M2019KE Document

Matias Hyyppä

30.8.2019

Contents

1 Introduction

Self-driving cars and many modern consumer cars are equipped with sensors which observe the environment around them and make driving decision based on the gathered information. Self-driving cars attempt to make these decision without compromising the safety of the driver or bystanders. The decisions made could include e.g. stopping at the red light and waiting for a pedestrian to cross the road. The sensors that are widely in use include radar, LiDAR (Light Detection and Ranging) and RGB cameras. LiDAR can be used for classification and measuring distances to nearby objects. At the moment LiDAR technology is rather expensive but there's a strong motivation to push the prices down so that consumer level vehicles could be equipped with the technology (for example in the form of solid-state LiDARs). LiDAR sensors have problems performing in certain weather conditions such as snow or rain, however don't depend on external light sources and perform well in dark. The lidar points are made into point clouds in which each point has a known location (x, y, z) in the 3D space. For detecting and classifying objects such as traffic signs, other cars and road lane markings, deep neural networks are widely used.

Self-driving cars can be tested and refined using simulators. When the environment and the actors are completely simulated, it's convenient to generate a lot of data (especially of more rare situations) to train and develop self-driving car models and algorithms. There exists many such simulators e.g. CARLA [16] and NVIDIA Constellation [31].

In recent years neural networks (NN) have gained more and more traction in solving problems in wide range of different disciplines and they have become essential part when designing algorithms for self-driving cars. The popularity can be credited to two major factors: increase in computing power and available data. At the early stages of neural networks the large amount of data needed for training the networks was simply not available. With the inception of internet this hasn't been the case anymore. Same can be said about the computing power, the central processing units (CPU) just few decades ago were too slow to create sophisticated deep learning models in reasonable time, nowadays graphics processing units (GPU) are heavily used for this purpose.

However, due to neural networks requiring a lot of computational power, their scalability and optimization has become increasingly important. In this report we go through some of the general technologies and concepts related to neural network stack as well as how it can be scaled and optimized. At the end we perform a small experiment related to the concepts presented in the report.

2 Background

2.1 CARLA

CARLA is an open-source simulator built for autonomous driving research [16, 10]. It is built on top of Unreal Engine and includes number of assets created solely for CARLA, such as cars and buildings, which can be seen in figure 1 [16, 10]. The simulator includes several premade maps, several which of have been made by dedicated team of digital artists [16]. Number of community maps are also freely available. The world is dynamic and lets the agents to interact with the environment through an interface [16]. CARLA follows the client-server architecture, the server renders the scene and runs the simulation [16]. The client can be interacted with through a Python API [16].

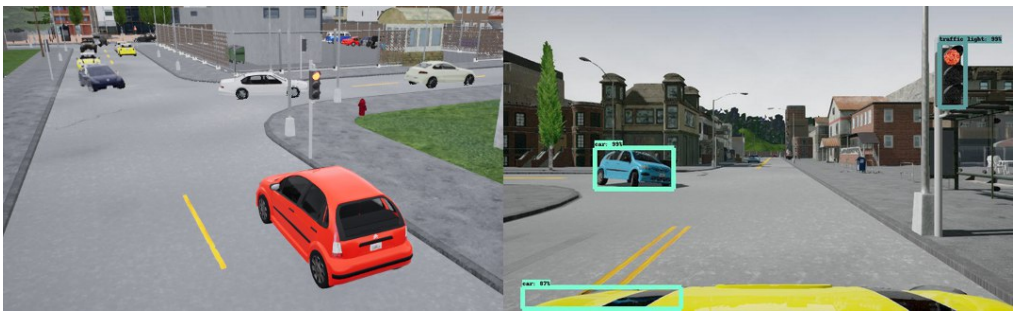

Figure 1: CARLA simulation with object detection

2.2 Microservices

Back in the day (still today but less and less commonly) all software applications were monolithic, i.e. they were self-contained and all the components were combined into a single program. Monolithic applications have slow release cycle and deployment is typically handled by an operations team [28]. Nowadays, the monolithic applications are often separated into individual components called microservices [28, 42]. This enables easier deployment, scaling and update of a software [42]. With microservices the developers can easily deploy the software themselves without the help of operations team due to the smaller deployable units and convenient tools such as Docker and Kubernetes [28]. The smaller deployable units also help with updating or adding a feature to the software [42]. It's easy to scale microservices due to the ease of multiplying the number of instances of certain microservice instead of creating multiple instances of the entire application [42]. Most popular method to orchestrate microservices today is by using Docker and Kubernetes [14, 58].

2.3 Docker

Docker is a containerization technology which enables executing applications simultaneously on same operating system (OS) so that each application has an isolated user-space and share

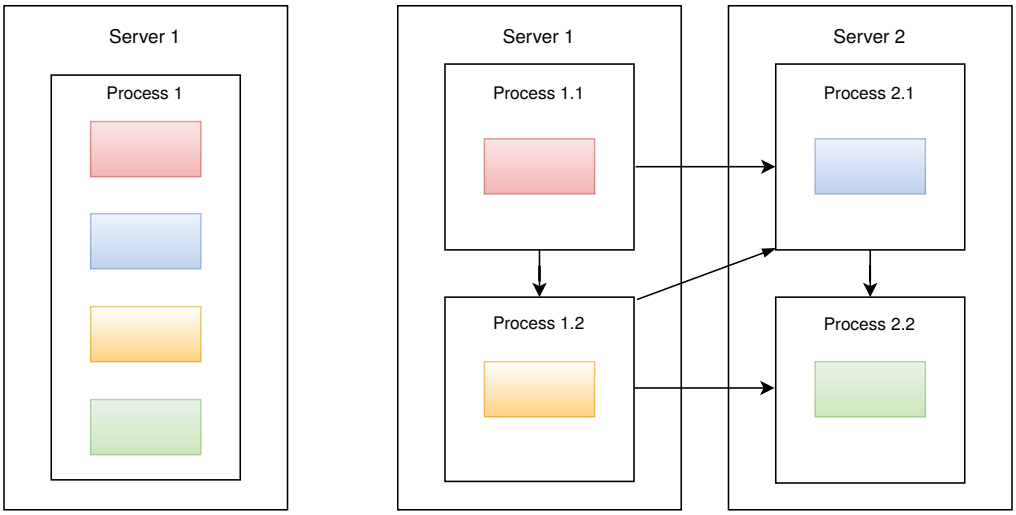

Figure 2: On left side and right side we have examples of monolithic and microservices-based applications, respectively. Adapted from [28]

