Received August 15, 2019, accepted September 16, 2019, date of publication September 19, 2019,
date of current version October 2, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2942441

High Performance Neural Network Inference, Streaming, and Visualization of Medical Images Using FAST

ERIK SMISTAD[1,2], ANDREAS ØSTVIK[1,2], AND ANDRÉ PEDERSEN[1]
[1]SINTEF Medical Technology, 7465 Trondheim, Norway
[2]Department of Circulation and Medical Imaging, Norwegian University of Science and Technology, 7491 Trondheim, Norway

Corresponding author: Erik Smistad (ersmistad@gmail.com)

This work was supported by the Research Council of Norway under Grant 270941 and Grant 237887.

ABSTRACT Deep convolutional neural networks have quickly become the standard for medical image analysis. Although there are many frameworks focusing on training neural networks, there are few that focus on high performance inference and visualization of medical images. Neural network inference requires an inference engine (IE), and there are currently several IEs available including Intel's OpenVINO, NVIDIA's TensorRT, and Google's TensorFlow which supports multiple backends, including NVIDIA's cuDNN, AMD's ROCm and Intel's MKL-DNN. These IEs only work on specific processors and have completely different application programming interfaces (APIs). In this paper, we presents methods for extending FAST, an open-source high performance framework for medical imaging, to use any IE with a common programming interface. Thereby making it easier for users to deploy and test their neural networks on different processors. This article provides an overview of current IEs and how they can be combined with existing software such as FAST. The methods are demonstrated and evaluated on three performance demanding medical use cases: real-time ultrasound image segmentation, computed tomography (CT) volume segmentation, and patch-wise classification of whole slide microscopy images. Runtime performance was measured on the three use cases with several different IEs and processors. This revealed that the choice of IE and processor can affect performance of medical neural network image analysis considerably. In the most extreme case of processing 171 ultrasound frames, the difference between the fastest and slowest configuration were half a second vs. 24 seconds. For volume processing, using the CPU or the GPU, showed a difference of 2 vs. 53 seconds, and for processing an whole slide microscopy image, the difference was 81 seconds vs. almost 16 minutes. Source code, binary releases and test data can be found online on GitHub at https://github.com/smistad/FAST/.

INDEX TERMS Deep learning, inference, neural networks, medical imaging, high performance.

I. INTRODUCTION
Deep convolutional neural networks (NNs) have shown great results for several medical image processing tasks, such as image classification, segmentation, image-to-image transformation, reconstruction and registration [1]. Although there are many libraries for training NNs, there are few that focus on high performance inference for medical imaging. There is a need for easy-to-use, high performance medical image computing frameworks which support NN inference as well as medical image formats and visualizations. NN training is primarily done using the Python programming language, which is not well suited for large-scale software development, deployment and real-time processing. Currently, one of the most popular NN frameworks is Google's TensorFlow [2]. Although TensorFlow is primarily written in C++, their main focus is on the Python API. Therefore the TensorFlow C++ API can be difficult to use due to its complexity and lack of examples. Furthermore, processor companies such as AMD, Intel and NVIDIA are making their own high performance inference libraries, called *inference engines* (IEs), which can accelerate NN inference on specific processors. Each of these

The associate editor coordinating the review of this manuscript and approving it for publication was Chenguang Yang.

IEs are developed in C++ using completely different APIs, which makes it difficult to test and deploy to all platforms.

FAST (Framework for Heterogeneous Medical Image Computing and Visualization) was proposed in Smistad [3] as an open-source cross-platform framework with the main goal of making it easier to do efficient processing and visualization of medical images on heterogeneous systems. *Heterogeneous systems* are machines with multiple processors for different purposes such as CPUs, integrated GPUs, dedicated GPUs and vision processing units (VPUs). FAST has been used successfully in several projects such as ultrasound-guided regional anesthesia assistant [4], [5], automatic echocardiography image analysis [6], [7] and the ultrasound robot framework EchoBot [8]. In these projects, FAST is used to create complete software which can handle all necessary steps: data streaming/import/export, processing, visualization, and graphical user interface.

This article is intended to give the reader an overview of NN IEs and present methods for combining them with existing software for medical image processing and visualization such as FAST. Three different medical use cases are investigated with high performance in focus:

Case 1 - Real-Time Ultrasound Image Segmentation: Ultrasound is a real-time imaging modality which can provide a stream of many images per second. In this case, real-time data streaming, inference and visualization is often essential and very demanding in terms of implementation. Medical ultrasound images are typically grayscale, noisy and have a small size of less than 1000×1000 pixels. Most medical ultrasound images are in 2D, but 3D is also possible.

Case 2 - CT Volume Segmentation: The typical size of CT and MRI images is about $512 \times 512 \times 512$ grayscale pixels. This large size means that most processors do not have enough memory available for a neural network to process the entire volume at once. Therefore it is necessary to split the volume into subvolumes, often called *patches*, then perform inference on every patch independently, and finally stitch the results back together.

Case 3 - Patch-Wise Classification of a Whole Slide Image: Whole slide microscopy images (WSIs) used in digital pathology are often extremely large. A typical $40\times$ WSI has approximately $200k \times 100k$ color pixels resulting in an uncompressed size of $\sim$ 56 gigabytes [9]. As these images often exceed the amount of memory available on a machine, virtual memory is needed to load, process and visualize these images. As with case 2, this limits processing to one patch at a time.

A. RELATED WORK

As mentioned, there are many libraries for training NNs but few that focus on high performance inference for medical imaging. Compared to other computer vision and image analysis tasks, medical imaging have many different modalities which are often stored in unconventional image formats such as DICOM, metaimage and pyramidal tiled TIFFs. Medical imaging libraries also need to be able to process and visualize 3D images which are common in medicine. And since these 3D images and WSIs can be very large, medical imaging libraries require processor and memory efficient analysis and visualization. In this section, a few relevant frameworks are described and compared to the proposed methods with FAST.

The popular open-source computer vision library **OpenCV** has NN inference support in the latest major release (version 4) [10]. The primary inference engine used in OpenCV is the OpenVINO toolkit by Intel [11]. Unlike FAST, OpenCV does not focus on medical imaging, but computer vision in general, and therefore lack support for specific medical image formats such as DICOM, metaimage and WSI formats. OpenCV is primarily made for 2D images, and therefore need additional steps for processing 3D images while FAST has generic support for 2D and 3D images. While OpenCV has support for some GPU accelerated algorithms as an optional module, FAST uses parallel and GPU computing at its core, thereby removing the need for explicitly moving data back and forth to the GPU. The benefits of OpenCV are its large number of features, user base and good Python bindings.

NVIDIA has launched a collection of healthcare AI tools called **NVIDIA Clara AI** which currently focus on medical imaging and genomics [12]. Comparing Clara to FAST show that most NVIDIA tools are not open-source and only work on NVIDIA GPUs. While Clara has tools for importing medical image data and visualization, these tools are currently provided through third party software such as MITK [13], while FAST provides all of this in one software package. Clara uses TensorRT for inference, which is also supported in FAST.

