ecological models.

There are several types of camera traps in deployment on the tundra. In this paper, we use images from traps designed to observe small rodents. These are constructed using aluminium boxes with an opening at each end, which lead to a larger space in the middle where the camera is attached. The images used in the experiments are captured in gray-scale using a Reconyx™ SM750 HyperFire™ at 1280×720. A full description of the setup can be found in [6][14]. A typical deployment period is up to 1 year during which the camera trap produces 1500-3000 JPEG images for a total size of 120-250 MB per trap.

On the tundra, data networks are limited, unreliable, and often unavailable for long durations of time. UAVs can in limited cases provide networking for shorter periods for areas without other infrastructure[10]. Using satellite communication does not scale with regards to bandwidth and the number of nodes. For wireless networking, transmitting and receiving are energy costly. Consequently, reduced data size means shorter transmission time and reduced energy use.

III. CNN for identifying species in images

We use a CNN to identify and classify species in the images collected from the camera traps. For this paper, we use a YOLOv3 [12] model implementation from ImageAI [7] trained on images from the COAT camera traps.

For the training set, the images are selected from different camera traps at different locations from deployments in 2018. The training set consists of 1328 images that were classified by a human. It comprises 400 images of voles, 381 images of shrews, 376 images of lemmings, 76 images of birds, and 95 images of stoats. Birds and stoats are found infrequently in the camera traps. They are over-represented in the training set. The validation set contains images of 89 voles, 92 shrews, 103 lemmings, 16 birds, and 19 stoats , which is again selected arbitrarily among the available images from 2018. There is no overlap with the training set. Each of the images contains at most one animal. On the validation set, we get mean precision 0.970 and mean recall 0.921.

For the experiments, we need a typical sequence of images from a camera trap. The selected images are 1896 images from a single camera trap during a 1-year deployment from 2017. We add 255 images of lemmings from two other camera traps for a total of 2151 images. We call this image set the production set. The added images of lemmings are selected from a different area than the production set and therefore from a different camera trap. The production set comprises 371 images with voles, 409 images with shrews, 255 images with lemmings, and 1116 images with no animals.

IV. Reducing the size of transferred images

The collected images can be processed by an OU to reduce data size. The size-reduced images are then sent to the back-end, which runs the CNN. The size of data to transfer can be reduced in several ways.

We identify three types of data reductions used in this paper:

1) Selecting the images to transfer. Images with no animals do not need to be transferred.
2) Reducing the size of individual image files by combining one or more functions (e.g cropping and scaling to produce smaller files to transfer).
3) Aggregation. Similar images can be better compressed if we aggregate them and only save the differences between the images.

Images can be selected by, for instance, detecting changes between subsequent images. An aggregation function uses methods for combining multiple images and compressing the results. An example is to make a small video file from the images. A sequence of functions can be combined into a *pipeline* to provide better aggregated compression. Each *stage* in the pipeline is a chosen function and a parameter that controls the function. An example of a pipeline is image cropping, followed by a re-scale function with a size parameter. Figure 1 illustrates a pipeline with an aggregating function.

Different combinations of functions, as well as the order of functions can potentially influence both the size of the resulting images and the information loss. In this paper, we use the 17 functions detailed in Section V-C, of which 11 accept a parameter value. Each stage of the pipeline is chosen from 87 different function-parameter combinations. The pipelines range from 1 to 4 stages. Running all possible pipelines on the entire dataset is intractable as we would have to run on the order of $6 \cdot 10^7$ pipelines on the dataset, and the CNN on the resulting size-reduced data. It is, therefore, necessary to reduce the number of pipelines.

We assume that running the same function multiple times does not produce a significantly different output than running it once in a pipeline with a different parameter value. For example, re-scaling an image to half pixel count twice produces a similar output to re-scaling to quarter pixel count once. We, therefore, reduce the number of pipelines to explore by choosing to use permutations of functions instead of combinations.