a common OS kernel [42]. Due to the kernel sharing, containers have a lot less overhead compared to VMs (virtual machines) [29, 42]. It allows a convenient way to deploy software, the software stack that is being used for development can be packaged in a container and shared [29]. Compared to its predecessors, it's easier to use and presents a more standardized way to run containers [42].

According to Stallings [42], Docker consists of the following components: Docker image, Docker client, Docker host, Docker engine, Docker machine, Docker registry and Docker hub. Docker images are read-only templates which contain the instructions that the Docker container is built from, the built container satisfies the requirements an application has to execute inside the said container [3, 29]. Docker client requests for the creation of a new container using an image [42]. Docker host executes Docker containers, Docker engine is required for creating and running the Docker containers on the Docker host [42]. Docker machine is used for setting up Docker engine on virtualized hosts and managing them, the Docker machine makes it possible for the Docker client to talk to the Docker machine. [13, 42]. Docker registry is a stateless, scalable server side application which is used to store and share Docker images. This means that it can be used to publish Docker container images in a public registry (or keep it in a private registry) like Docker hub. [12, 15, 42].

2.4 Kubernetes

Kubernetes is an open-source platform used for automated scaling, deployment and monitoring of containerized workloads and services [37, 57, 58]. It was originally developed by Google but now managed by Cloud Native Computing Foundation [37, 58]. Kubernetes cluster consists of two types of nodes: master nodes and worker nodes [25, 28]. Master node

provides the control plane that manages the entire cluster [25, 28]. Typically, all master nodes are made to run on the same machine for simplicity [25]. Worker nodes are the physical machines or VMs which run the deployed applications using pods [26, 27, 28].

The components of the master nodes control plane, or what is simply referred to as mas-

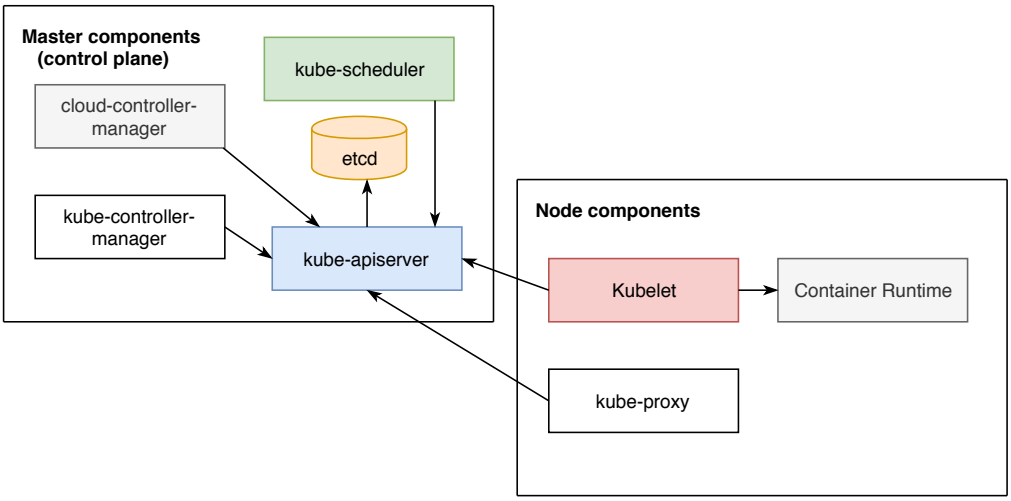

Figure 3: On left side we see the master node components and on right side the worker node components. Adapted from [28]

ter components in the documentation, include the following components: kube-apiserver, etcd, kube-scheduler, kube-controller-manager and cloud-controller-manager [25, 28]. Kube-apiserver is the API component with which users and other master components communicate with, it scales by deploying more instances [25, 28]. Etcd is a distributed key value store to store the cluster data [25, 28]. Kube-scheduler assigns workers to new pods [25, 28]. Kube-controller-manager runs controllers i.e., controllers for replicating, handling failures and such [25, 28]. On the other hand, cloud-control-manager runs controllers which are in touch with different cloud providers [25].

The node components consist of kubelet, kube-proxy and Container Runtime [25]. Kubelet manages the worker nodes' containers, i.e., that they are running in a pod [25, 28]. It also interacts with kube-apiserver [28]. Kube-proxy load-balances network traffic inside the node [28]. Container Runtime represents the runtime used for running containers, for example Docker [25, 28]. Kubernetes supports runtimes which implement the Kubernetes CRI (Container Runtime Interface) [25].

2.5 TensorFlow

TensorFlow is a machine learning framework developed by Google [41]. It is implemented in C++ and provides a Python API as well as C++ API along with several others [40, 41]. The Python API currently supports most features [43, 46]. Keras, a popular high-level API for deep learning is integrated in the TensorFlow core [24]. GPU acceleration in TensorFlow is dependent on using Nvidias parallel computing platform CUDA [40].