DeepInfer is an open-source deep learning toolkit specifically designed for medical data analysis [14]. This toolkit is basically a collection of Docker containers with models, libraries and code needed to run each model. DeepInfer also includes a plugin to the popular medical processing and visualisation software Slicer, enabling users to run these Docker containers from a graphical user interface and view the results in Slicer. With this approach, DeepInfer can in theory, like FAST, use any IE. The downside with such an approach is performance loss due to multiple data transfers between Slicer making for instance real-time ultrasound processing and visualization challenging. It also requires users to make Docker images, while FAST only requires the model to be exported to a single file. Another downside with DeepInfer, is code fragmentation, as the visualization and data loading happens in Slicer, while inference happens in the Docker container. With FAST everything happens in one software package. The benefit with DeepInfer is that by using Docker, different models can use different libraries and driver versions on the same system. Also, to our knowledge, Slicer does not support processing and visualization of whole slide microscopy images.

B. CONTRIBUTIONS

The main contributions of this article is an optimized neural network inference module for FAST including:

- A generic plugin system enabling any inference engine to be used and loaded at runtime. The complex C++ code of each inference engine is hidden from the user by providing a common interface.
- Support for several different IEs: Intel's OpenVINO, NVIDIA's TensorRT and Google's TensorFlow with CPU, CUDA and ROCm backends.
- Highly efficient pipelines using parallel and GPU computing enabling real-time streaming and NN inference.
- Performance comparison on three use cases, using different medical data (ultrasound, CT and WSI), IEs and processors.
- Easy to set up pipelines: only a few lines of code are needed as shown in the Appendix.
- Tensor data structures in FAST allowing processing of any N-dimensional data.
- Support for complex NNs with multiple inputs and/or outputs, batch and sequence processing, and patch generation and stitching.
- Concurrent GPU based visualization of classification and segmentation.

II. METHODS

This section starts with describing how FAST, neural networks and IEs work and how they were combined. Finally, the data and the pipelines used for each use case are described.

A. FAST

FAST uses a demand-driven execution pipeline similar to the visualization toolkit (VTK) and the insight toolkit (ITK). This means that each process step is contained in a *process object* (PO). Each PO can have an arbitrary number of input and output data ports. An output port can be connected to another PO's input port, creating a data channel in which data can flow between POs. Several POs can be connected to form complex pipelines such as shown in figures 2, 3 and 4. The pipelines are executed on demand, while doing so, each PO is executed in order.

Pipelines are only executed when a PO has new input data, or a parameter has changed. Due to this generic setup, both static and streamed data can be processed without any code changes. Thus, the same pipeline can be used to process a stream of ultrasound images coming from a scanner as a single ultrasound image stored on disk. Such data streams can be processed using only the latest frame, potentially accepting data loss, which is often desirable in real-time scenarios were the goal is to always process the latest frame. Data streams can also be processed for every frame, potentially creating a queue which needs synchronization due to the producer-consumer problem, which is solved in FAST using multi-threading and semaphores.

Data, such as images and geometric meshes, are represented by abstract data objects which handle both organization and synchronization of this data in different memory such as CPU and GPU RAM. Most processing and visualization in FAST are implemented to run on the GPU for high performance using the libraries Open Computing Library (OpenCL) and Open Graphics Library (OpenGL).

A typical FAST pipeline consists of data sources, processing steps, and data sinks. Data sources can be images stored on disk, or streamed images from a ultrasound scanner or camera. Processing steps can be anything including image filters, segmentation, registration and neural network inference. Data sinks are typically renderers which visualize data on screen, or exporters which store data on disk.

B. NEURAL NETWORKS

Modern neural networks typically use an N-dimensional data structure called a *tensor*. A scalar is a 0D tensor, a list of numbers is a 1D tensor, an image a 2D tensor, a volume a 3D tensor and so on. The number of dimensions and the size of each dimension is referred to as the *shape* of the tensor. In general, neural networks operate on input tensors of a fixed size. A neural network can process a set of samples at a time, namely a *batch*. When performing inference on trained neural network, the batch size is usually 1.

A neural network which needs an 2D image as input, requires a 4D tensor as input, where the four dimensions are sample, y, x and channel in that order. Thus the shape of the tensor for a single color image of size $W \times H$ would be $(1, H, W, 3)$. In this case, the last dimension is the number of channels, this is called *channel last* (CL) ordering. Some IEs require the channel dimension right after the batch dimension instead, which is called *channel first* (CF) ordering. With CF ordering, the tensor shape of the color image would be $(1, 3, H, W)$. For 3D images of size $W \times H \times D$, another dimension is simply added to the data tensor resulting in a shape $(1, D, H, W, C)$.

Feedforward neural networks consist of a set of layers which are connected together to form a graph. Each layer usually have several hyperparameters set by the user, and a large set of weights which are learned during the training process. During execution, the layers are processed in order with input data from the previous layer. The layer setup is defined before training. After training, a neural network model consisting of the layer setup and the learned weights are produced. This model can be stored in one or more files on disk for later use.

C. INFERENCE ENGINES

In this work, five IEs are considered: OpenVINO, TensorRT and TensorFlow with CPU, CUDA and ROCm backends. The API of each IE is different, but the following is assumed to be common for all:

1) A trained NN can be loaded from a file stored on disk.
2) A NN has one or more input and output nodes represented as N-dimensional tensors.
3) A NN can be executed on demand by providing a NN definition and input tensors. Data from output nodes can be read after execution.

TABLE 1. An overview of the inference engines investigated, highlighting some of their major differences. The application footprint refers to the size of binaries and the number of dependencies.

Inference engine	TensorFlow 1.14	OpenVINO 2019 R1	TensorRT 5.1
Dimension ordering	CL default, CF optional	CF	CF
Input/output node detection	Only input	Yes	No
Input/output shape detection	Yes	Yes	No
Processors supported	CPUs, NVIDIA GPUs (cuDNN) and AMD GPUs (ROCm)	CPUs and Intel GPUs, VPUs, FPGAs	NVIDIA GPUs
Open-source (license)	Yes (Apache 2.0)	Yes (Apache 2.0)	No
Application footprint	High	Low	Low
Implemented standard layers	All	Many, 3D not supported.	Few, 3D not supported.
Storage file formats	Google Protobuf (.pb)	Intel Intermediate Representation (.xml + .bin)	Caffe (txt + protobuf), ONNX and UFF

What varies between the IE APIs are 1) the tensor dimension ordering (channel last or channel first), 2) whether input and output nodes are detected automatically or if they have to be set manually by the user, 3) whether the shape of the input and output nodes are detected automatically or if they have to be set manually by the user, 4) which processors the IE can execute on, 5) which layers and file formats are supported and 6) if it is open source or not. The differences for each investigated IE are listed in Table 1. To hide the differences and complexity of each IE, an abstract interface for IEs was created in FAST. This interface was implemented for each of the five IEs used here, but in general any new IE could potentially be added to FAST using this interface.

The next sections describe each of the IEs in more detail.

