Deep Learning Frameworks

Leon Blumenthal
Technische Universität München
Munich, Germany
leon.blumenthal@tum.de

Jakob Taube
Technische Universität München
Munich, Germany
jakob.taube@tum.de

***Abstract*—Deep Learning is undeniably the current reality. It affects our daily life with prevalent results in voice assistants, image recognition, medical diagnosis, autonomous vehicles, and many more areas. In order to harness and promote deep learning, frameworks have evolved in recent years making it accessible to everyone. Therefore, we provide an overview of a selection of five frameworks, namely TensorFlow, PyTorch, Caffe, CNTK, and MXNet. In this publication, we highlight strengths and weaknesses of each one by conducting a comparative analysis in which we consider popularity, functionality, usability, performance, and some of the differences in their implementations. Finally, we finish by presenting a case-study featuring TensorFlow and PyTorch, the most popular frameworks, in a convolutional neural network implementation. Our work aims to enable readers to make a well-grounded decision about the most suitable framework for their needs.**

Keywords - Deep Learning, Framework, Neural Network, TensorFlow, PyTorch, Caffe, CNTK, MXNet

I. Introduction

Deep learning has revolutionized traditional approaches in machine learning. Instead of mapping inputs directly to outputs, it uses multiple hidden layers, thus creating a deep hierarchical architecture. These hidden layers add several levels of abstraction which allows the networks to recognize complex patterns in data and extract certain features. It has become especially famous in recent years due to the availability of powerful hardware, which is necessary for executing large networks in a reasonable time. Simultaneously, huge amounts of data have become available to train these networks. Deep learning algorithms have proven to deliver extremely promising results, hence outperforming alternative methods. However, implementing these concepts from scratch can quickly become very cumbersome. Therefore, suitable frameworks have been developed in order to provide building blocks for designing, training, and validating neural networks through high-level interfaces. Additionally, these frameworks offer a variety of pre-built and optimized components to define models quickly in a clear and concise way without the need to implement the underlying algorithms from the ground up.

In this paper, we present five popular frameworks: TensorFlow, PyTorch, Caffe, CNTK, and MXNet. The paper is organized as follows. In section II, we provide an introduction to each framework including its history, intrinsic design principles, basic implementation approaches, and real-world applications. Section III compares the popularity, functionality, and usability of the frameworks. Furthermore, it explains what effects different implementation methods have and how this affects the performance of the framework. It elaborates on the performance by reviewing previously published studies. Section IV then conducts a case-study implementing a neural network using TensorFlow and PyTorch. This is intended to provide insight into the programming interfaces of the two most employed frameworks in order to demonstrate how theoretical concepts are put into practice. Lastly, we conclude our work in section V.

II. Frameworks

A. TensorFlow

Released as an open-source package in November 2015 by Google Brain, TensorFlow [1] is a framework designed for various data processing tasks. However, use cases mainly focus on neural networks and artificial intelligence for which it provides a large selection of training and inference algorithms, automatic gradient computation, a broad range of neural network models, and additional tools like *TensorBoard*[1] for visualization purposes. Its design principles aim to support distributed execution and flexibility for research without compromising high-performance and robustness for the industry. Due to sufficient abstraction, it is also very portable and allows for it to be readily maintained. TensorFlow provides multiple front-end languages such as Python, C++, JavaScript, and Java while its core is written in C++.

It is important to recognize that since its initial release, TensorFlow underwent some major changes particularly in version 2.0 (issued in 2019) [2]. We therefore briefly cover its former implementation as described in [1] and follow up by explaining key changes of the new version, TensorFlow 2. In TensorFlow version 1.X, directed data flow graphs are used in order to describe computations. *Nodes* in the graph represent operations with an arbitrary number of inputs and outputs. *Edges*, on the other hand, describe the direction of data flow where units of data are called *tensors*. Here, tensors can be understood as multi-dimensional arrays. Additionally, some special nodes are provided:

- nodes with persistent, mutable state (also known as variables)
- control nodes for conditioning, looping, and ordering
- checkpoint and recovery nodes for saving values of variables during run time

[1] https://www.tensorflow.org/tensorboard

A small example of a computational graph can be seen in figure 1. We want to emphasize that, in TensorFlow, the graph is defined *symbolically*, meaning the client describes the structure of the graph in a declarative manner using symbolic handles.