In TensorFlow core the computational graph (tf.Graph) is represented as a dataflow graph which can be executed (tf.Session) as a multi-step computation [45, 47, 48, 49]. The dataflow graphs can be broken down into two parts: the nodes and the edges [47]. In TensorFlow the nodes (tf.Operation) are operations which input and output tensors [47, 50]. The edges (tf.Tensor) represent the tensors which flow through the graph [47, 49, 51]. As stated previously, the entire dataflow graph can be executed using the session object [47]. The session encapsulates the state of the TensorFlow runtime, in which operation objects are run and tensor objects (partially defined computations) evaluated [47, 48].

Overview of the TensorFlow architecture can be seen in Figure 4. The client start the

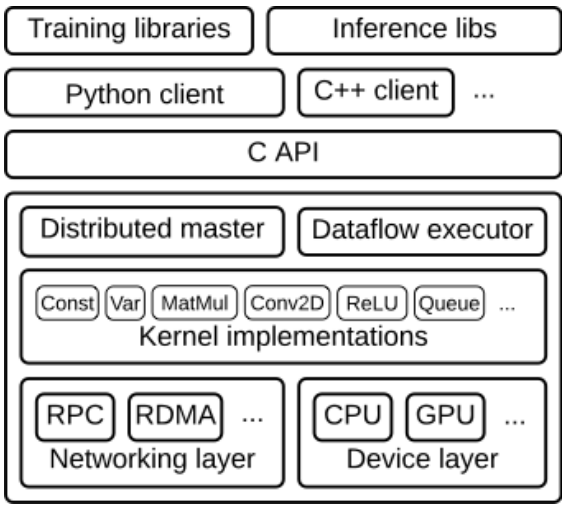

Figure 4: TensorFlow architecture [43]

execution of the dataflow graph by creating a session and sending the graph definition (tf.GraphDef) to the distributed master [43, 52]. Distributed master cuts subgraphs from the graph sent by the client, then distributes the subgraphs to a number of worker services [43, 52]. The dataflow executor is used to partition the assigned subgraph so that the pieces of the subgraph are run in different processes and devices [43, 61]. Pieces of the subgraph are cached for possible later use [43]. Distributed master also distributes and initiates pieces of the graph to worker services which schedule the use of kernel implementations for operations which the received subgraph consists of [43].

2.6 Convolutional Neural Networks

Convolutional neural networks are a type of neural networks, which are used for analyzing data that has grid-like topology). CNNs are widely used for analyzing time-series (1-D grid) as well as image data (2-D grid). Convolution network is simply a network that has one or more convolution operations. The general form of convolution is done on two functions which take as an input a value t, where t $\in \mathbb{R}$. [17]

$$s(t) = (x * w)(t) = \int x(a)w(t-a)da \tag{1}$$

In machine learning the functions on which the convolution operation operates on, are usually tensors. The input consists of e.g. image data and the filter consists of learnable parameters. For example when talking about filters and image data, the filters are smaller along width and height but when it comes to depth, it's same as in the image data. The output of the convolution operation is often called a feature map (also activation map). It's calculated by sliding the filter across the width and height of the input data and compute the dot product between the entries in the filter and the input data as seen in (4). Below we can see an example equation (2) of the discretized form, where the input image and the filter (also called kernel) are convolved. [8, 17]

$$S_{convolution}(i,j) = (Input * Filter)(i,j) = \sum_m \sum_n Input(m,n)Filter(i-m, j-n) \tag{2}$$

Due to convolution being commutative, we can change the order and end up with an equation (3) that can be useful for writing proofs due to obtaining the commutative property [7, 17].

$$S_{convolution}(i,j) = (Filter * Input)(i,j) = \sum_m \sum_n Input(i-m, j-n)Filter(m,n) \tag{3}$$

Typically in neural network and deep learning libraries such as TensorFlow and PyTorch the convolution is actually implemented as a cross-correlation (4) which is almost the same thing except that the kernel is mirrored, which is often seen as more intuitive [17, 36, 44].

$$S_{cross-corr}(i,j) = (Input * Filter)(i,j) = \sum_m \sum_n Input(i+m, j+n)Filter(m,n) \tag{4}$$

In convolutional layer, as opposed to fully connected layers, each neuron is connected to local regions of the input data. This is known as sparse connectivity, it reduces the memory requirements, requires fewer operations and improves statistical efficiency. For example, let **a** be the number of inputs and **b** the number of outputs. Now as an example if we consider

the number of operations required in matrix multiplication we will have a $\times$ b parameters and the asymptotic time complexity of the operation is O(a $\times$ b). With sparse connectivity we can limit the number of outputs to some arbitrary value k. Now the number of needed parameters is a $\times$ k, thus the time complexity is O(a $\times$ k). The dichotomy of sparsely and fully connected layers can be seen in Figure 5. Parameter sharing means that the same parameter is used for more than one function i.e., in CNNs, rather than learning separate set of parameters when sliding filter through the input data, we learn learn only one set of parameters. [8, 17]

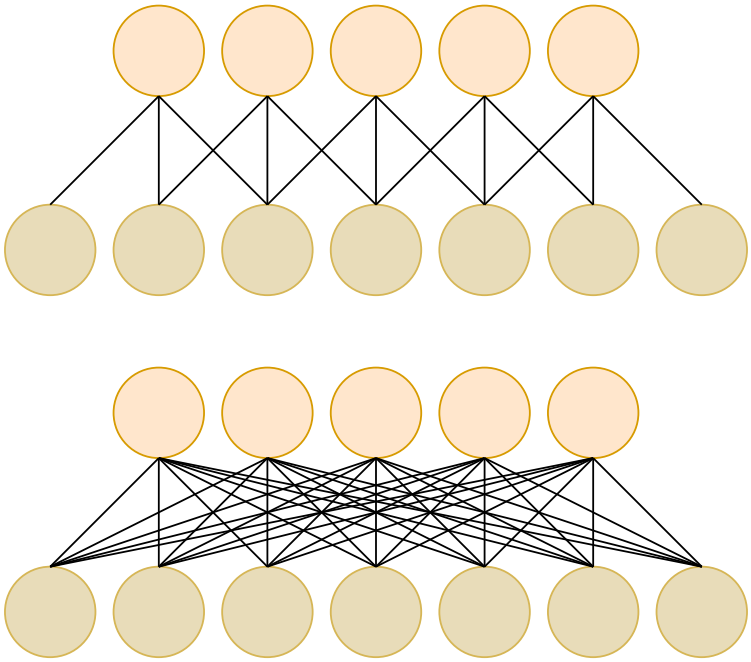

Figure 5: Upper graph is an example of the spatial arrangement of one dimensional sparsely connected CNN. The lower graph is a fully connected graph. Adapted from [8, 17]