1) TENSORFLOW

Google's TensorFlow [2], together with Keras, is perhaps the most popular machine learning framework at the moment. This framework is open-source and has several thousand contributors on GitHub. The main target of the framework is to facilitate training in Python, however, the framework also has a C++ API. TensorFlow has implementations of most, if not all, neural network layers and operations. Due to large number of features in TensorFlow, its application footprint is quite large with many third-party dependencies which can be a downside when developing a production ready application.

High performance training and inference in TensorFlow is normally achieved by NVIDIA's proprietary CUDA and cuDNN frameworks which only works with GPUs from NVIDIA. TensorFlow does also support normal CPU execution, CPU execution with Intel's Math Kernel Library for Deep Neural Networks (MKL-DNN) [15], SYCL using the ComputeCpp compiler from Codeplay, NVIDIA's TensorRT and also recently support for AMD CPUs and GPUs using ROCm and MIOpen.

In this work, we have added support in FAST for three versions of TensorFlow: 1) using CPU and no extra libraries, 2) with CUDA and cuDNN and 3) ROCm, thus covering CPUs, and GPUs from both NVIDIA and AMD. TensorFlow can be compiled to support several of these in one build, but the binary would then require all dependencies to be present on the system to run. Therefore a separate IE for each of the three builds of TensorFlow were created in FAST.

The CMake superbuild system of FAST downloads TensorFlow automatically and builds the entire C++ library and links it with FAST. In this article, TensorFlow version 1.14 was used with CUDA 10.0, cuDNN 7.6, and ROCm 2.6.

2) OPENVINO

The OpenVINO toolkit [11] from Intel enables inference on different processors from Intel using a C++ API. This toolkit can execute neural networks on multi-core CPUs, the integrated GPUs (HD graphics), as well as the neural compute stick, a USB stick with a dedicated vision processing unit (VPU).

OpenVINO is open-source with an Apache 2.0 license. When targeting GPUs, OpenVINO uses OpenCL and a library called clDNN to perform inference. For CPUs, Intel's MKL library is used.

The OpenVINO toolkit can only be used for inference, not training. Thus you need a trained model, e.g. a protobuf file from TensorFlow, which you have to convert to an intermediate format consisting of an .xml and a .bin file using the model optimizer tool included in the toolkit. In this article, the 2019 R1 version of OpenVINO was used.

3) TENSORRT

NVIDIA's proprietary TensorRT [16] platform enables high performance inference on NVIDIA's GPUs using several optimizations such as precision calibration, layer fusing and kernel auto-tuning.

The current version of TensorRT, version 5.1, only supports reading files stored in Caffe, UFF or ONNX format. Thus, if you have TensorFlow models stored in the protobuf format, you will first need to convert to the UFF format. Also, TensorRT requires channel first dimension ordering, but currently the model converter can not do this

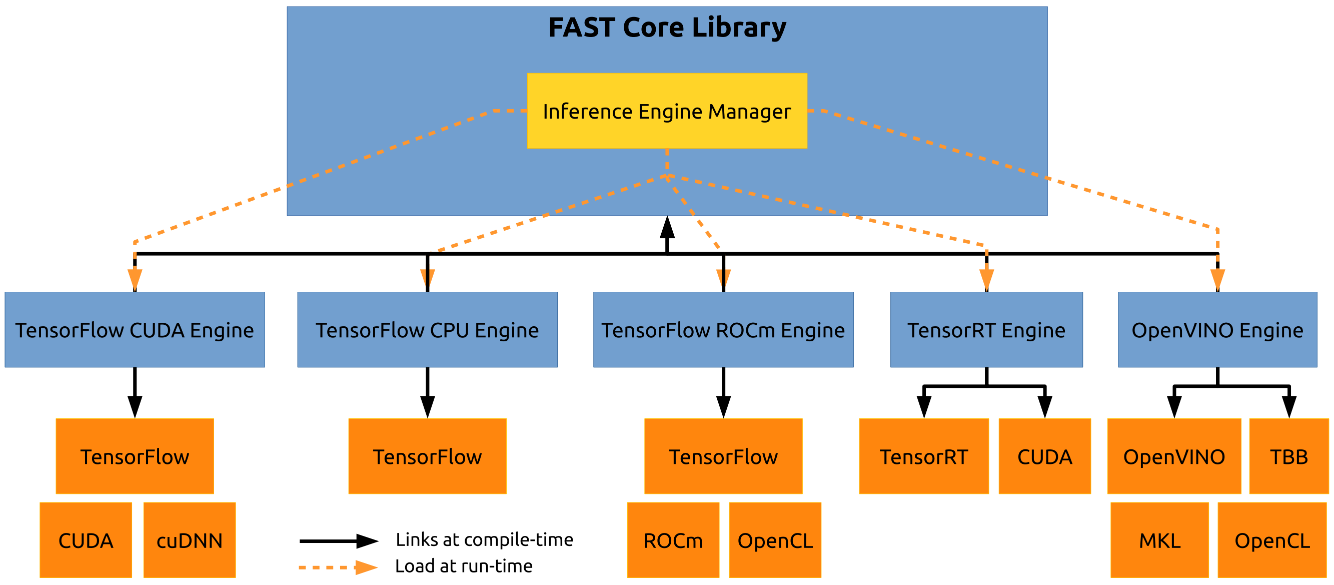

FIGURE 1. A block diagram of the libraries involved. Each inference engine implements a common inference engine interface and is compiled as a separate library which links the FAST core library. The inference engine manager in FAST load the inference engines at runtime. Each inference engine can have several external dependencies as illustrated with orange boxes.

automatically. You therefore have to train your model with the channel first ordering in order to use TensorRT.

When the model is loaded in FAST with the TensorRT IE, a kernel auto tuning will be performed the first time. This takes a lot of time, and therefore FAST will automatically store a cached version on disk of the optimized model which will be loaded instead of the original model while it is unchanged.

D. INFERENCE ENGINE MANAGER

Each IE has several library dependencies, some of which have to be installed separately by the user and cannot be distributed with FAST. Linking all dependencies to FAST at compile time would make it only possible to run FAST on a system that has all IEs and their dependencies installed on the system. Thus an inference engine plugin system was created in FAST in which every engine is compiled as a separate dynamic library. Each engine then link to FAST and their respective dependencies as shown in Fig. 1. When a neural network is set up in FAST, an *inference engine manager* will try to load all the inference engines during runtime. If an IE is missing any of its dependencies, it will simply register that IE as unavailable. The inference engine manager can then be used to query which IEs are available, and load a specific or the best available IE.

E. FAST NEURAL NETWORK IMPLEMENTATION

A neural network process object was added to FAST with methods for loading a neural network from disk, setting input/output nodes, specifying input normalization schemes and also specifying which IE to use in case several IEs are available on the system. The neural network process object is completely generic in the sense that it accepts any number of input/output nodes of any tensor shape. To reduce the amount of code needed to setup a neural network, specialized process object for two common networks types were created. These are *ImageClassificationNetwork*, and *SegmentationNetwork* which are neural networks that perform image classification and segmentation respectively. The *ImageClassificationNetwork* takes an input image and outputs a list of confidence values for each possible image class. A list of image class names can be provided to this class for convenience. The *SegmentationNetwork* takes an input image and outputs the segmentation either as a label image, where each pixel has a positive integer value representing its class which may be visualized with the *SegmentationRenderer*, or as a tensor with confidence values for each class which can be visualized as a heatmap using the *HeatmapRenderer*.