Three of the functions encode the images. When the images are encoded, they must be decoded to arrays of pixel values,

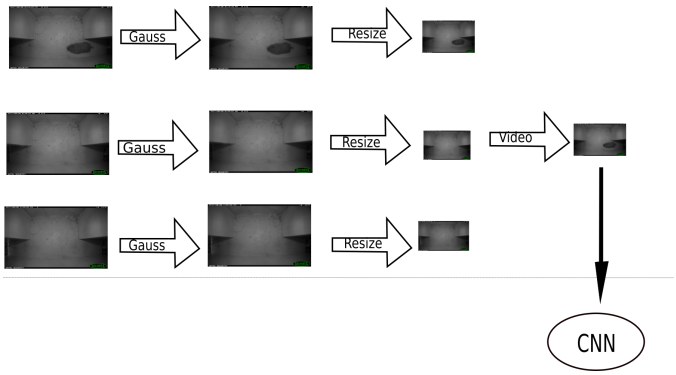

Fig. 1: An example pipeline which includes an aggregating function. Multiple images are processed through the first functions in the pipeline (gauss and resize in this example) before the results are combined using the aggregating function (video). When the images have been processed by all the functions in the pipeline the output may be sent to the remote CNN.

before they can be passed to the next function in a pipeline. This causes a loss of image quality without the benefits to size reduction from the encoding. We, therefore, require that the encoding functions are used as the last stage in each pipeline.

To further reduce the number of pipelines to explore, we introduce three phases to evaluate and discard functions and pipelines. The phases are covered in detail in Section V-B.

V. Experiments and metrics

This section describes the experiments done to reduce the number of candidate pipelines and to evaluate the final candidate pipelines. The metrics used to select pipelines between each phase are described in further detail in Section V-A.

Experiments measuring the execution time are run in a Docker container running on an Intel i7-7700K CPU with 32GB RAM and an NVidia GTX 1080 GPU. The container is based on the nvidia/cuda:10.0-cudnn7-devel image.

A. Metrics

The pipeline performance is defined by four metrics:

Size reduction. The size reduction is reported as the total size in bytes of the output files relative to the input files size. A reported reduction of 90% means that the sum of the sizes of the output files is 10% of the sum of the sizes of the input files.

Execution time of pipeline per image. We report the average execution time per image. To measure the execution time, one timestamp is taken before the first image is processed by the pipeline and another timestamp after the last image is finished. We then divide the elapsed time by the number of images. The execution time of functions and pipelines does not include the time to read or write the images to storage.

Mean precision. To quantify the trade-off in model precision and data size reductions, we measure the mean precision, mP, from running the CNN on the output from a pipeline. For a prediction to be counted as a true positive, its intersection over union, IoU, with the reference annotation must be larger than 0.5. When there are multiple overlapping predictions for the same object as determined by $IoU > 0.5$, we consider the prediction with the highest confidence score to be the only prediction for the object. We report the change between the mean precision on the pipeline outputs and the original image series, ΔmP.

Mean recall. The mean recall, mR, is computed similarly to mean precision. Some functions remove parts of the images or entire images from the output image series. When the functions remove an image with an animal or a region of an image containing an animal, the animal cannot be detected by the CNN. This is counted as a false negative.

B. Experiment phases

The experiments are conducted in three phases for selecting candidate pipelines. The main objective for the first two phases is to reduce the number of pipelines to evaluate in the final phase by discarding potential pipelines that are unlikely to perform well in the final phase. The third phase evaluates the size reduction, and CNN performance of the remaining pipelines.

The *first phase* reduces the solution space by removing functions and possible parameter values. The pair ⟨function, parameter value⟩ is called a *candidate function*. We select candidate functions based on three criteria: Execution time, ΔmP, and ΔmR. For each candidate function we measure the time to process all 2151 images in the production set. Candidates with a significantly higher execution time than other functions are discarded. The threshold is set to 0.1 seconds per image. The output images are then passed directly to the CNN for prediction without saving them to external files. The effect is to evaluate the CNN performance directly from the candidate functions without adding an extra encoding function. ΔmP and ΔmR are then calculated and used to discard candidate functions. Candidate functions with $\Delta mP \leq -0.03$ or $\Delta mR \leq -0.03$ are discarded.

The *second phase* combines the remaining candidate functions into pipelines with up to four functions. The pipelines are evaluated for size reduction and execution time on the first 100 images of the production set. We only run the pipelines on the first 100 images for practical reasons. For each pipeline, we note the size of the output and the time required to process all the images. Pipelines with a size reduction of less than 90% and execution time of more than 0.02 seconds per image are discarded. The requirement for execution time is set based on observation of the results presented in Section VI.

The *third phase* evaluates the mean precision and mean recall of the CNN on the output images from each remaining pipeline as well as the size reduction and execution time on the entire production set.