Convolutional layers accept volume consisting of width (**W**), height (**H**) and depth (**D**) i.e., W × H × D. For example in RGB image data of size 150x150, both the width and height would be 150 and the depth would be 3. The output size of the convolutional layers is dependent on the the following hyperparameters: stride (**S**), zero-padding (**P**), number of used filters (**K**), and the filter size is called the receptive field (**R**). The stride means the number of pixels the filter moves while doing the convolution operation. Zero-padding is used to ensure that no important information is being thrown out from the edges of the input data and to manipulate the spatial size of the output. With the following equation it can be ensured that the input and output have the same volume [8, 17]

$$P = (R - 1)/2, \text{when S} = 1 \tag{5}$$

The output size can be calculated with the following equations

$$W_2 = (W_1 - R + 2P)/S + 1 \tag{6}$$

$$H_2 = (H_1 - R + 2P)/S + 1 \tag{7}$$

$$D_2 = K \tag{8}$$

In Figure 6 we can see two graphs, both have the same hyperparameters and volume except for the zero-padding, which in the upper graph is $P_u = 0$ and in the lower $P_l = 1$. The shared variables are: $R = 3$, $W_1 = 5$, $H_1 = 1$, $D_1 = 1$, $K = 1$ and $S = 1$. Now if we were to calculate the output volume for the upper graph with weights [1, 0, -1] we would get
$W_2 = (W_1 - R + 2P_u)/S + 1 = (5 - 3 + 0)/1 + 1 = 3$
$H_2 = (H_1 - R + 2P_l)/S + 1 = (1 - 3 + 2)/1 + 1 = 1$
$D_2 = K = 1$
Then calculating the dot products with weights [1, 0, -1] we would get [2, 1, 2] as the output. In the lower graph more information is preserved with zero-padding as can be seen.

If we were to increase the stride to 2 but kept all other variables present in the lower graph the same, the output would look like in in Figure 7. If we increased the stride too much, for example by setting it to 4, we would get a fraction:
$W_2 = (7 - 3 + 2)/4 + 1 = 1.5$

Non-integer result meaning that the convolution can't be calculated so that the filter would fit the input data properly.

After having acquired the feature map, in many cases it's time to apply the nonlinear activation function, such as rectified linear units (ReLU) or sigmoid function (less popular nowadays) [17].

To downsample the input, reduce the number of parameters and the computational time of the network, pool layers are commonly used [17, 8]. One of the most used ones is max pooling which downsamples the input using the max function [17, 8]. Pooling layer output

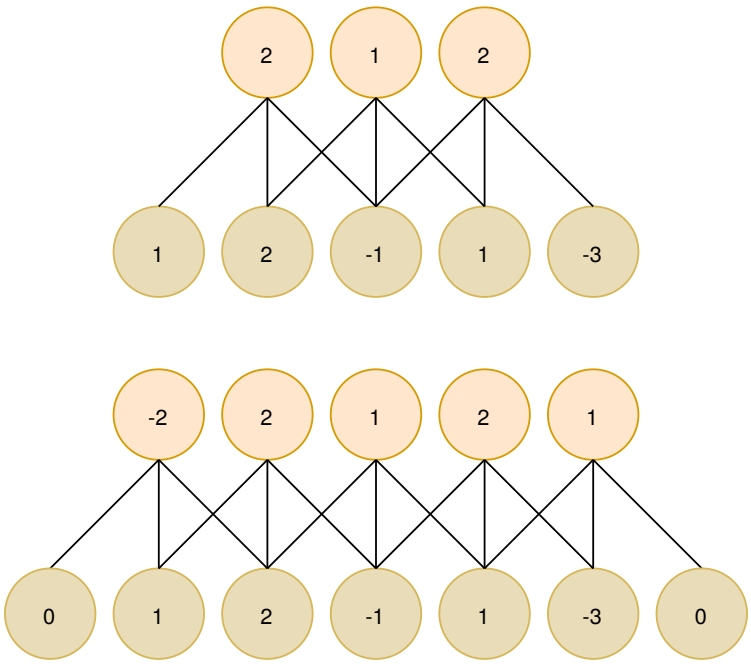

Figure 6: Example of calculating the output volume in convolutional layer with one spatial dimension using stride of 1 and comparing the effects of zero-padding. Adapted from [8, 17]

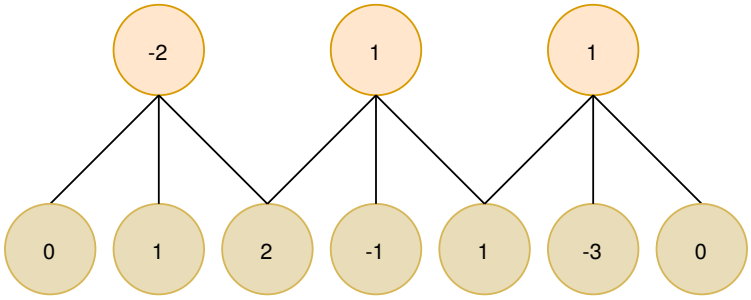

Figure 7: Example of calculating the output volume in convolutional layer with one spatial dimension using stride of 2 and zero-padding. Adapted from [8, 17]

volume can be calculated with functions (6) and (7) with minor adjustment; the zero-padding (P) can be omitted from the equations [8]. The depth stays the same [8]. Example of max pooling and average pooling can be seen in Figure 8. There both the receptive field and stride are set to equal 2. The output comes from calculating the following equations as mentioned before:
$W_2 = (W_1 - R)/S + 1 = (4 - 2)/2 + 1 = 2$
$H_2 = (H_1 - R)/S + 1 = (4 - 2)/2 + 1 = 2$
$D_2 = D_1 = 1$