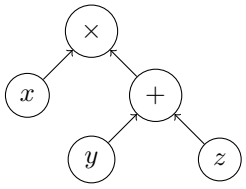

Fig. 1. An example of a computational graph with three inputs x, y, and z representing the calculation $x \times (y + z)$.

For the execution of the graph, TensorFlow version 1.X distinguishes between four groups. A *client program* interacts with the *master* of the TensorFlow system, that then coordinates one or more *workers*, which ultimately supervise several *devices* that carry out computations [3]. When executing the static computational graph, the system will first examine the graph and assign nodes to hardware via a placement algorithm while respecting the dependencies between nodes. The placement algorithm works with a cost function that considers

- the estimated input and output size of tensors
- the estimated static execution time/actual execution time of operations from earlier runs
- a heuristic for delays resulting from cross-device communication

Since the introduction of *eager execution* and the integration of Keras in TensorFlow 2.0, the framework follows quite a different approach. Keras is a high-level deep learning API written in Python that runs on top of other frameworks [4]. Developed by Google engineer François Chollet in 2015, it was originally designed to support multiple back ends [5]; now it goes hand in hand with TensorFlow. Its core structures are complete models and layers of neural networks. Details and examples can be seen in IV. As opposed to the declarative style of TensorFlow 1, eager execution now adds an imperative programming style to the framework [6]. Instead of building a computational graph, operations are evaluated immediately, returning concrete values instead of symbolic handles. Even though declarative programming is still possible, eager execution is the new default.

For optimization, TensorFlow uses lossy compression besides common sub-expression elimination and effective node scheduling. In lossy compression, data is compressed for transfer between devices in a way that accuracy is lost but the amount of data reduced. For example, a 32bit float is converted into a 16bit float for transfer and afterward, it is just filled up with zeros to obtain back a 32bit float. This works well since high precision is often not needed in neural networks and instead provides additional robustness.

TensorFlow is used in speech recognition, computer vision, robotics, computational drug discovery, and many more. Companies like Uber, Dropbox, and Airbnb make use of it [7].

B. PyTorch

PyTorch is an open-source machine learning framework created by Facebook AI Research Lab (FAIR) [8], [9]. The alpha version was released in September 2016 and version 1.0 followed two years later in December 2018 [10]. PyTorch's objective is to accelerate the path from research to production-ready results.

In order to achieve its main purpose, the authors outline four main design principles in a publication [11]:

- *Be Pythonic*: The framework should be used just like any other Python library.
- *Put researchers first*: The complexity is handled by the framework to facilitate the work and increase productivity.
- *Provide pragmatic performance*: The performance should be as good as possible without introducing too much complexity for the user.
- *Worse is better*: A slightly incomplete library with more features is better than a polished one with fewer features and a more complicated design.

PyTorch has many components [12] but the most notable ones are as follows:

- `torch`: A fully featured tensor library with strong GPU support.
- `torch.autograd`: A library which implements automatic gradient computation on tensors. The specific implementation of automatic differentiation is called "reverse-mode auto-differentiation"
- `torch.nn`: Fully integrating tensors and automatic differentiation, this library defines neural networks with layers, activation functions, loss functions.
- `torch.optim`: A package including many different optimizers and utilities for neural networks.
- `torchvision`: A library with many popular models, datasets, and image transformations.
- `torch.jit`: TorchScript is used to serialize models from Python code.

PyTorch also allows distributed training, deployment on mobile devices, and is supported by major cloud providers [13], [14].

PyTorch uses dynamic eager execution to behave like an iterative program [11]. In contrast to frameworks with static computation graphs, the graph is constructed during execution and always rebuilt for every iteration. This allows dynamically changing functions, arbitrary Python control flow structures, and standard functionality (e.g. print statements and debugging). Layers, models, optimizers, etc. are implemented as a class and can therefore be easily exchanged.

In order to provide similar performance to other frameworks and combat the challenges that arise from its dynamic view, PyTorch optimizes every step in its calculations by using additional strategies [11]. We will present the main ideas in the following. The performance-critical parts like tensors, parallelization, and automatic differentiation are implemented in C++, which is tightly integrated with the Python bindings,

therefore making it pythonic. Another possibility is to use TorchScript for deployment. This allows serializing the trained model and running it in a pure performance environment like C++. The control and data flow are separated in a way that allows the user to run the operations on the GPU to increase performance. To do this efficiently, reference counting and improved memory allocation are employed, and the Python multiprocessing library is extended to share tensors via working memory.

PyTorch is used by Tesla, Twitter, Salesforce, the University of Oxford, and many other institutions. [15].