C. Function description

Below is a description of each of the functions used. The functions are implemented in Python using Numpy and OpenCV.

resize: Re-sizing an image to a lower resolution reduces the number of bytes representing the image. The function is implemented using the resize function in OpenCV. The parameter is a factor for scaling the output dimensions. We use $\{0.1, 0.2, \ldots, 0.8, 0.9\}$.

cropCenter: Cropping removes a fraction of the image from each edge. A parameter of 0.1 keeps the middle 10%, i.e. removing 45% of the image from each side. We use $\{0.1, 0.2, \ldots, 0.8, 0.9\}$.

seamCarving: Seam carving is using genetic algorithms to perform content-aware re-scaling [1]. The parameter value is a real number between 0 and 1 representing the fraction of lines to remove in both horizontal and vertical directions. We use $\{0.1, 0.2, \ldots, 0.8, 0.9\}$. For the implementation we use [15].

fourierTransform: We use the discrete Fourier transform function *dft* in OpenCV on a gray-scale version of the image. We select low-frequency components, effectively removing details from the image while preserving large-scale features. We use $\{0.1, 0.2, \ldots, 0.8, 0.9\}$ for the parameter value.

blackWhite: Gray-scale conversion lowers the color information from 3 channels to 1. The grayscale conversion is done by OpenCV.

JPEG: Encodes the images in the JPEG format. The quality setting of the JPEG encoding determines the level of compression. We use the quality settings $\{10, 20, \ldots, 90\}$.

keepEveryNPixels: Keeps every n'th pixel of the image in width and height. The parameter is an integer for which pixels should be kept: Every 2, every 3, etc. We use $\{2, 3, \ldots, 9\}$.

removeHigh: The image is modified to keep all pixel values below the selected threshold and set the values above to the threshold value. We use $\{128, 160, 192, 224\}$.

removeLow: Like removeHigh, but uses a lower bound on pixel values. Values below the threshold are set to zero. As a lower bound we set $\{32, 64, 92, 128\}$.

halfpixels: Divides every pixel value by two, rounding down. The pixel values are multiplied by two before prediction by the CNN. No parameters.

gauss: Applies a Gaussian Blur using the GaussingBlur function in OpenCV. We use a kernel of size $\{3 \times 3, 5 \times 5, 7 \times 7, 9 \times 9\}$.

hidePics: Hide 6 gray-scale images in one color image, keeping only the most important bits of six images allows hiding another image in the lower bits, thereby storing more data in one image.

mergeImages: Select n subsequent images from the series. For each pixel coordinate (x, y, channel) calculate the mean pixel value from each of the images. Select the pixel value with the largest absolute difference from the mean. We use $n = \{3, 4, \ldots, 9\}$

HEIF: Stores the image files in the High Efficiency Image File Format [2]. We first store the JPEG encoded files with quality setting 100. The images are then converted to HEIF using heif-enc version 1.6.1. We use $\{10, 20, ..., 90\}$ for the HEIF quality setting.

changeDetection1: This function selects rectangular areas bounding the regions where change is detected. First, a Gaussian blur with a 25×25 kernel is applied to the latest image. Then the difference between the image and a running average of previous images is calculated. Contours in the difference are then detected. If the area covered by a contour is larger than a threshold value, it is registered as a detected change.

changeDetection2: Like changeDetection1 but if change is detected, the entire image is selected.

videoFromImages: Each image is inserted as a frame in an H.264 encoded video.

VI. Results

In this section, we present the results from each of the phases described in Section V. In practice, the trained model does not perform as well as expected from the validation set. We get $mR = 0.851$ and $mP = 0.844$ on the production set.

A. Phase 1: Discarding candidate functions and parameters

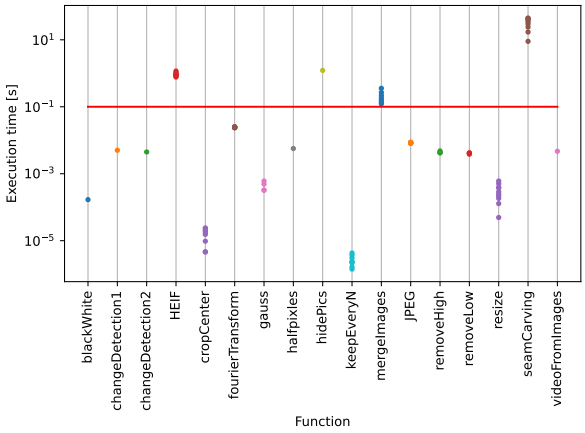

Fig. 2: Execution time per image for each candidate function on the images in the production set. Each of the used parameter values are plotted as separate dots. Candidate functions with execution time higher than 0.1 seconds per image are discarded. The red line marks the cutoff point.