1) TENSORS

FAST already has a data object for representing 2D and 3D images. This data object can store images as OpenCL buffers, images/textures on the GPU or as a 1D array in CPU memory. To support any N-dimensional data used for neural network input and output, a tensor data object was added to FAST. This object in FAST uses channel last ordering since this is the same used in OpenCL and OpenGL which FAST uses. The network process object accepts both tensor and image data objects, but internally images are converted to tensors before execution. At the same time as the tensor conversion, intensity normalization is performed such as converting pixels to a 0-1 range, subtracting mean and dividing by a standard deviation, and also converting to the required dimension ordering of the selected IE. This conversion all happens on the

GPU using OpenCL and the tensors can be read on the GPU using OpenCL buffers and on the CPU as Eigen tensors. After the conversion, the tensor data is given to the selected IE. After inference, the output is converted to FAST tensor data objects and may be passed on to other process objects or renderers for visualization.

2) BATCH PROCESSING

Most often during inference, only one sample is processed at a time, while in some cases, there is a need to process several. Processing several samples at the same time can provide a performance gain in terms of samples/seconds when using GPUs. To support this, a new data object *Batch* was added to FAST which simply is a list of tensors or images. During inference, a Batch data object is converted to a tensor of shape (B, X) where X is the shape of the tensors in the Batch object, which all must have the exact same shape.

3) MULTI-INPUT AND MULTI-OUTPUT NETWORKS

Most neural networks have only one input and one output, but in some cases neural networks will have multiple inputs or outputs, for instance in the case of multi-task learning. In FAST, an input and output data port is created for each input and output node of the neural network. Thus, complex pipelines can be created by connecting various processing objects with these input/output ports. Each input and output port is identified by a port ID (a positive integer) and a name. Depending on which IE used, it may be necessary to define the input/output nodes and/or their shapes manually in code, while some IEs can automatically detect this from the network definition file. In case of batch processing, the batch sizes has to be the same for all input nodes.

4) SEQUENCE DATA

A sequence of data can for instance be image frames in a video, or a set of slices in a volume. For this kind of data another FAST data object called *Sequence* is made, which is a list of tensors. If one sequence of S images of size $H \times W$ and C channels is sent to a neural network, it is converted to a tensor of shape $(1, S, H, W, C)$ or $(1, S, C, H, W)$ depending on dimension ordering.

5) PATCH PROCESSING

To facilitate patch processing, which is necessary in cases where the input image or volume is too large to be processed directly, two process objects are used, a *patch generator* and a *patch stitcher*. The patch generator creates a separate thread of execution which creates a stream of patches from the original full image/volume. The use of multi-threading here, enables patches to be generated during network inference and other processing. After, network inference, the patch stitcher converts the stream of patches into a full image. In order to do this, the patch stitcher need information such as: 1) the size of the original image, 2) the size of the patches 3) the position of the given patch, and 4) the pixel spacing. All this information is embedded as *frame data* in the data object of each patch. This frame data is always copied from input to output data after being processed by a process object. The patch stitcher keeps a copy of the image it is creating from the patches, and updates it continuously until the patch generator has stopped producing patches.

F. USE CASES

Three different medical use cases were investigated with high performance inference in focus. In this section, the neural network, data, and pipeline used for each use case are described.

Example code for setting up these pipeline in FAST are provided in appendix A, B and C. This code is included to give the reader an impression of how little code is needed to set up these kind of pipelines in FAST.

1) CASE 1 - REAL-TIME ULTRASOUND IMAGE SEGMENTATION

In this use case, a neural network for segmentation of the carotid artery and jugular vein from B-mode ultrasound images was used. The network is a fully convolutional encoder-decoder architecture with skip-connections, 3×3 convolutions and ReLU activations. Max pooling was used in the encoder, while upsampling was used in the decoder. For the TensorRT IE, upsampling is not currently supported, and was therefore replaced with transposed convolution. The final layer is a pixelwise softmax activation with 3 channels. The size of the input and output is a 256×256 image and the input is normalized to a 0-1 range before inference. The network has about 2 million parameters.

Figure 2 shows the FAST pipeline for this use case. An ultrasound image streamer produces a stream of images either directly from an ultrasound scanner or from disk. The images are passed directly to a neural network for segmentation. The ultrasound images and the segmentations are rendered together to visualize the segmentation on top of the ultrasound image.

2) CASE 2 - CT VOLUME SEGMENTATION

This use case focus on volumetric segmentation of lung nodules from thoracic CT volumes. CT volumes have a large intensity range and were therefore clipped to the range $[-1200, 400]$ Hounsfield units and then normalized to a 0-1 range. The network used is similar to the one in case 1, but in 3D instead using 3D convolutions. Due to memory constraints, the network process 32 slices per sample. Thus, the input and output size is $32 \times 256 \times 256$. The model has ~ 5.9 million parameters. The network was trained using the The Lung Image Database Consortium and Image Database Resource Initiative (LIDC-IDRI) dataset [17]. A thoracic CT image of patient 72 from this dataset was used for inference in this use case. The image has 305 slices of size 512×512.

Figure 3 shows the FAST pipeline for this use case. The volume is first imported from a DICOM file using the DCMTK library [18], and passed on to the patch generator. The patch generator generates a stream of subvolumes that are passed on to the neural network for segmentation.

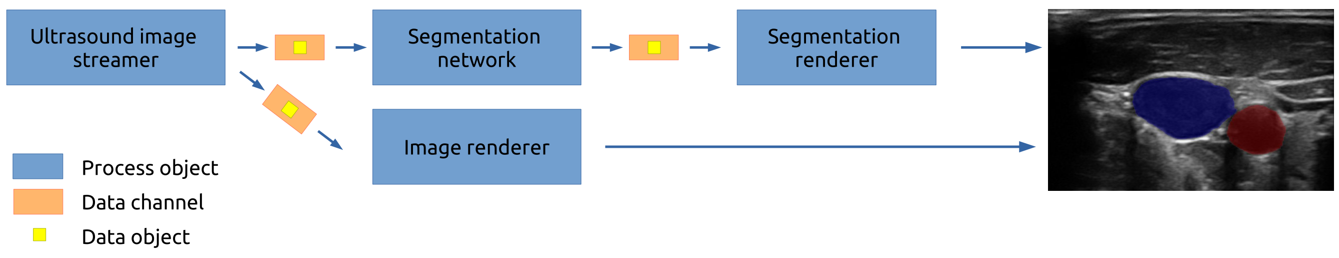

FIGURE 2. Case 1 - Real-time ultrasound image segmentation: An ultrasound image streamer produces a stream of images either directly from an ultrasound scanner or from disk. The images are passed directly to a neural network for segmentation and the result is rendered with transparent colors on top of the ultrasound image. A video of this is available online in the supplementary material and on YouTube https://youtu.be/iuevRnZMDgg.