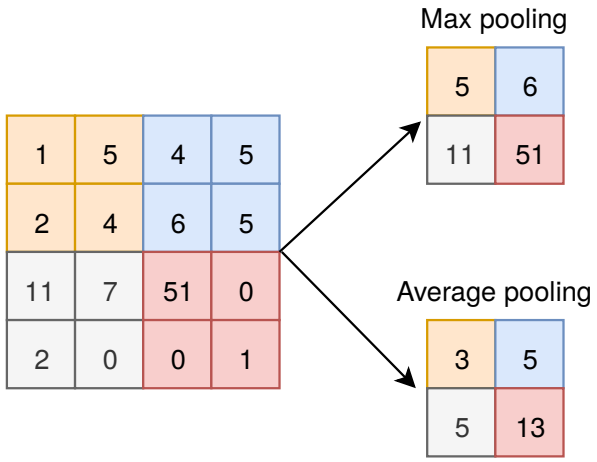

Figure 8: Max pooling and average pooling using standard hyperparameters. Adapted from [8]

2.7 Recurrent Neural Networks

One of the neural network variations which has gained a lot of traction recently is recurrent neural network (RNN). Unlike in feed-forward networks such as CNNs, In RNNs the neural network is constructed so that it can use past information when forming an opinion of the present. This happens by maintaining a state which is updated on every iteration as the algorithm loops through the input data step-by-step.

However it has been noted that in vanilla RNNs it's incredibly hard to handle long-term dependencies. This is due to the gradient based learning algorithm that RNNs use. Gradient descent becomes more and more inefficient as the temporal span of the dependencies increases. [4] This problem is also known as vanishing gradient problem. In the problem the gradient gets so small that subtracting the gradient from the weights makes no difference. The weights essentially stop changing and training slows down.

The other problem that RNNs face is the exploding gradient problem. Now instead of gradient becoming extremely small, it explodes and becomes extremely large.

Long Short Term Memory (LSTM) networks are a form of RNN networks that can handle long-term dependencies. LSTMs were first introduced in 1997 [19] and they were explicitly made to combat the long-term dependency problem. [56]

3 Neural network compiler technologies

Traditionally machine learning focused frameworks have executed the nodes of the graph-like structures one by one, this however is extremely slow [39]. For inference it is increasingly important that graph optimization techniques are utilized, this can be achieved by taking a compiler-oriented approach and handling the graphs to a compiler [39]. In this chapter we briefly go through some of the compiler techniques for optimizing neural networks.

3.1 ONNX

ONNX (Open Neural Network Exchange) is an open deep learning format introduced by Facebook and Microsoft [34]. It enables interoperability of deep learning models between frameworks [34]. Currently ONNX models are supported natively in Caffe2, Microsoft Cognitive Toolkit, MXNet, PyTorch and there exists converters for Theano, TensorFlow and CoreML [34]. Though frameworks such as Tensorflow support static graphs and frameworks like PyTorch support dynamic graphs, they provide interfaces that can be used to construct computational graphs and runtimes [32]. The computational graphs are called intermediate representations (IR) because they can store information without losing any of it and the information can be used for optimization and translation for running the code on arbitrary device [32]. Having a common IR makes switching between frameworks easier [32]. It makes it possible to test and use the most suitable framework for the task at hand e.g., we could be interested in using the framework with most optimal mobile device inference [32]. ONNX operations are versioned as operation sets. Runtimes specify which opsets they support for compatibility purposes.

3.2 XLA

XLA (Accelerated Linear Algebra) is neural network compiler for CPUs, GPUs and accelerators [39, 60]. It attempts to provide more flexibility for TensorFLow when choosing a backend [39]. XLA optimizes the TensorFlow model speed and memory usage, improvements are made by optimizing memory operations by using a process called fusion. All TensorFlow operations have a precompiled GPU kernel implementation. With XLA, the TensorFlow graph can be compiled into a sequence of optimized computation kernels. XLA does this by fusing together different kernels so that the number of kernel launches is reduced. For example, if we have kernel launch for addition and multiplication, now instead of doing two kernel launches we fuse them together into one kernel launch. The intermediate values produced by this operation are not written to memory but instead the results are streamed to the users while having them in GPU registers. [60]

The input language in XLA documentation is referred to as HLO IR (High Level Optimizer Intermediate Representation) [22, 59] as well as simply HLO. It can be thought of as a compiler intermediate representation (compiler IR) [59]. XLA compiles the computational graphs into machine instructions [59]. XLA uses target-independent optimization and

analysis passes, such as operation fusion, buffer analysis and CSE (Common Subexpression Elimination), which seeks expressions which evaluate to same value and analyzes whether to replace them with a variable that holds the computed value [6, 59]. Having done the target-independent optimization and analysis part (see Figure 9) the HLO computation is sent to XLA backend where target-dependent optimization and analysis takes place [59]. In target specific code generation the CPU/GPU backends use LLVM for code generation, low-level IR and optimization [39, 59].