The measured execution time per image for each candidate function is shown in Figure 2. The differences in execution time span several orders of magnitude. The fastest candidate functions *cropCenter* and *keepEveryNPixels* select pixels from the original image without other modifications. We discard the following candidate functions that have an execution time longer than 0.1 seconds per image from further consideration: *seamCarving*, *HEIF*, *hidePics*, and *mergeImages*. The cutoff point of 0.1 seconds per image is selected because there appears to be a gap in execution time of more than an order of magnitude between functions above and below the cutoff point. There are still significant differences in execution time of the remaining candidate functions, which may impact the execution time of pipelines. This is addressed in phase 2.

Figure 3 shows the change in CNN performance on the output from the candidate functions. The worst performers, in the lower-left corner of the figure, are *cropCenter* and

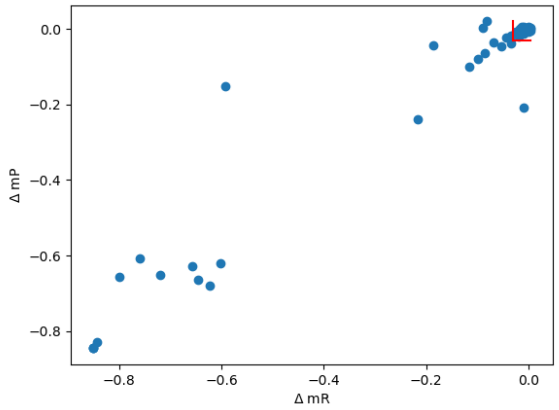

Fig. 3: CNN performance metrics on output from the candidate functions. The red lines indicate the selection criteria of $\Delta mP > -0.03$ and $\Delta mR > -0.03$

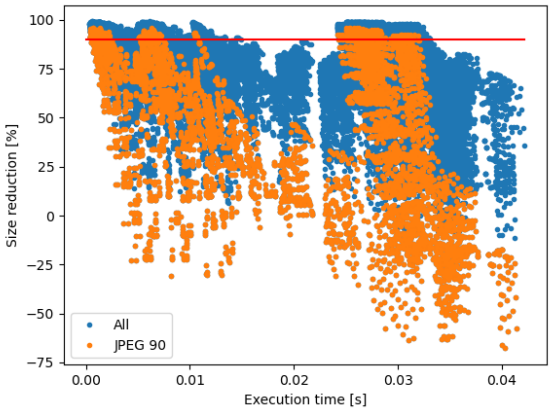

Fig. 4: Execution time and size reduction for each pipeline in phase 2. The pipelines are executed on the first 100 images of the production set. The red line shows a size reduction of 90%.

removeLow. The *cropCenter* function removes parts of the images, which could have contained animals. The *removeLow* function sets many of the colors to completely black, thereby removing animals from the images. We discard candidate functions with $\Delta mP \leq -0.03$ and $\Delta mR \leq -0.03$ to further decrease the number of candidate functions to include in the next phases. The accepted functions and parameter values for phase 2 are shown in Table I.

Function	Parameter values	Encoding
JPEG	30, 40, 50, 60, 70, 80, 90	✓
videoFromImages	No parameter	✓
blackWhite	No parameter	
resize	0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9	
halfpixels	No parameter	
gauss	3, 5	
keepEveryNPixels	2, 3, 4, 5, 6	
removeHigh	128, 160, 192, 224	
fourierTransform	0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9	

TABLE I: Remaining candidate functions after phase 1. The encoding functions produce image or video files. The other functions produce arrays of pixel values.

B. Phase 2: Discarding pipelines

The functions *resize* and *keepEveryNPixels* both re-scale the images but with different implementations. Because of the similarity between the functions, we discard pipelines that include both functions. We create 60736 pipelines of up to four stages comprised of the candidate functions remaining after phase 1.

Figure 4 shows the execution time and size reduction for each pipeline on the first 100 images of the production set. Some pipelines produce output with a larger size than the JPEG-encoded input files. Figure 4 highlights pipelines where *JPEG 90* is used. This explains the majority of the pipelines with negative size reduction.

Since we are primarily interested in pipelines with large size reductions, we discard pipelines that reduce the input by less than 90%. This reduces the number of pipelines to 23427.