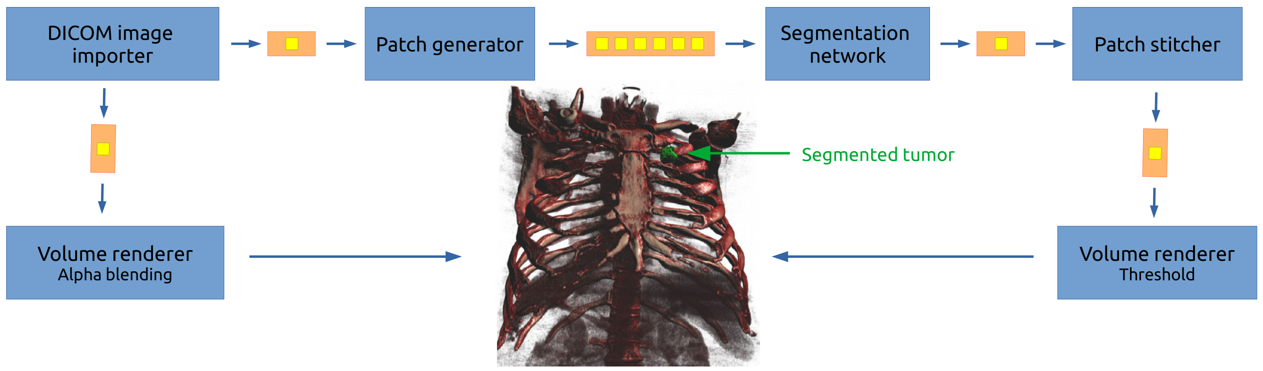

FIGURE 3. Case 2 - CT Volume segmentation: The volume is imported from a DICOM file, and passed on to the patch generator and the volume renderer. The patch generator generates a stream of subvolumes that are passed on to the neural network for segmentation. This is necessary because there is usually not a enough GPU memory to process the entire volume in one go. After neural network processing, the volume patch is stitched to create a volume of the same size as the CT volume. Finally, the segmented lung tumour is rendered using a threshold volume renderer shown here in green. A video of this is available online in the supplementary material and on YouTube https://youtu.be/iuevRnZMDgg.

After the neural network processing, the volume patch is stitched to create a volume of the same size as the input volume. Finally, the CT volume and the segmentation volume are rendered using a GPU-based ray casting volume renderer.

3) CASE 3 - PATCH-WISE CLASSIFICATION OF A WHOLE SLIDE IMAGE

In this use case, a convolutional neural network for patch-wise classification of whole slide microscopy images was used. These kind of images are usually stored as tiled image pyramids, in various proprietary formats. FAST uses the OpenSlide [19] library to load WSIs from disk. As with use case 2, due to memory constraints, the neural network is forced to process patches. A modified Inception v3 network [20], using 64 hidden neurons in the first fully-connected layer and a dropout rate of 0.5, was trained on the Grand Challenge on Breast Cancer Histology Images (BACH) dataset [21]. Input to the network are RGB color patches of size 512×512, extracted from the 20x optical magnification level and RGB intensities normalized to 0-1. The model classifies breast tissue into the four classes: normal tissue, benign lesion, in-situ carcinoma and invasive carcinoma. In the last layer, a softmax activation function with 4 channels was used. The model has $\sim$ 21.9 million parameters. The test image used for inference is A05 from the BACH dataset. It is a H&E stained brightfield image of $\sim 52k \times 43k$ pixels.

Figure 4 illustrates the FAST pipeline for this use case. The large WSI is imported using OpenSlide and virtual memory. Like with use case 2, the image is processed through the patch generator which generates patches which are passed on to the neural network for classification. The patch classifications are passed on to the patch stitcher which merges the results to form a result image which is renderer as a colored heatmap, where each color represents a different class, and the opacity represents the class confidence. Since WSIs often contain many pixels corresponding to the glass slide and not actual tissue, a rough tissue segmentation was first performed. This segmentation is used as a mask in the patch generator such that the generator only creates patches were there is tissue.

Due to the parallel nature of GPUs, these processors can batch process several samples in parallel, thereby increasing throughput by processing several images per second. For this purpose, an image to batch generator was made and tested in this use case. This generator takes in a stream of images and converts them to a stream of batches, where B number of images are put into a Batch data object.

III. RESULTS

Runtime for each use case was measured with different IEs and processors and collected in tables 2, 3 and 4. For the neural network step, the runtime of three parts were measured: 1) Pre-processing and transfer of input data

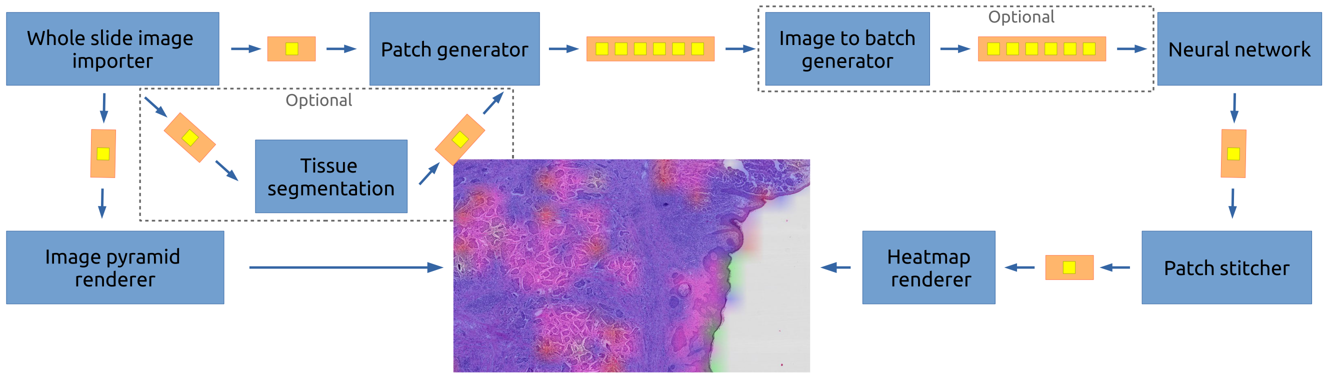

FIGURE 4. **Case 3 - Patch-wise classification of a whole slide image: A large whole slide microscopy image is imported using virtual memory. Like the CT volume, these images are also too large to process in one go, and a patch generator and stitcher is used. To speed up the processing, an optional tissue segmentation and image to batch generator can be used. Finally, a visualization is created using an image pyramid renderer, and a heatmap renderer displaying the class of each patch with specific class colors overlaid on the microscopy image. A video of this is available online in the supplementary material and on YouTube https://youtu.be/iuevRnZMDgg.**

TABLE 2. **Runtime measurements of Case 1 - Ultrasound image segmentation. The total includes everything except rendering, such as loading data from disk and neural network processing of 171 ultrasound frames. The total average and standard deviation was calculated based on 10 consecutive runs. The neural network input, inference and output are reported as the average per frame.**