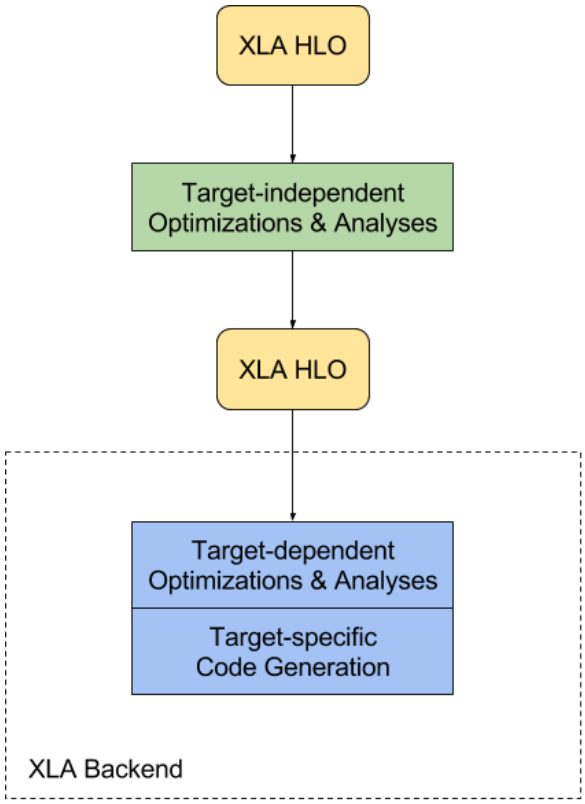

Figure 9: XLA architecture [59]

3.3 TVM/NNVM

NNVM compiler consists of two components: NNVM (Neural Network Virtual Machine) and TVM (Tensor Virtual Machine). NNVM is used for computational graphs and TVM for tensor operators [2, 5, 23, 55]. NNVM attemps to translate workloads from frameworks such as PyTorch to a general format and then into an executable graph [23].

TVM implements low level representation i.e., the operators used in the computational graphs and optimizes them for the target backend [23, 55]. TVM IR is based on domain-specific language (DSL) called Halide [5, 18, 39, 55]. Then according to [39] Halide is used to generate LLVM or CUDA/Metal/OpenCL source code.

3.4 nGraph

Just like NNVM and XLA, nGraphs implements high level conputation graph representation [55]. nGraph IR is a DAG (directed acylic graph), which consists of stateless operation nodes [9]. Figure 10 presents the general architecture of nGraph. When nGraph receives computation graph from a deep learning framework such as TensorFlow, the framework bridge is used to construct the nGraph IR [9, 30]. The nGraph IR is passed to hybrid transformer which generates code for the selected backend by partitions the nGraph IR into a subgraphs [9, 30]. The backends provide pattern matching, liveness analysis, memory management and such [9].

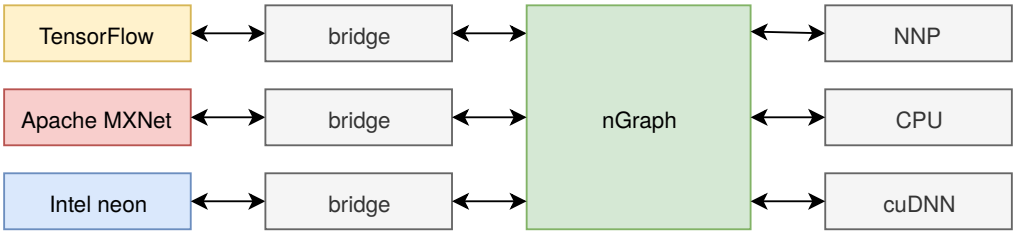

Figure 10: nGraph architecture. Adapted from [9]

4 Experimentation

Next we perform a short experimentation where the ResNet50 convolutional neural network pretrained with ImageNet dataset [20] is loaded into PyTorch and we attempt to export the model into ONNX format. After this, we import the model to TensorFlow and attempt a simple image classification to see what kind of results we get.

```
from torch.autograd import Variable
import torch.onnx
import torchvision
import onnx
import numpy as np
import imageio
import tensorflow as tf
from onnx_tf.backend import prepare
```

First we have to import all the necessary libraries such as ONNX, TensorFlow and torchvision [54], which is a package that includes numerous datasets, models and such. All of them can be installed conveniently using pip.

```
# Load the model
model = torchvision.models.resnet50(pretrained=True)
# Dummy input
imagenet_input = Variable(torch.randn(1, 3, 224, 224))
# Export to ONNX
torch.onnx.export(model, imagenet_input, 'resnet50.onnx')
onnx_model = onnx.load('resnet50.onnx')
tf_rep = prepare(onnx_model)
```

Next we load the pretrained ResNet50 model using torchvisio and create a dummy image input. The Dummy input size is one of the standard ImageNet input sizes, ResNets in Pytorch takes 224x224 inputs. Then we export the model to ONNX format using torch.onnx.export [53]. The exporter runs the model once in order to get a trace of the execution of the model [53]. Now having the ONNX file saved on the disk, we load the file with onnx.load and then import the model to TensorFlow. The prepare function converts the ONNX model to an internal representation of the computational graph i.e., the function returns a TensorflowRep class object [35].

```
def preprocess(img_data):
```

```
    mean_vec = np.array([0.485, 0.456, 0.406])
    stddev_vec = np.array([0.229, 0.224, 0.225])
    norm_img_data = np.zeros(img_data.shape).astype('float32')
    for i in range(img_data.shape[0]):
         # for each pixel in each channel, divide the value by 255 to get value
             between [0, 1] and then normalize
        norm_img_data[i,:,:] = (img_data[i,:,:]/255 - mean_vec[i]) / stddev_vec[i]
    return norm_img_data
```