We further reduce the number of pipelines by comparing similar pipelines. If a pipeline extends an existing pipeline by adding stages, it should improve its size reductions. Otherwise, it will be discarded. For example, the pipeline *blackWhite* → *gauss 3* → *resize 0.5* → *JPEG 80* must have output with data size smaller than both *blackWhite* → *gauss 3* → *JPEG 80*, and *blackWhite* → *JPEG 80*. With this requirement, the number of pipelines is reduced to 20755.

Re-scaling influences execution time of the functions following in a pipeline. Figure 5 highlights pipelines where *resize* is the first function. When *resize* is the first function, the execution time is shorter for a pipeline than when it is not. A similar pattern is observed for the *keepEveryNPixels* function.

There is a clear distinction between two groups of pipelines. The group with longer execution time is explained by running the slowest candidate function, *fourierTransform*, before re-scaling. We discard pipelines with an execution time longer than 0.02 seconds per image. The remaining number of pipelines for evaluation on the CNN is reduced to 12527.

All of the 12527 remaining pipelines include either *resize* or *keepEveryNPixels*. The survival of the pipelines depends on how much the two re-scaling functions reduce the number of pixels in each image. Smaller output images generally lead to a higher number of surviving pipelines for both functions.

The last stage of each pipeline is an encoding function. Using a low quality setting for the JPEG encoding gives larger size reduction and thereby more selections. Whether a pipeline is discarded or not, is primarily determined by the parameter for the re-scale function and the choice of encoding function and its parameter. While the other candidate functions contribute to reducing the output sizes, they are not the primary reason.

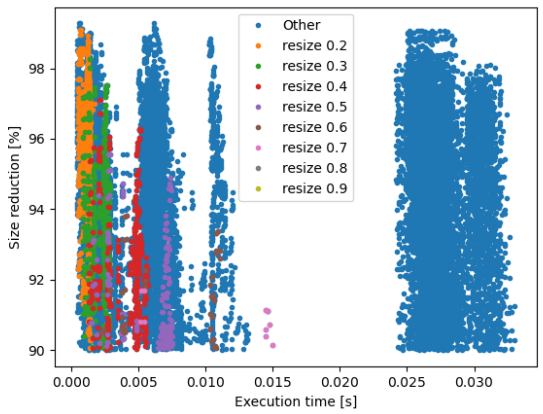

Fig. 5: Pipelines with more than 90% size reduction. Pipelines where *resize* is the first function are highlighted. The group of pipelines with high execution time is explained by running *fourierTransform* before re-scaling.

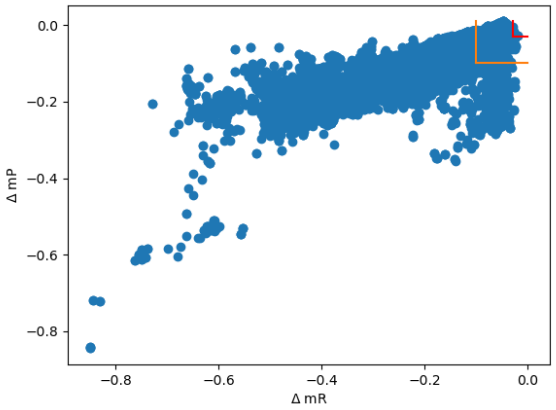

Fig. 6: The change in mean recall and mean precision for the output of each pipeline compared to the original image series. Red: $\Delta mP > -0.03, \Delta mR > -0.03$, Orange: $\Delta mP > -0.1, \Delta mR > -0.1$

C. Phase 3: Evaluating CNN impact from pipelines

The images from the camera traps have a Reconyx™ logo in the lower right area. Making predictions on the logo is a possible error, which we in practice can ignore because we know the location of the logo in every image. When we evaluate the performance of the CNN, we disregard predictions overlapping this logo.

The change in mean recall and mean precision compared to the original image series is shown in Figure 6. In phase 1 we required, for each candidate function, that the mean precision and mean recall does not decrease by more than 0.03. Only 12 pipelines meet that requirement. Therefore we also show pipelines, where mean precision and mean recall do not decrease by more than 0.1.

The relationship between size reduction and ΔmR for the pipelines with $\Delta mP > -0.1$ and $\Delta mR > -0.1$ is shown in Figure 7. In Figure 7 we see that a larger size reduction comes at the cost of worse CNN performance. The pipelines meeting criteria $\Delta mP > -0.03$ and $\Delta mR > -0.03$ for the CNN performance yield a size reduction between 90.2% and 92.1%. If we allow a decrease in mR and mP of -0.1, the size reductions range from 89.8% to 97.3%.