C. Caffe

As one of the first deep learning libraries, Caffe [16] was introduced in 2014 by Yangqing Jia, of Berkeley AI Research. He created this project while working on his Ph.D. [17]. The main purpose was to streamline and unify research and development. It was first designed for computer vision (i.e. CNNs - convolutional neural networks) but was later adopted in other areas. Version 1.0 was released in April 2017 which was also the last release because the project was discontinued [18]. Its successor Caffe2, developed by FAIR, was then merged into PyTorch in 2018 for the purpose of minimizing overhead for the Python community and combining the best out of the frameworks [19].

In the following, we will summarize the core features of Caffe based on the initial paper [16]. The key design principle was the separation of model representation and actual implementation. The framework offers Python and Matlab bindings for fast prototyping, but the main logic is implemented in C++ with strong support for GPUs. In contrast, the model structure which can be any directed acyclic graph (DAG), is defined in a config using the Protocol Buffer language before runtime. This allows for an increase in production efficiency, easier replication of research results, and exact memory allocation that is independent of CPU or GPU execution. Deployment of deep learning models was enhanced through improved computational efficiency compared to previous methods. Batches of images, parameters, and other data are stored in 4-dimensional arrays called blobs, which are then saved on disk as Google Protocol Buffers. Layers take blobs as input and produce blobs as output in the forward pass and compute gradients in the backward pass. Many layers and functions are already implemented, but it is also easy to add new functionality due to the modularity of the library. Additionally, many pre-trained models are provided for academic use.

D. CNTK

The Microsoft Cognitive Toolkit (formerly known as the **C**omputational **N**etwork **T**ool**k**it) [20] was released as an open-source project by Microsoft Research in 2016. As of 2019, it is no longer actively developed [21]. According to the developers, it is entirely designed around efficiency. Moreover, it is intended for business-related applications [22]. It features many types of neural networks and algorithms for training and testing those networks [23]. Its core is implemented in C++, and neural networks are programmed via either a text configuration file using BrainScript or other supported languages(e.g. Python, C#, and Java)

We will focus on the implementation as described in [23] using the BrainScript API[2]. CNTK, similar to other frameworks, uses a computation graph which is to be understood as a series of computations. The framework does not provide a concept for layers, as it does to single nodes. This approach is more burdensome but makes elaborate adjustments possible. A program implementing a neural network can be divided into different command blocks offering various parameters for detailed specifications. Among the most common blocks are:

- Network Builders to specify whether a predefined network should be loaded using the `SimpleNetWorkBuilder` or a new network is to be defined from ground up using the `BrainScriptNetworkBuilder`
- a Learner indicating what training algorithm to use (usually the stochastic gradient descent using the `SGD` block)
- Data Readers like the `CNTKTextFormatReader` to specify the format of input data and how to read it

Upon execution, the graph will be transformed into a DAG, and the order of calculations determined via a depth-first search algorithm. In addition to extensive numerical optimizations, CNTK removes duplicated computations in the graph, implements batch processing, performs factorization of common prefix terms and parallelization techniques, resulting in further speed-up.

CNTK is used for speech, image, and text processing. It is essential to many Microsoft products including Cortana, Xbox, and Skype [15].

E. MXNet

MXNet was published by the Apache Software Foundation in 2015 [24]. It is a vastly portable, lightweight, open-source framework for deep learning. Its name MXNet ("mix-net") originates from the idea to mix the declarative symbolic and the imperative programming styles in order to obtain the benefits from both. Its back end is written in C++, and it supports many programming languages in the front end such as Python, Java, Julia, Scala, and R. Despite that, the main focus lies on the Gluon Python API which is an imperative programming interface but allows the ability to switch to symbolic mode at any time. Therefore, the main idea is to develop a network using imperative programming for easier debugging and then converting it to symbolic expressions for optimization. MXNet features multiple common forward evaluation and backward propagation algorithms, different model and layer types, and distributed execution.

For the implementation, we refer to [24] as a basis, but consult [25] for up-to-date documentation. MXNet adopts the basic idea of a computation graph composed of operators. It provides both an imperative implementation of

[2]https://docs.microsoft.com/en-us/cognitive-toolkit/BrainScript-Basic-Concepts

tensor operations via the `NDArray` (for asynchronous n-dimensional arrays) module and a symbolic implementation via the `Symbol` module. For distributed execution, it implements the `KVStore` key-value store which is responsible for parameter synchronization across multiple devices.