Before testing the model we must preprocess the input images, for this we can use the above script which forces the image data to a range of [0, 1] by dividing each pixel of each channel by 25. After this we normalize the data by subtracting the mean from the data and dividing by the standard deviation [33].

```
results = []
images = ['wine_bottle.jpg', 'carla_small.jpg']
for image_name in images:
    img = imageio.imread(image_name)
    # (H, W, D) => (D, H, W)
    img_swap_ax = np.swapaxes(img, 0, 2)
    # (D, H, W) => (N, D, H, W)
    img_expanded = np.expand_dims(img_swap_ax, axis=0)
    img_processed = preprocess(img_expanded)
    output = tf_rep.run(img_processed)
    output_list = np.asarray(output)[0][0].tolist()
    output_with_index = []
    for i in range(0, len(output_list)):
        output_with_index.append((output_list[i], i))
    output_sorted = sorted(output_with_index, key=lambda x: x[0])
    results.append(image_name+": "+str(output_sorted[999]))
for res in results:
    print(res)
```

Here we read the images we are going to classify, Figure 11 and Figure 12. Then we swap and expand the axes and feed the data to the preprocessing function defined above. Then we can perform inference by feeding the data to the model and printing the results of the inference.

wine_bottle.jpg (Figure 11): 14.333847999572754
carla_small.jpg (Figure 12): 6.421350002288818

Figure 12 is classified as a tennis ball and Figure 11 as wine bottle.

5 Discussion

The results of the inference were not surprising considering that the ImageNet dataset [1] that the ResNet50 was trained with does not include many classes related to humans and that the image used for inference is somewhat distorted due to scaling. The wine bottle was however correctly classified as a wine bottle as there exists a class for that in the ImageNet dataset.

Though ONNX shows great potential, there still remains many problems with it. One of them is the problem with rapidly changing operation sets and how the frameworks lag behind implementing their support for them. If you are in hurry, you will most likely end up digging through the development branches of the frameworks for operation set support. Many frameworks that support ONNX are still lacking many key features, such as PyTorch which does not have functionality for importing ONNX models [21]. Then some frameworks such as TensorFlow don't officially support ONNX which makes things more difficult. In many cases it can be more convenient to use converter e.g., one of [11].

References

[1] *1000 classes of ImageNet*. https://github.com/onnx/models/blob/master/vision/classification/synset.txt. Accessed: 29-08-2019.

[2] *About TVM*. https://tvm.ai/about. Accessed: 26-08-2019.

[3] John Arundel and Justin Domingus. *Cloud Native DevOps with Kubernetes*. 1st ed. O'Reilly Media, Mar. 2019, p. 9. ISBN: 978-1492040767.

[4] Y. Bengio, P. Simard, and P. Frasconi. "Learning Long-term Dependencies with Gradient Descent is Difficult". In: *Trans. Neur. Netw.* 5.2 (Mar. 1994), pp. 157–166. ISSN: 1045-9227. DOI: 10.1109/72.279181. URL: https://doi.org/10.1109/72.279181.

[5] Tianqi Chen et al. "TVM: End-to-End Optimization Stack for Deep Learning". In: *CoRR* abs/1802.04799 (2018). arXiv: 1802.04799. URL: http://arxiv.org/abs/1802.04799.

[6] *Common subexpression elimination*. https://en.wikipedia.org/wiki/Common_subexpression_elimination. Accessed: 25-08-2019.

[7] *Commutative property*. https://en.wikipedia.org/wiki/Commutative_property. Accessed: 22-08-2019.

[8] *CS231n Convolutional Neural Networks for Visual Recognition*. http://cs231n.github.io/convolutional-networks/. Accessed: 14-08-2019.

[9] Scott Cyphers et al. "Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning". In: *CoRR* abs/1801.08058 (2018). arXiv: 1801.08058. URL: http://arxiv.org/abs/1801.08058.

[10] Anton Debner. "Performance evaluation of a machine learning environment for intelligent transportation systems; Robottiautojen tutkimukseen tarkoitetun virtuaalisen koneoppimisympäristön suorituskyvyn evaluointi". en. G2 Pro gradu, diplomityö. 2019-06-17, p. 61. URL: http://urn.fi/URN:NBN:fi:aalto-201906234064.

[11] *Deep Learning Model Converter*. https://github.com/ysh329/deep-learning-model-convertor. Accessed: 30-08-2019.

[12] *Docker Hub Quickstart*. https://docs.docker.com/docker-hub/. Accessed: 05-08-2019.

[13] *Docker Machine Overview*. https://docs.docker.com/machine/overview/. Accessed: 05-08-2019.

[14] *Docker overview*. https://docs.docker.com/engine/docker-overview/. Accessed: 05-08-2019.

[15] *Docker Registry*. https://docs.docker.com/registry/. Accessed: 05-08-2019.

[16] Alexey Dosovitskiy et al. *CARLA: An Open Urban Driving Simulator*. 2017. arXiv: 1711.03938 [cs.LG].

[17] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[18] *Halide*. https://halide-lang.org/. Accessed: 28-08-2019.

[19] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Comput.* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: http://dx.doi.org/10.1162/neco.1997.9.8.1735.

[20] *ImageNet*. http://www.image-net.org/. Accessed: 29-08-2019.

[21] *Import ONNX model to Pytorch issue*. https://github.com/pytorch/pytorch/issues/21683. Accessed: 30-08-2019.

[22] *Intermediate representation*. https://en.wikipedia.org/wiki/Intermediate_representation. Accessed: 25-08-2019.

[23] *Introducing NNVM Compiler: A New Open End-to-End Compiler for AI Frameworks*. https://aws.amazon.com/blogs/machine-learning/introducing-nnvm-compiler-a-new-open-end-to-end-compiler-for-ai-frameworks/. Accessed: 26-08-2019.

[24] *Keras*. https://www.tensorflow.org/guide/keras. Accessed: 09-08-2019.

[25] *Kubernetes components*. https://kubernetes.io/docs/concepts/overview/components/. Accessed: 06-08-2019.