The pipelines meeting the criteria $\Delta mP > -0.03$ and $\Delta mR > -0.03$ are shown in Table II. The largest size reduction is reached by the pipeline *blackWhite* → *removeHigh128* → *resize 0.5* → *videoFromImages*. It is the only of the remaining pipelines which include the *videoFromImages* encoding function. The size reduction from the pipeline with *videoFromImages* is noticeably larger than the other pipelines, but the change in mean precision and mean recall is comparable to most of the other pipelines. There are several other pipelines containing the same four candidate functions in a different order, which fall just short of the requirements for either ΔmP or ΔmR. They give comparable size reduction. The pipeline *blackWhite* → *removeHigh128* → *keepEveryNPixels* → *videoFromImages* where *resize 0.5* is replaced by *keepEveryNPixels 2* was not selected from phase 2, because the size reduction did not meet the requirement. There are, however, other pipelines comprising *keepEveryNPixels 2* and *videoFromImages* along with one or two other candidate functions, which show results comparable to the selected pipeline. Although, they do not meet the requirements for CNN performance.

The smallest change in mean precision is from the pipeline *blackWhite* → *resize 0.4* → *fourierTransform 0.5* → *JPEG 40*. However, it has the largest change in mean recall. The smallest change in mean recall is from the two pipelines *keepEveryNPixels 3* → *removeHigh 192* → *fourierTransform 0.8* → *JPEG 50* and *removeHigh 192* → *keepEveryNPixels 3* → *fourierTransform 0.8* → *JPEG 50*. Those two pipelines produce exactly the same output images and therefore get the exact same CNN predictions. The pipeline beginning with *keepEveryNPixels* has a shorter execution time. However, the ten pipelines with *keepEveryNPixels* have markedly higher execution times than the other pipelines in the table. The difference is not because the *keepEveryNPixels* function is slower than *resize*. In fact, we observed it to be faster in phase 1. Rather, the difference is caused by *fourierTransform*, because *keepEveryNPixels 3* re-scales to an odd-sized resolution. The dft in *fourierTransform* operates faster on even-sized resolutions than odd-sized.

11 of the 12 pipelines include the *fourierTransform* candidate function. There are pipelines which include the same functions, but leave out the fourierTransform. They have noticeably lower execution time but fall short on some of the other requirements. We also notice from the remaining pipelines, that the lower the resolution of the output images is, the higher the quality setting of the encoding function. Since this pattern is present, it may be possible, we can create pipelines meeting our performance criteria entirely by the selection of output resolution and encoding function.

TABLE II: Pipelines with $\Delta mP > -0.03$, $\Delta mR > -0.03$ evaluated on the 2151 images in the production set. Execution time is per image.

Pipeline	Size reduction	ΔmP	ΔmR	Execution time [ms]
blackWhite → removeHigh128 → resize 0.5 → videoFromImages	**92.1%**	-0.025	-0.026	**2.88**
blackWhite → resize 0.4 → fourierTransform 0.5 → JPEG 40	90.8%	**-0.005**	-0.030	4.14
keepEveryNPixels 3 → removeHigh 192 → fourierTransform 0.6 → JPEG 50	90.7%	-0.029	-0.029	7.18
keepEveryNPixels 3 → removeHigh 192 → fourierTransform 0.7 → JPEG 50	90.8%	-0.025	-0.028	7.18
keepEveryNPixels 3 → removeHigh 192 → fourierTransform 0.8 → JPEG 50	90.6%	-0.027	**-0.019**	7.06
keepEveryNPixels 3 → removeHigh 224 → fourierTransform 0.6 → JPEG 60	90.2%	-0.021	-0.028	7.05
keepEveryNPixels 3 → removeHigh 224 → fourierTransform 0.7 → JPEG 60	90.2%	-0.016	-0.027	7.05
removeHigh 192 → keepEveryNPixels 3 → fourierTransform 0.6 → JPEG 50	90.7%	-0.029	-0.029	10.5
removeHigh 192 → keepEveryNPixels 3 → fourierTransform 0.7 → JPEG 50	90.8%	-0.025	-0.028	10.5
removeHigh 192 → keepEveryNPixels 3 → fourierTransform 0.8 → JPEG 50	90.6%	-0.027	**-0.019**	10.3
removeHigh 224 → keepEveryNPixels 3 → fourierTransform 0.6 → JPEG 60	90.2%	-0.021	-0.028	10.2
removeHigh 224 → keepEveryNPixels 3 → fourierTransform 0.7 → JPEG 60	90.2%	-0.016	-0.027	10.2