On the graph several optimizations can be performed. According to [24], these include:

- grouping operators into single GPU calls
- memory allocation heuristics for reuse or sharing
- effective scheduling

Some applications of MXNet are in handwriting recognition, natural language processing, and forecasting. It is used by large companies such as Microsoft, Intel, and Amazon. [15].

III. Comparison

After having looked at different frameworks individually, we now want to put them into comparison so that strengths and weaknesses become more tangible. In order to do so, we will evaluate the frameworks discussed in section II under the aspects of popularity, functionality, implementation, performance, and usability.

A. Popularity

While all frameworks enjoy a certain popularity in their communities, significant differences do occur between them. Table II shows the number of stars, the number of users, and the number of people who contribute to the repository of each framework on GitHub. Undoubtedly TensorFlow is the most popular framework. Its numbers are double or triple the size of its closest competitor PyTorch, thus making it the most used framework.

TABLE II
Popularity on GitHub[a]

Framework	Stars	Used by	Contributors
TensorFlow	151k	108k	2813
PyTorch	44.5k	51.3k	1685
Caffe	31.1k	-	269
MXNet	19.2k	1.6k	847
CNTK	16.9k	-	202

[a]numbers from github.com (accessed 2020-12-06)

However, in recent months, PyTorch has experienced an increasing interest among developers, especially in the field of research as it enables fast prototyping [26]. As a result, it is the second most popular framework on our list. Figure 2 suggests that CNTK, MXNet, and Caffe, which was previously the second most common framework, were replaced by TensorFlow and PyTorch. The apparent decline of interest in frameworks in general since summer 2020 might possibly be a consequence of the COVID-19 pandemic.

B. Functionality

In their core, all frameworks provide very similar functionality. They feature common algorithms for forward evaluation and backward propagation, common models for neural networks like fully connected neural networks (FCN), convolutional neural networks (CNN) or recurrent neural networks (RNN), and well-known labeled data sets for training and testing like the MNIST data set of handwritten digits or the ImageNet data set with images of various objects and animals. A feature like distributed training is also something, most of the frameworks provide. Unsurprisingly, popular frameworks like TensorFlow and PyTorch provide a broader range of tools and extensions that add further functionality.

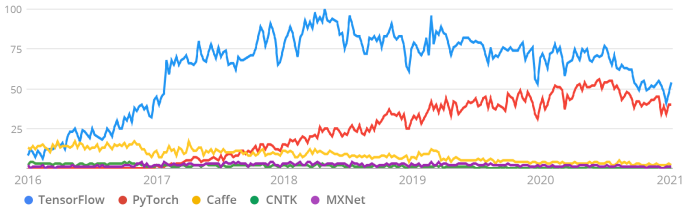

Fig. 2. A timeline showing the relative number of web searches on Google of the frameworks from January, 2016 until today. (from trends.google.com accessed 2021-01-17)

C. Implementation

Common ground in most of the frameworks is the idea of a computational graph. The construction, however, may differ from the ground up. This brings us to the declarative versus imperative programming style. Even though declarative frameworks were popular in earlier years (see II and III-A), the imperative style for deep learning became more and more predominant. The declarative style expresses computations by saying *what* is to be computed. This is achieved by describing the structure of the graph, thereby offering more optimization opportunities using typical techniques presented in II. In contrast, the imperative style describes computations by saying *how* something is to be computed. This is achieved by defining concrete computations. Thus, a more flexible and intuitive interface is provided, control flow is more natural, and debugging can be done interactively [6]. After all, it really comes to personal preference and sometimes frameworks even try to implement both (e.g. MXNet and TensorFlow).

D. Performance

In order to evaluate the performance of our frameworks in terms of speed, hardware utilization, and memory consumption we will review previously published comparisons and present relevant results.