Inference engine	**Processor**	**Runtime (ms)**			
		NN input	NN inference	NN output	Total
TensorFlow CPU	Intel i5-4460	1 ± 0	140 ± 30	0 ± 0	$24,193 \pm 1,226$
TensorFlow CUDA	NVIDIA GTX 1080 Ti	1 ± 0	7 ± 33	0 ± 0	$1,547 \pm 17$
TensorFlow ROCm	AMD Radeon R9 Fury	1 ± 0	12 ± 34	1 ± 0	$2,364 \pm 23$
OpenVINO	Intel i7-7990	1 ± 0	105 ± 10	1 ± 0	$18,222 \pm 2,708$
	Intel HD Graphics 620	1 ± 0	32 ± 1	1 ± 0	$5,898 \pm 26$
	Intel Neural Compute Stick 2	2 ± 1	85 ± 1	2 ± 1	$15,145 \pm 52$
TensorRT	NVIDIA GTX 1080 Ti	1 ± 0	2 ± 0	0 ± 0	545 ± 22

TABLE 3. **Runtime measurements of Case 2 - CT volume segmentation. OpenVINO and TensorRT are not included here as they do not support 3D convolutions. Also, the TensorFlow ROCm failed to run this use case. The total includes everything except rendering, such as loading data from disk and neural network processing of the entire CT volume. The total average and standard deviation was calculated based on 10 consecutive runs. The patch generator, stitcher and neural network input, inference and output are reported as the average per patch.**

Inference engine	**Processor**	**Runtime (ms)**					
		Patch generator	NN input	NN Inference	NN output	Patch stitcher	Total
TensorFlow CPU	Intel i5-4460	26 ± 32	14 ± 0	$10,441 \pm 145$	12 ± 0	67 ± 113	$53,296 \pm 278$
TensorFlow CUDA	NVIDIA GTX 1080 Ti	25 ± 31	14 ± 1	302 ± 139	12 ± 0	67 ± 112	$2,477 \pm 6$

to the IE, 2) the actual inference time, and 3) post-processing and transfer of output data from the IE to FAST. These runtime measurements were included to give an impression of how quickly different data can be processed with NNs using FAST and which processing steps take most time.

For use case 2 and 3, the runtime of the patch generator and stitcher were calculated as the average time to process generate/stitch one patch. The total runtime of use case 2 includes all steps necessary to process the entire volume, except the rendering which runs in real-time. For this use case, only the TensorFlow IE was used, since it is the only IE that supports 3D convolutions at the moment. Also, the TensorFlow ROCm failed to run this use case. Similarly, the total runtime of use case 3 includes all steps necessary to process the entire WSI, except rendering, using all 3,320 patches classified as tissue using tissue segmentation.

The total runtime average and standard deviation was calculated using 10 consecutive runs. The neural network input, inference and output, as well as the patch generator and stitcher, are reported as the average per frame/patch.

TABLE 4. Runtime measurements of Case 3 - Patch-wise classification of a whole slide image. The total includes everything except rendering, such as loading data from disk and neural network processing of the entire image. The total average and standard deviation was calculated based on 10 consecutive runs. The patch generator, stitcher and neural network input, inference and output are reported as the average per patch of 3,320 patches classified as tissue.

Inference engine	Processor	Runtime (ms)					
		Patch generator	NN input	NN inference	NN output	Patch stitcher	Total
TensorFlow CPU	Intel i5-4460	29 ± 5	2 ± 0	152 ± 19	0 ± 0	0 ± 0	$510,564 \pm 1,044$
TensorFlow CUDA	NVIDIA GTX 1080 Ti	25 ± 4	2 ± 0	19 ± 26	0 ± 0	0 ± 0	$84,686 \pm 1,033$
TensorFlow ROCm	AMD Radeon R9 Fury	24 ± 4	1 ± 0	29 ± 26	0 ± 0	0 ± 0	$102,594 \pm 269$
OpenVINO	Intel i7-7990	67 ± 11	4 ± 2	283 ± 39	0 ± 0	0 ± 0	$954,563 \pm 29,203$
	Intel HD Graphics 620	174 ± 8	6 ± 4	167 ± 5	0 ± 0	0 ± 0	$581,054 \pm 16,304$
	Intel Neural Compute Stick 2	54 ± 10	14 ± 8	248 ± 14	0 ± 0	0 ± 0	$873,093 \pm 1,576$
TensorRT	NVIDIA GTX 1080 Ti	24 ± 4	1 ± 0	9 ± 0	0 ± 0	0 ± 0	$81,298 \pm 180$

Note that data streaming, processing, GPU scheduling, and visualization in FAST all run concurrently in different threads.

The runtime of the TensorFlow and TensorRT IEs were measured on Ubuntu 18.04 64 bit operating system, while OpenVINO runtime was measured on Windows 10 64 bit.

A video of these three uses cases and more are available online in the supplementary material and on YouTube (https://youtu.be/iuevRnZMDgg).

IV. DISCUSSION

The world of inference engines is currently fragmented. Although they are all made to process neural networks, they differ in several aspects as shown in Table 1. FAST manages to hide these differences by providing a common interface for all of them and still provide high performance inference. Still, an issue that remains is the storage format of neural networks. For each use case in this article, the original TensorFlow protobuf file created after training had to be converted to the UFF format used by TensorRT and the OpenVINO format using scripts provided by the vendors. Hopefully, the industry will land on a common storage format in the future which will remove the need for manually converting between formats. At the moment there are two candidates for a standard storage format: Open Neural Network eXchange format (ONNX) [22] initiated by Facebook and Microsoft, and the Neural Network Exchange Format (NNEF) [23], developed by the Khronos Group, an industry consortium.

Of the three inference engines investigated, TensorFlow is currently the only one supporting 3D convolutions which is useful when processing CT and MRI volumes.

The deep learning scene has long been dominated by NVIDIA and their proprietary software CUDA, cuDNN and TensorRT, and thus NVIDIA GPUs have, and still are, considered almost mandatory for doing deep learning. Recently, other options such as Intel's OpenVINO and AMD's ROCm/MIOpen have arrived which are using open-source libraries such as OpenCL. This is great news for reducing the NVIDIA dominance and their proprietary software in deep learning. However, developers are still faced with multiple libraries with different API's. In this article, we have showed how FAST was extended to support all of these inference engines and processors using a common interface making it easier for the developer.

Performance comparison of the three different use cases, inference engines and processors indicated that TensorRT was the fastest IE at the moment, beating TensorFlow with cuDNN on the same GPU, and was able to do ultrasound image segmentation inference in just 2 milliseconds on a consumer NVIDIA GPU. A fair comparison with AMD's ROCm was not possible since only an older AMD GPU was available for this study. Taking into consideration that the R9 Fury is two years older than the 1080 Ti GPU, the performance numbers show a decent runtime for the ROCm platform. Still, the choice of which inference engine and processor to use for medical neural network image analysis can affect runtime performance considerably. The biggest relative difference was seen in case 1 of processing ultrasound frames, where the difference between the fastest and slowest cases were half a second vs. 24 seconds. For volume processing, using the CPU or the GPU, showed a difference of 2 vs 53 seconds. Due to the extreme size of the whole slide microscopy images used in digital pathology, processing these images using deep neural networks is very time consuming. The fastest configuration tested in this article used about 81 seconds to process a single WSI, while the slowest IE used almost 16 minutes. On OpenVINO, using the integrated GPU was generally fastest, followed by the Neural Compute Stick VPU and the slowest which was the CPU.