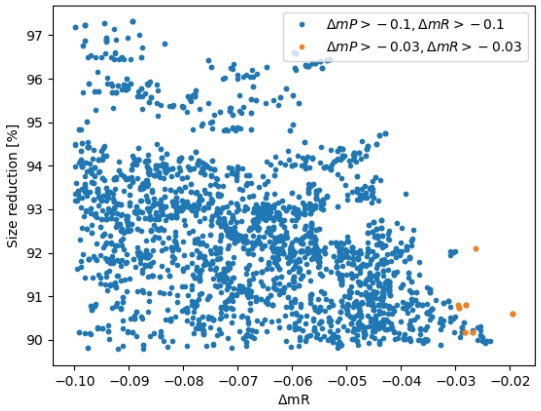

Fig. 7: Size reduction for varying ΔmR. $\Delta mP > -0.1$, $\Delta mR > -0.1$

VII. Discussion

The added images of lemmings are from two other camera traps with slightly different interiors. We only include the images of lemmings. This means the differences between those images are larger than between the first 1896 images. Both *changeDetection2* and *changeDetection1*, which exploit differences between images, will therefore detect more changes. This should improve ΔmR, because fewer images would have been discarded. Still, too many images containing animals are discarded.

The *HEIF* function uses an external tool, which requires encoding to a JPEG file which is then read and re-encoded as a HEIF image. This adds execution time and some information loss in the image. Furthermore, we have not explored using HEIF to store multiple images in one file similar to video encoding. When HEIF-encoding is available in OpenCV, we may re-evaluate the function.

The images from the cameras are already grayscale except for the Reconyx™ watermark. Therefore applying *blackWhite* does not change the image contents outside the watermark. If the camera traps had captured color images, we would expect a higher impact on file sizes. There could also be an impact on CNN performance if color information is necessary for the classification.

The *removeHigh* function does not modify the image contents significantly. There are no white animals and only a few images with snow or other white features, which could have been removed by the function. The camera adds metadata to the images in the form of a black bar with white text. The white color of the text is modified by the *removeHigh* function. Without the white text, many of the images would not have been changed by the *removeHigh* function. While *blackWhite* and *removeHigh* allow a slightly higher size reduction on our images, they may not work as well for bright or colored image series.

Re-scaling to half width and height corresponds to a decrease in the number of pixels by 75%. In combination with encoding, a size reduction around 90% makes sense. When using JPEG encoding, the quality setting needs to be reduced to reach the 90% size reduction.

For video encoding to exploit similarities between images, we need several images. We find that 10-25 images per video file is sufficient to reach the largest size reductions from video encoding. The actual size reductions vary depending on the difference between images. We do observe a smaller size reduction from pipelines with the video encoding in phase 2 than in phase 3. In phase 2, we only used the first 100 images of the image series. At the time those images were captured, it was possible for light to enter the box. Therefore there is a change of brightness between the images near the entrances to the box. This may have the effect of reducing the size reduction from the *videoFromImages* function. It is therefore possible, we have discarded pipelines with *videoFromImages* in phase 2, which we would otherwise not have discarded. It is however realistic and not uncommon for light to enter the boxes. Generally, choosing cutoff points in each of the three phases means, we are discarding functions or pipelines, which only barely miss the criteria.

VIII. Related work

In this paper, we follow up on the work in [11], which considers the effects of image re-sizing on energy consumption and CNN confidence score but does not explore other ways

of modifying images. The size in bytes of a series of images can be changed in several ways. In [8] k-means clustering is used to reduce the color space of images. This reduces file sizes and the cost of sending the images to a back-end. They also found that the cost of running the k-means learning on edge nodes may consume more energy from processing the images than what can be saved due to the lower number of bytes for transmission. The work in [5] explored the energy cost of running JPEG compression at different quality setting as well as exploiting differences between images to encode successive images. We apply the considerations for execution time when discarding and evaluating pipelines. In [8] and [5] image quality was evaluated on the mean squared error and peak signal to noise ratio. We focus on CNN performance for evaluation of image quality instead. Similarities between multiple overlapping images from different cameras are calculated in [4] to reduce image size. We only use images from a single camera in a trap, but we do attempt functions, which exploit similarities between images.