In 2016, Shi [27] tested TensorFlow, Caffe, CNTK, MXNet, and Torch (not PyTorch) on synthetic data sets (e.g. ImageNet) for execution time measurements and on real-world data sets (e.g. MNIST) for convergence rates using CPU and GPU environments. The paper concludes, that the usage of GPU environments leads to significantly faster execution times. In the tests, multi-GPU setups gave better convergence rates over a single GPU with the best results by MXNet and CNTK. Compared to the other frameworks, TensorFlow achieved the best results on a CPU, especially when the number of threads

TABLE I
COMPARATIVE OVERVIEW

	front end languages	style	distributed execution	open-source	GPU support	actively developed	pros	cons
TensorFlow	C++, Python, JavaScript, R, Go, Swift, Java	hybrid	✓	✓	✓	✓	flexibility, portability, good support, user-friendly, fast prototyping, many features	slower than other frameworks
PyTorch	C++, Python, Java	imperative	✓	✓	✓	✓	user-friendly, fast prototyping, good support, flexibility, many features	slower than other frameworks
Caffe	C++, Python, Protobuf, Matlab	declarative	✗	✓	✓	✗	fast prototyping, good performance, modularity	poor flexibility, low customization
MXNet	C++, Python, R, Scala, Julia, Perl, Closure, Java	hybrid	✓	✓	✓	✓	lightweight, efficient, good performance, scalability, portability	limited support
CNTK	C++, Python, BrainScript, C#, Java	declarative	✓	✓	✓	✗	excellent performance, scalable, highly customizable	limited support, high learning curve

and cores were scaled up, whereas Caffe dominated on few CPU threads. On a GPU, CNTK was clearly the fastest. For FCN and CNN, CNTK and Caffe were faster than MXNet, followed by TensorFlow. For RNN, CNTK turned out to be many times faster than its competitors.

Chintala [28] used native versions of some deep learning frameworks and tested their execution times on a variety of CNN types running on a single GPU. In his results, TensorFlow achieved rather good results compared to Caffe or other frameworks. However, he urges the reader not to jump to conclusions since it was benchmarked with CuDNN, possibly making TensorFlow look faster than it is.

In 2020, Al-Bdour [29] compared TensorFlow, CNTK, and Theano in terms of running time, memory consumption, and hardware utilization. The paper confirms some results already found by Shi [27]. In the tests, both CNTK and TensorFlow showed high CPU and GPU Utilization. However, TensorFlow had higher memory usage and needed more epochs to reach the same accuracy as CNTK. Table III shows an excerpt from the test results.

TABLE III
PERFORMANCE METRICS ON THE MNIST DATASET[a]

Metric	Enviromment	CNTK	TensorFlow
Accuracy%	CPU	99.27	99.10
	GPU	99.26	99.11
Utilization%	CPU	99.6	92.2
	GPU	92	77
Memory%	CPU	1.7	2.2
	GPU	3.6	5.2
Epochs#	CPU	7	15
	GPU	7	15

[a]numbers as provided in [29]

Measuring the performance of different frameworks is important for comparisons and conclusions. For example, we could conclude that TensorFlow's inferior performance on GPUs compared to CPUs relative to other frameworks could be caused by its high memory consumption while memory on GPUs is scarce. However, it must be ensured that testing conditions are fair since performance strongly depends on the network architecture, its size, and the hardware used. Moreover, versions of underlying dependencies (e.g. cuDNN) can result in performance discrepancies as proposed by Bahrampour [30]. Last but not least, performance varies depending on the size and type of data set used.

E. Usability

In general, the imperative style interfaces of Caffe, PyTorch, or TensorFlow 2 seem to be easier to use than the declarative interfaces of TensorFlow 1 and CNTK. Bahrampour [30] explicitly praises the ease of use of Caffe. In addition, especially TensorFlow and PyTorch, the most popular frameworks, provide well designed and detailed documentation. Moreover, they offer a rich collection of guides and online tutorials. Unsurprisingly, many institutions use them for their courses (e.g. TensorFlow in MIT 6.S191 Introduction to Deep Learning[3]). Without a doubt, their users benefit from the abundant support provided by the large communities reinforcing these frameworks.

F. Comparative Overview

We have studied different aspects of the frameworks in our previous paragraphs in a detailed fashion. In this paragraph, we provide an overview of the most crucial points, a user would want to be informed of, at a glance. Therefore, the framework's essential properties are depicted in table I, sorted by the popularity of the frameworks from top to bottom.

[3]http://introtodeeplearning.com/

IV. Case Study: Convolutional Neural Network

As pointed out in section III-A, PyTorch and TensorFlow are the most popular frameworks. In this section, we will show with a practical example involving image classification with CNN how to actually use them. For that, we go through every step of a possible solution and compare the Python implementations for both frameworks. The example implementations are based on official guides and tutorials [12], [31]–[33]. It is important to point out that we chose the simplest variant and do not try to achieve good performances but rather showcase the functionality. Due to space constraints, we only show a selection of code snippets that sometimes are simplified for visualization purposes. The complete code can be found in our GitHub repository [34].