Most scientists use Python to train and test their neural networks due the simpler syntax and semantics of the Python language. Since the neural network libraries, such as TensorFlow, are essentially written in C++ and optimized for GPUs, inference is still fast when using these libraries in

Python. The real benefit of using a C++ framework such as FAST, instead of Python libraries, comes when you want to use trained neural networks in practice, such as doing real-time inference on streamed ultrasound images, or processing large images and volumes with complex pipelines and visualize them with advanced rendering techniques such as GPU-based ray casting and tiled image pyramid rendering of extremely large microscopy images. In these cases, utilizing the parallel capabilities of modern CPUs and GPUs, as well as avoiding unnecessary large data copies and the slow nature of the interpreted Python language is essential for achieving high performance use of neural networks in medical image analysis.

In the introduction, the proposed FAST solution was compared to related frameworks such as OpenCV, Clara AI and DeepInfer. All of these four frameworks have different pros and cons, and in the end it is all about selecting the best tool for the problem at hand. Currently, FAST is the only one of these frameworks to support whole slide imaging, real-time ultrasound streaming, and multiple IEs with a common interface. It is also the only open framework that provides data loading, streaming, processing, NN inference and visualization of medical images in one single software package.

V. CONCLUSION

Methods for extending the medical open-source framework FAST were presented together with an overview of current inference engines. It was shown how these methods enable FAST to perform high performance neural network inference using different inference engines and processors with a common API on three different medical use cases.

Most of the inference engines and necessary libraries investigated were open-source, with the exception of NVIDIA's software CUDA, cuDNN and TensorRT. TensorFlow was the only IE to support 3D convolutions.

Performance comparison of the three different use cases, inference engines and processors showed large differences in runtime, especially when using different processors such as GPUs and CPUs. While an ultrasound image can be processed by a neural network in only a couple of milliseconds, processing a whole slide microscopy image can take several minutes.

Source code, documentation, binary releases and test data can be found online on GitHub at https://github.com/smistad/FAST/.

APPENDIX. USE CASE SOURCE CODE

This section is meant to show the reader the amount of code needed to set up the use cases in this article using FAST. Header includes and the C main function have been omitted, and only setup for TensorFlow protobuf files is shown. An executable binary for each use case is available in the FAST version 3 release on GitHub https://github.com/smistad/FAST/releases/. The executables for use case 1, 2 and 3 are named *neuralNetworkUltrasoundSegmentation*, *neuralNetworkCTSegmentation* and *neuralNetworkWSIClassification* respectively.

A. USE CASE 1: REAL-TIME ULTRASOUND IMAGE SEGMENTATION

```
// Stream data from an ultrasound device
// OpenIGTLink is used as an example here, but it
// is also possible to stream from a Clarius
// scanner or from files stored on disk.
auto streamer = OpenIGTLinkStreamer::New();
streamer->setConnectionAddress("192.168.0.1");

// Set up neural network
auto segmentation = SegmentationNetwork::New();
segmentation->setOutputNode(0,
    "conv2d_23/truediv");
segmentation->load("path_to_some_neural_network.pb");
segmentation->setInputConnection(
    streamer->getOutputPort());
// Neural network input pre processing parameters
segmentation->setScaleFactor(1.0f/255.0f);

// Set up renderers
auto imageRenderer = ImageRenderer::New();
imageRenderer->addInputConnection(
    streamer->getOutputPort());

auto segmentationRenderer =
    SegmentationRenderer::New();
segmentationRenderer->getOutputPort(
    segmentation->getOutputPort());

// Set up window and start pipeline
auto window = SimpleWindow::New();
window->addRenderer(imageRenderer);
window->addRenderer(segmentationRenderer);
window->set2DMode();
window->start();
```

B. USE CASE 2: CT VOLUME SEGMENTATION

```
// Import CT image from a file
auto importer = ImageFileImporter::New();
importer->setFilename("/path/to/CT-thorax.dcm");

// Generate patches from volume
auto generator = PatchGenerator::New();
generator->setPatchSize(512, 512, 32);
generator->setInputConnection(
    importer->getOutputPort());

// Set up neural network
auto network = SegmentationNetwork::New();
network->setOutputNode(0, "conv3d_18/truediv");
network->load("/path/to/some/neural_network.pb");
network->setInputConnection(
    generator->getOutputPort());
network->setResizeBackToOriginalSize(true);
// Neural network input pre processing parameters
network->setMinAndMaxIntensity(-1200.0f, 400.0f);
network->setScaleFactor(1.0f/(400+1200));
network->setMeanAndStandardDeviation(-1200.0f,
    1.0f);

// Stitch the patch results back together
auto stitcher = PatchStitcher::New();
stitcher->setInputConnection(
    network->getOutputPort());

// Set up renderers
auto renderer =
    AlphaBlendingVolumeRenderer::New();
```

```
renderer->setTransferFunction(
    TransferFunction::CT_Blood_And_Bone());
renderer->addInputConnection(
    importer->getOutputPort());

auto renderer2 = ThresholdVolumeRenderer::New();
renderer2->addInputConnection(
    stitcher->getOutputPort());

// Setup window
auto window = SimpleWindow::New();
window->addRenderer(renderer);
window->addRenderer(renderer2);
window->start();
```

C. USE CASE 3: PATCH-WISE CLASSIFICATION OF WHOLE SLIDE IMAGE

```
auto importer = WholeSlideImageImporter::New();
importer->setFilename("/path/to/some/WSI.tiff");

auto tissueSegmentation =
    TissueSegmentation::New();
tissueSegmentation->setInputConnection(
    importer->getOutputPort());

// Generate patches from highest resolution level
    (0) of the WSI
auto generator = PatchGenerator::New();
generator->setPatchSize(512, 512);
generator->setPatchLevel(0);
generator->setInputConnection(0,
    importer->getOutputPort());
generator->setInputConnection(1,
    tissueSegmentation->getOutputPort());

// Set up neural network
auto network = NeuralNetwork::New();
network->setOutputNode(0, "dense_1/Softmax");
network->load("/path/to/some/neural_network.pb");
network->setInputConnection(
    generator->getOutputPort());
// Neural network input pre processing parameters
network->setScaleFactor(1.0f/255.0f);

// Stitch patch wise classifications together
auto stitcher = PatchStitcher::New();
stitcher->setInputConnection(
    network->getOutputPort());

// Set up renderers
auto renderer = ImagePyramidRenderer::New();
renderer->addInputConnection(
    importer->getOutputPort());

auto heatmapRenderer = HeatmapRenderer::New();
heatmapRenderer->addInputConnection(
    stitcher->getOutputPort());

// Set up window and start
auto window = SimpleWindow::New();
window->addRenderer(renderer);
window->addRenderer(heatmapRenderer);
window->set2DMode();
window->start();
```

REFERENCES

[1] G. Litjens, T. Kooi, B. E. Bejnordi, A. A. A. Setio, F. Ciompi, M. Ghafoorian, J. A. W. M. van der Laak, B. van Ginneken, and C. I. Sánchez, "A survey on deep learning in medical image analysis," *Med. Image Anal.*, vol. 42, pp. 60–88, Dec. 2017.