In [9] the effects of image degradation on CNN-based classification is described. They compared different types of degradation such as low resolution, motion blur, and Gaussian blur on both real and synthetic image sets. A similar experiment is conducted in [13], which also considered the effect of JPEG encoding. The change in performance for Gaussian blur and decreasing JPEG quality appear to be consistent with our experiments. Neither [9] nor [13] combined multiple image modifications in their evaluations. We consider multiple modifications to the image series as well as the order, in which they are applied. When multiple modifications are applied to a series of images, the CNN performance is altered differently than when only considering single modifications.

IX. Conclusion

The Distributed Artic Observatory (DAO) researches cyber-physical systems on the Artic Tundra. Connectivity and bandwidth are limited in the area, restricting how much data can be transferred. This paper explores compressing and filtering images to conserve bandwidth when transferring images from camera traps to backend computers for later CNN analytics.

We explore how pipelines constructed from 17 functions change data size of an image series from the Arctic tundra, and how the performance of a CNN on the output is affected by the results of each pipeline. We measure the reduction in the number of bytes required to represent the image series (size reduction), the execution time of the pipelines, the change in mean precision (ΔmP), and the change in mean recall (ΔmR) compared to doing nothing to the images.

From a solution space of $6 \cdot 10^7$ pipelines, we reduce the number of pipelines in three phases and evaluate the resulting pipelines. We find 12 pipelines with $\Delta mP > -0.03, \Delta mR > -0.03$, execution time between $2.88 \cdot 10^{-3}$ and $10.5 \cdot 10^{-3}$ seconds per image, and size reduction between 90.2% and 92.1%.

We find that re-scaling and the choice of encoding contribute the most to size reduction. Other functions either do not contribute much to size reduction, or reduce CNN performance too much.

The best pipelines should in the future be evaluated for the combined energy consumption when running the pipeline and sending the output images.

Acknowledgment

The DAO project is supported by the Research Council of Norway (RCN) IKTPluss program, project number 270672. Thank you very much to the COAT ecologists, UiT.

References

[1] Shai Avidan and Ariel Shamir. "Seam carving for content-aware image resizing". In: 2007. DOI: 10.1145/1275808.1276390.

[2] M. M. Hannuksela, J. Lainema, and V. K. Malamal Vadakital. "The High Efficiency Image File Format Standard [Standards in a Nutshell]". In: (2015).

[3] R.A. Ims et al. *Science plan for COAT: Climate-ecological Observatory for Arctic Tundra*. 2013.

[4] Maryam Asadzadeh Kaljahi et al. "A new image size reduction model for an efficient visual sensor network". In: (2019). DOI: 10.1016/j.jvcir.2019.102573.

[5] Dong-U Lee et al. "Energy-Efficient Image Compression for Resource-Constrained Platforms". In: (2009). DOI: 10.1109/TIP.2009.2022438.

[6] Eivind Flittie Kleiven Eeva M. Soininen John Markus Bjørndalen Otto Anshus Michael J. Murphy Øystein Tveito. "Experiences Building and Deploying Wireless Sensor Nodes for the Arctic Tundra". In: (2021).

[7] Moses Olafenwa and John Olafenwa. *ImageAI*. Mar. 2018. URL: https://github.com/OlafenwaMoses/ImageAI.

[8] J. Paek and J. Ko. "K-Means Clustering-Based Data Compression Scheme for Wireless Imaging Sensor Networks". In: (2017). DOI: 10.1109/JSYST.2015.2491359.

[9] Yanting Pei et al. "Effects of Image Degradation and Degradation Removal to CNN-based Image Classification". In: (2019). DOI: 10.1109/tpami.2019.2950923.

[10] I. Raïs et al. "UAVs as a Leverage to Provide Energy and Network for Cyber-Physical Observation Units on the Arctic Tundra". In: 2019. DOI: 10.1109/DCOSS.2019.00114.

[11] Issam Raïs, John Markus Bjørndalen, and Otto Ansus. "Trading Data Size and CNN Confidence Score for Energy Efficient CPS Node Communications". In: (2019).

[12] Joseph Redmon and Ali Farhadi. "YOLOv3: An Incremental Improvement". In: (Apr. 8, 2018).

[13] Prasun Roy et al. *Effects of Degradations on Deep Neural Network Architectures*. 2018.

[14] Eeva M. Soininen et al. "Under the snow: a new camera trap opens the white box of subnivean ecology". In: (2015). DOI: 10.1002/rse2.2.

[15] Spellstaker. *Seam Carving*. 2018. URL: https://github.com/Spellstaker/Seam-Carving.