A. Data Loading and Pre-processing

We chose the popular MNIST database [35] which consists of 60000/10000 labeled grayscale training/test images of single handwritten digits with a resolution of 28x28 and pixel values in the interval [0, 255]. Before using the images, we want to map the pixel values onto the interval [0,1] and standardize them afterward (i.e. mean=0, std=1). Furthermore, we shuffle and batch the training data.

```
### PyTorch ###
# Load data and standardize images.
transform = Compose(
  [ToTensor(), Normalize((0.1307,), (0.3081,))]
)
train_data = datasets.MNIST(train=True, transform)
test_data = datasets.MNIST(train=False, transform)
# Shuffle images and split into mini-batches.
train_loader = DataLoader(
  train_data, BATCH_SIZE, shuffle=True
)
test_loader = DataLoader(test_datas, BATCH_SIZE)

### TensorFlow ###
train_data, test_data = datasets.mnist.load_data()
# Transform grayscale values and add depth dimension.
train_images = train_data[0][..., tf.newaxis] / 255
test_images = test_data[0][..., tf.newaxis] / 255
# Standardize images.
train_images = per_image_standardization(train_images)
test_images = per_image_standardization(test_images)
```

Both frameworks include the functionality to access popular datasets like MNIST and allow efficient pre-processing. In TensorFlow, we do not shuffle and batch the data at this point because it is more convenient at a later step. However, it looks very similar.

B. Network Structure

Next, we need to define the network structure. In this example, we use a CNN as depicted in figure 3. It shows the network from input image to final digit prediction. The feature maps are created using three convolutional layers with pooling layers in between. The final feature map is flattened and fed through two fully connected layers. After the second pooling layer and the first dense layer, we also include *dropout layers*. As the activation functions for the convolutional layers, we use *ReLu*. Then after the first dense layer, *tanh* is applied. Finally, we use *softmax* for the final output.

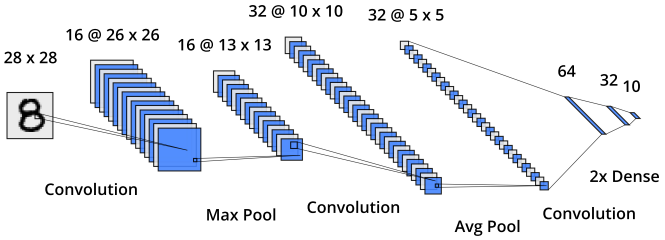

Fig. 3. CNN structure

```
### TensorFlow ###
net = models.Sequential([
  layers.Conv2D(16, 3, activation='relu'),
  layers.MaxPool2D(2),
  layers.Conv2D(32, 4, activation='relu'),
  layers.AvgPool2D(2),
  layers.Dropout(0.3),
  layers.Conv2D(64, 5, activation='relu'),
  layers.Flatten(),
  layers.Dense(32, activation='tanh'),
  layers.Dropout(0.2),
  layers.Dense(10),
  layers.Softmax()
])
```

Here, we only show the TensorFlow variant because defining the structure using the `Sequential` model looks nearly identical in PyTorch. The only difference is that activation functions cannot be specified directly as part of a layer but need to be defined as a separate layer (e.g. `nn.ReLU()`). The sequential structure is the least verbose variant to implement this network structure. Both frameworks also offer other forms of defining networks, which we show in the complete implementation.

C. Training Parameters

Now, we want to optimize the parameters of the network based on cross-entropy loss with stochastic gradient descent, momentum, and a decaying learning rate.

```
### PyTorch ###
optimizer = optim.SGD(net.parameters(), LR, MOMENTUM)
scheduler = StepLR(optimizer, 1, LR_DECAY)
loss_function = nn.CrossEntropyLoss()

### TensorFlow ###
scheduler = schedules.ExponentialDecay(
  LR, 6000, LR_DECAY, staircase=True
)
optimizer = optimizers.SGD(scheduler, MOMENTUM)
loss_function = losses.SparseCategoricalCrossentropy()
```

Both frameworks offer the same functionality here and only differ in minor aspects. Many other loss functions, optimizers, and schedulers can be found in each framework.