[2] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous distributed systems," Mar. 2016, *arXiv:1603.04467*. [Online]. Available: https://arxiv.org/abs/1603.04467

[3] E. Smistad, M. Bozorgi, and F. Lindseth, "FAST: Framework for heterogeneous medical image computing and visualization," *Int. J. Comput. Assist. Radiol. Surgery*, vol. 10, no. 11, pp. 1811–1822, 2015. [Online]. Available: http://link.springer.com/10.1007/s11548-015-1158-5

[4] E. Smistad, D. H. Iversen, L. Leidig, J. B. L. Bakeng, K. F. Johansen, and F. Lindseth, "Automatic segmentation and probe guidance for real-time assistance of ultrasound-guided femoral nerve blocks," *Ultrasound Med. Biol.*, vol. 43, no. 1, pp. 218–226, 2017.

[5] E. Smistad and F. Lindseth, "Real-time automatic artery segmentation, reconstruction and registration for ultrasound-guided regional anaesthesia of the femoral nerve," *IEEE Trans. Med. Imag.*, vol. 35, no. 3, pp. 752–761, Mar. 2016. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7305813

[6] A. Østvik, E. Smistad, S. A. Aase, B. O. Haugen, and L. Lovstakken, "Real-time standard view classification in transthoracic echocardiography using convolutional neural networks," *Ultrasound Med. Biol.*, vol. 45, no. 2, pp. 374–384, Feb. 2019. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0301562918303259

[7] E. Smistad, A. Østvik, I. M. Salte, S. Leclerc, O. Bernard, and L. Lovstakken, "Fully automatic real-time ejection fraction and MAPSE measurements in 2D echocardiography using deep neural networks," in *Proc. IEEE Int. Ultrason. Symp. (IUS)*, Oct. 2018, pp. 1–4. [Online]. Available: https://ieeexplore.ieee.org/document/8579886/

[8] A. Østvik, L. E. Bø, and E. Smistad, "EchoBot: An open-source robotic ultrasound system," in *Proc. IPCAI*, 2019, pp. 1–4. [Online]. Available: http://www.ipcai2019.org/pdf/Long_Abstracts/Ostvik_etal_IPCAI_2019_Long_Abstract.pdf

[9] P. Bándi *et al.*, "From detection of individual metastases to classification of lymph node status at the patient level: The CAMELYON17 challenge," *IEEE Trans. Med. Imag.*, vol. 38, no. 2, pp. 550–560, Feb. 2019.

[10] OpenCV. (2019). *OpenCV*. Accessed: Aug. 28, 2019. [Online]. Available: https://opencv.org

[11] Intel. (2019). *OpenVINO Toolkit*. Accessed: Jun. 10, 2019. [Online]. Available: https://software.intel.com/openvino-toolkit

[12] NVIDIA. (2019). *Clara AI*. Accessed: Aug. 29, 2019. [Online]. Available: https://developer.nvidia.com/clara

[13] M. Nolden, S. Zelzer, A. Seitel, D. Wald, M. Müller, A. M. Franz, D. Maleike, M. Fangerau, M. Baumhauer, L. Maier-Hein, K. H. Maier-Hein, H. P. Meinzer, and I. Wolf, "The medical imaging interaction toolkit: Challenges and advances: 10 years of open-source development," *Int. J. Comput. Assist. Radiol. Surg.*, vol. 8, no. 4, pp. 607–620, 2013.

[14] A. Mehrtash, M. Pesteie, J. Hetherington, P. A. Behringer, T. Kapur, W. M. Wells, III, R. Rohling, A. Fedorov, and P. Abolmaesumi, "DeepInfer: Open-source deep learning deployment toolkit for image-guided therapy," in *Proc. Med. Imag., Image-Guided Procedures, Robot. Intervent., Modeling*, vol. 10135, 2017, Art. no. 101351K.

[15] Intel. (2019). *Intel MKL DNN*. Accessed: Jun. 10, 2019. [Online]. Available: https://github.com/intel/mkl-dnn

[16] NVIDIA. (2019). *TensorRT*. Accessed: Jun. 10, 2019. [Online]. Available: https://developer.nvidia.com/tensorrt

[17] S. G. Armato, III, *et al.*, "The lung image database consortium (LIDC) and image database resource initiative (IDRI): A completed reference database of lung nodules on CT scans," *Med. Phys.*, vol. 38, no. 2, pp. 915–931, 2011.

[18] M. Eichelberg, J. Riesmeier, T. Wilkens, A. J. Hewett, A. Barth, and P. Jensch, "Ten years of medical imaging standardization and prototypical implementation: The DICOM standard and the OFFIS DICOM toolkit (DCMTK)," *Proc. SPIE*, vol. 5371, p. 57, Apr. 2004.

[19] M. Satyanarayanan, A. Goode, B. Gilbert, J. Harkes, and D. Jukic, "OpenSlide: A vendor-neutral software foundation for digital pathology," *J. Pathol. Informat.*, vol. 4, no. 1, p. 27, 2013. [Online]. Available: http://www.jpathinformatics.org/text.asp?2013/4/1/27/119005

[20] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Dec. 2015, pp. 2818–2826. [Online]. Available: http://arxiv.org/abs/1512.00567

[21] G. Aresta *et al.*, "BACH: Grand challenge on breast cancer histology images," *Med. Image Anal.*, vol. 56, pp. 122–139, Aug. 2019.

[22] ONNX Organization. (2019). *Open Neural Network Exchange Format (ONNX)*. Accessed: Aug. 15, 2019. [Online]. Available: http://onnx.ai/

[23] The Khronos Group. (2019). *Neural Network Exchange Format (NNEF)*. Accessed: Aug. 15, 2019. [Online]. Available: https://www.khronos.org/nnef

ERIK SMISTAD received the Civil Engineering degree in computer science and the Ph.D. degree in medical image processing from the Norwegian University of Science and Technology (NTNU), Trondheim, Norway, in 2012 and 2015, respectively. He is currently a Research Scientist with SINTEF Medical Technology, an independent not-for-profit research organization, and a Postdoctoral Researcher with NTNU. His main research interests include medical image processing, deep learning, ultrasound, and GPU and parallel computing.

ANDREAS ØSTVIK received the M.Sc. degree in physics and mathematics with specialization within biophysics and medical technology from the Norwegian University of Science and Technology (NTNU), Trondheim, Norway, in 2016, where he is currently pursuing the Ph.D. degree in medical image analysis with focus on machine learning and ultrasound. He is also an Early Stage Research Scientist with SINTEF Medical Technology. His main research interests include medical image analysis, deep learning, ultrasound, and robotics.

ANDRÉ PEDERSEN received the Civil Engineering degree in applied physics and mathematics from the Arctic University of Norway (UiT), Tromsø, Norway, in 2019, specializing in machine learning and statistics. He currently holds a position at SINTEF Medical Technology. His main research interests include medical image analysis, deep learning, and computational pathology and radiology.