[26] *Kubernetes nodes*. https://kubernetes.io/docs/concepts/architecture/nodes/. Accessed: 06-08-2019.

[27] *Kubernetes pods*. https://kubernetes.io/docs/concepts/workloads/pods/pod/. Accessed: 06-08-2019.

[28] Marko Lukša. *Kubernetes in Action*. 1st ed. Manning Publications, Dec. 2017, pp. 1–2, 18–19. ISBN: 978-1617293726.

[29] Ian Miell and Aidan Hobson Sayers. *Docker in Practice*. 2nd ed. Manning Publications, Feb. 2019, pp. 6–7, 9–10. ISBN: 978-1617294808.

[30] *nGraph architecture*. https://www.ngraph.ai/architecture. Accessed: 12-09-2019.

[31] *NVIDIA DRIVE Constellation*. https://www.nvidia.com/en-us/self-driving-cars/drive-constellation/. Accessed: 09-09-2019.

[32] *ONNX documentation*. https://github.com/onnx/onnx/tree/master/docs. Accessed: 23-08-2019.

[33] *ONNX ResNett*. https://github.com/onnx/models/tree/master/vision/classification/resnet. Accessed: 29-08-2019.

[34] *ONNX website*. https://onnx.ai. Accessed: 23-08-2019.

[35] *onnx-tensorflow*. https://github.com/onnx/onnx-tensorflow/. Accessed: 29-08-2019.

[36] *PyTorch convolution layers*. https://pytorch.org/docs/stable/nn.html#convolution-layers. Accessed: 15-08-2019.

[37] Pierluigi Riti. *Pro DevOps with Google Cloud Platform. With Docker, Jenkins, and Kubernetes*. 1st ed. Apress, Oct. 2018, p. 79. ISBN: 978-1484238967.

[38] *Rombauer Chardonnay white wine*. https://www.totalwine.com/dynamic/490x/media/sys_master/twmmedia/hcd/h87/8796517138462.png. Accessed: 30-08-2019.

[39] Nadav Rotem et al. *Glow: Graph Lowering Compiler Techniques for Neural Networks*. 2018. arXiv: 1805.00907 [cs.PL].

[40] Sipi Seppälä. "Performance of Neural Network Image Classification on Mobile CPU and GPU; Kuvanluokittelija-neuroverkkojen suorituskyky mobiilisuorittimilla". en. G2 Pro gradu, diplomityö. 2018-05-14, p. 86. URL: http://urn.fi/URN:NBN:fi:aalto-201806012991.

[41] Nishant Shukla. *KMachine Learning with TensorFlow*. 1st ed. Manning Publications, Feb. 2018, p. 21. ISBN: 978-1617293870.

[42] William Stallings. *Operating Systems. Internals and Design Principles*. 9th ed. Pearson, Mar. 2017, pp. 635–642. ISBN: 978-0134670959.

[43] *TensorFlow Architecture*. https://www.tensorflow.org/guide/extend/architecture. Accessed: 09-08-2019.

[44] *TensorFlow convolution layers*. https://www.tensorflow.org/api_docs/python/tf/nn/convolution. Accessed: 15-08-2019.

[45] *TensorFlow Graph*. https://www.tensorflow.org/api_docs/python/tf/Graph. Accessed: 11-08-2019.

[46] *TensorFlow in other languages*. https://www.tensorflow.org/guide/extend/bindings. Accessed: 09-08-2019.

[47] *TensorFlow low-level introduction*. https://www.tensorflow.org/guide/low_level_intro. Accessed: 09-08-2019.

[48] *TensorFlow Session*. https://www.tensorflow.org/api_docs/python/tf/Session. Accessed: 11-08-2019.

[49] *TensorFlow Tensors*. https://www.tensorflow.org/api_docs/python/tf/Tensor. Accessed: 11-08-2019.

[50] *TensorFlow Tensors*. https://www.tensorflow.org/api_docs/python/tf/Operation. Accessed: 11-08-2019.

[51] *TensorFlow Tensors*. https://www.tensorflow.org/guide/tensors. Accessed: 09-08-2019.

[52] *tf.GraphDef*. https://www.tensorflow.org/api_docs/python/tf/GraphDef. Accessed: 11-08-2019.

[53] *torch.onnx*. https://pytorch.org/docs/stable/onnx.html#functions. Accessed: 29-08-2019.

[54] *TORCHVISION*. https://pytorch.org/docs/stable/torchvision/index.html. Accessed: 29-08-2019.

[55] *TVM documentation*. https://github.com/dmlc/tvm/blob/master/docs/faq.md. Accessed: 28-08-2019.

[56] *Understanding LSTMs*. http://colah.github.io/posts/2015-08-Understanding-LSTMs/. Accessed: 27-08-2019.

[57] Deepak Vohra. *Kubernetes Microservices with Docker*. 1st ed. Apress, Apr. 2016, p. 3. ISBN: 978-1484219065.

[58] *What is Kubernetes*. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. Accessed: 05-08-2019.

[59] *XLA Architecture*. https://www.tensorflow.org/xla/architecture. Accessed: 25-08-2019.

[60] *XLA: Optimizing Compiler for TensorFlow*. https://www.tensorflow.org/xla/. Accessed: 25-08-2019.

[61] Feng Zhang et al., eds. *Network and Parallel Computing - 15th IFIP WG 10.3 International Conference, NPC 2018, Muroran, Japan, November 29 - December 1, 2018, Proceedings*. Vol. 11276. Lecture Notes in Computer Science. Springer, 2018, pp. 41–43. ISBN: 978-3-030-05676-6. DOI: 10.1007/978-3-030-05677-3. URL: https://doi.org/10.1007/978-3-030-05677-3.

A Images used in the experiment

Figure 11: Wine bottle [38]

Figure 12: CARLA NPC