D. Training

As common in deep learning, we use training data and separate test data to evaluate the accuracy of our network. Therefore, we train the network for a fixed number of epochs on the training data and evaluate the prediction results in every iteration using the test data.

```
### PyTorch ###
# Choose GPU if available.
device = torch.device(
  "cuda:0" if torch.cuda.is_available() else "cpu"
)
net.to(device)

def train():
  net.train()  # training mode.

  # Iterate through all mini-batches.
  for images, labels in train_loader:
    # Move tensors to GPU if available.
    images = images.to(device)
    labels = labels.to(device)

    # Compute loss based on outputs and labels.
    loss = loss_function(net(images), labels)
    # Compute gradients and update parameters.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    # Print progress ...

  scheduler.step()

def test():
  net.eval()  # test mode.

  # Iterate through mini-batches ignoring gradients.
  with torch.no_grad():
    for images, labels in test_loader:
      # Move tensors to GPU if available.
      images = images.to(device)
      labels = labels.to(device)

      # Compute guesses based on outputs and labels.
      guesses = net(images).argmax(dim=1)

      # Print progress ...

# Actual training and testing
for epoch in range(EPOCHS):
  train()
  test()


### TensorFlow ###
net.compile(
  optimizer, loss_function, metrics='accuracy'
)
net.fit(
  x=train_images,
  y=train_labels,
  epochs=EPOCHS,
  batch_size=BATCH_SIZE,
  shuffle=True,
  validation_data=(test_images, test_labels),
)
```

This is the first time we can see a major difference. In contrast to PyTorch, TensorFlow has built-in functions to perform the standard training and evaluation loop. With the `metrics` parameter of the `compile` function, we can easily select which aspects should be evaluated and printed in each iteration. In PyTorch, we have to implement this manually. In the end, TensorFlow supports manual training loops as well. We show how this feature can be used for customization purposes in our complete implementation. Another difference is GPU usage. While TensorFlow automatically uses a GPU if available, in PyTorch we have to manually move the network and tensors onto it.

E. Model Saving

After training, we want to save the parameters of the network for later usage.

```
### PyTorch ###
torch.save(net.state_dict(), 'pt_net')

### TensorFlow ###
net.save_weights('tf_net')
```

Saving the parameters can be easily achieved in both frameworks. If needed, both also allow to save the entire model including its structure.

F. Wrap-up

In subsections IV-A to IV-E, we demonstrated how to implement a simple CNN for image classification in TensorFlow and PyTorch. Therefore, we explained how to first load and pre-process the input data. Afterward, we defined the network structure and set training parameters. These steps were very similar in both frameworks. We continued with the actual training loop, where the first noteworthy difference appeared: PyTorch does not offer a pre-built training and evaluation loop whereas TensorFlow does. As a last step, we stored the trained network parameters. In conclusion, both frameworks allow an easy and straight-forward implementation of our example.

V. Conclusion

In this paper, we presented five deep learning frameworks: TensorFlow, PyTorch, Caffe, CNTK, and MXNet. After giving an overview, we evaluated different aspects of the frameworks. On a single GPU, CNTK proved to be the fastest framework followed by Caffe and MXNet, whereas TensorFlow and PyTorch are usually slightly slower. In return, they allow for rapid and easy development due to the well-designed interfaces and a plethora of additional resources including documentation, guides, forums, and examples. This is why they are currently the most popular frameworks. Therefore, we took a closer look into their similarities and differences in a subsequent case-study. It shows that both frameworks are greatly alike, offering similar modules for fast prototyping and still allowing for complex customization.

References

[1] Abadi et al., "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," 2015.

[2] TensorFlow Team, "Tensorflow 2.0 is now available!" [Online]. Available: https://blog.tensorflow.org/2019/09/tensorflow-20-is-now-available.html, (accessed: 2020-12-05).

[3] P. Goldsborough, "A tour of tensorflow," 2016. [Online]. Available: http://arxiv.org/abs/1610.01178

[4] "About keras," [Online]. Available: https://keras.io/about/, (accessed: 2020-12-05).

[5] "Keras: Deep learning for humans," [Online]. Available: https://github.com/keras-team/keras, (accessed: 2020-12-05).

[6] TensorFlow Team, "Eager execution," [Online]. Available: https://www.tensorflow.org/guide/eager, (accessed: 2020-12-05).

[7] "A detailed comparison of the popular deep learning frameworks," [Online]. Available: https://medium.com/manishmshiva/a-detailed-comparison-of-the-popular-deep-learning-frameworks-a0f65fddf276, 2020, (accessed: 2020-12-06).

[8] Facebook AI Research, "Pytorch," [Online]. Available: https://ai.facebook.com/tools/pytorch, (accessed: 2020-12-05).

[9] Facebook, Inc., "Pytorch," [Online]. Available: https://pytorch.org/, (accessed: 2020-12-05).

[10] [Online]. Available: https://github.com/pytorch/pytorch/releases, (accessed: 2020-12-05).

[11] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," 2019.

[12] [Online]. Available: https://pytorch.org/docs/stable/index.html, (accessed: 2020-12-06).

[13] Facebook, Inc., [Online]. Available: https://pytorch.org/features/, (accessed: 2020-12-05).

[14] ——, [Online]. Available: https://pytorch.org/get-started/cloud-partners/, (accessed: 2020-12-05).

[15] M. Opala, "Deep learning frameworks comparison – tensorflow, pytorch, keras, mxnet, the microsoft cognitive toolkit, caffe, deeplearning4j, chainer," [Online]. Available: https://www.netguru.com/blog/deep-learning-frameworks-comparison, 2019, (accessed: 2020-12-05).

[16] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[17] Berkeley AI Research, "Caffe — deep learning framework," [Online]. Available: https://caffe.berkeleyvision.org/, (accessed: 2020-12-05).

[18] ——, "Release 1.0 - bvlc/caffe," [Online]. Available: https://github.com/BVLC/caffe/releases/tag/1.0, (accessed: 2020-12-05).

[19] Facebook Open Source, "Caffe2 and pytorch join forces to create a research + production platform pytorch 1.0," [Online]. Available: https://caffe2.ai/blog/2018/05/02/Caffe2_PyTorch_1_0.html, (accessed: 2020-12-05).

[20] Seide et al., "Cntk: Microsoft's open source deep-learning toolkit," 2016, white Paper.

[21] CNTK Team, "Cntk v2.7 release notes," [Online]. Available: https://docs.microsoft.com/en-us/cognitive-toolkit/releasenotes/cntk_2_7_release_notes, 2019, (accessed: 2020-12-05).

[22] ——, "The microsoft cognitive toolkit," [Online]. Available: https://docs.microsoft.com/en-us/cognitive-toolkit/, (accessed: 2020-12-05).

[23] Agarwal et al., *An Introduction to Computational Networks and the Computational Network Toolkit*. [Online]. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2014/08/CNTKBook-20160217.pdf

[24] Chen et al., "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," 2015. [Online]. Available: https://arxiv.org/abs/1512.01274

[25] Apache Software Foundation, [Online]. Available: https://mxnet.apache.org/versions/1.7.0/, (accessed: 2020-12-05).

[26] J. Gupta, "Top 5 deep learning frameworks for developers that are most popular in 2020," [Online]. Available: https://www.quytech.com/blog/top-5-deep-learning-frameworks/, (accessed: 2020-12-06).

[27] S. Shi, Q. Wang, P. Xu, and X. Chu, "Benchmarking state-of-the-art deep learning software tools," *CoRR*, vol. abs/1608.07249, 2016. [Online]. Available: http://arxiv.org/abs/1608.07249

[28] S. Chintala, "convnet-benchmarks," [Online]. Available: https://github.com/soumith/convnet-benchmarks, (accessed: 2020-12-05).

[29] G. Al-Bdour, R. Al-Qurran, M. Al-Ayyoub, and A. Shatnawi, "A detailed comparative study of open source deep learning frameworks," *CoRR*, vol. abs/1903.00102, 2019. [Online]. Available: http://arxiv.org/abs/1903.00102

[30] S. Bahrampour, N. Ramakrishnan, L. Schott, and M. Shah, "Comparative study of caffe, neon, theano, and torch for deep learning," *CoRR*, vol. abs/1511.06435, 2015. [Online]. Available: http://arxiv.org/abs/1511.06435

[31] [Online]. Available: https://pytorch.org/tutorials/, (accessed: 2020-12-06).

[32] [Online]. Available: https://www.tensorflow.org/guide, (accessed: 2020-12-06).

[33] [Online]. Available: https://www.tensorflow.org/tutorials/, (accessed: 2020-12-06).

[34] L. Blumenthal and J. Taube, "Deep learning frameworks - case study: Convolutional neural network," [Online]. Available: https://github.com/leonblumenthal/deep_learning_frameworks_cnn, (accessed: 2020-12-06).

[35] Y. LeCun, C. Cortes, and C. J. Burges, "The mnist database of handwritten digits," [Online]. Available: http://yann.lecun.com/exdb/mnist/, (accessed: 2020-12-06).