Automating AI lifecycle: the DDoS use case
Research Project

Eros Zaupa

1. Introduction

Distributed Denial of Service (DDoS) attack is a relatively simple but very serious threat to Internet security. A successful attack deprives legitimate users of using web-based services by a large number of compromised machines. Attacks can be implemented in network, transport and application layers with different protocols (e.g. TCP, UDP, ICMP, HTTP). Several detection and defence techniques are available, each with different problems and shortcomings. Considering *Machine Learning* (ML) techniques, no prior knowledge of data distribution is required but defining the best feature-set is one of the main concerns. Also a typical ML workflow can be quite demanding: data preprocessing, training and testing of the model are costly operations in terms of hardware requirements and time.

In this context, our project has two main goals

- Study and develop a ML solution by relying on a new dataset for evaluation of algorithms and systems on DDoS attacks, namely *CICDDoS2019* [1]. The dataset is publicly available, a detailed showcase of its features and taxonomy can be found on the website of the University of New Brunswick[1].
- Deploy the previous solution to test the AI orchestration capabilities provided by Kubeflow v1.0[2] over the DiVINE[3] facilities. Kubeflow is a ML toolkit that uses Kubernetes[4] to make ML deployment simple, portable and scalable.

2. ANN classifier

2.1. Dataset

For a detailed description of the dataset see the dedicated page, below some considerations

- Traffic traces are provided with more than 80 features, but the paper shows also how only a subset of them is significant for the classification task using various ML implementations. We too rely on this choice of features for our model.

1. https://www.unb.ca/cic/datasets/ddos-2019.html
2. https://www.kubeflow.org/
3. https://divine.fbk.eu/
4. https://kubernetes.io/

- As shown in Figure 1, the test dataset is highly unbalanced and contains only a subset of the labels provided in the training dataset. As a result, our model will only be trained over these labels.

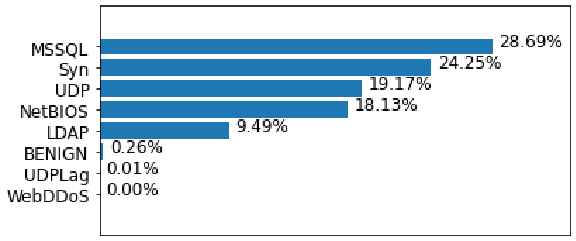


Figure 1. Percentage of labels over the traces in the test dataset

2.2. Model

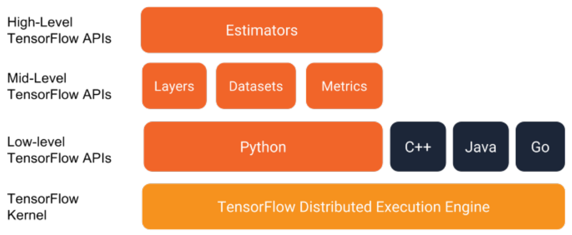

High-Level TensorFlow APIs
Estimators
Mid-Level TensorFlow APIs
Layers
Datasets
Metrics
Low-level TensorFlow APIs
Python
C++
Java
Go
TensorFlow Kernel
TensorFlow Distributed Execution Engine

Figure 2. TensorFlow provides a programming stack consisting of multiple API layers. The higher levels of abstraction are easier to use but are also (by design) less flexible.

The model is developed using TensorFlow 2.1.0[5] (TF2). The framework provides tf.estimator—a high-level TensorFlow API, with Estimators encapsulating training, evaluation, prediction and export for serving. An Estimator is any class derived from tf.estimator.Estimator.

TensorFlow provides a collection of pre-made Estimators (for example LinearRegressor) to implement common Machine Learning algorithms. These pre-implemented models allow quickly creating new models as need by customizing them. To develop our model we used the DNNClassifier pre-made estimator for our multiclassification tasks, providing both:

5. https://www.tensorflow.org/versions/r2.1/api_docs/python/tf

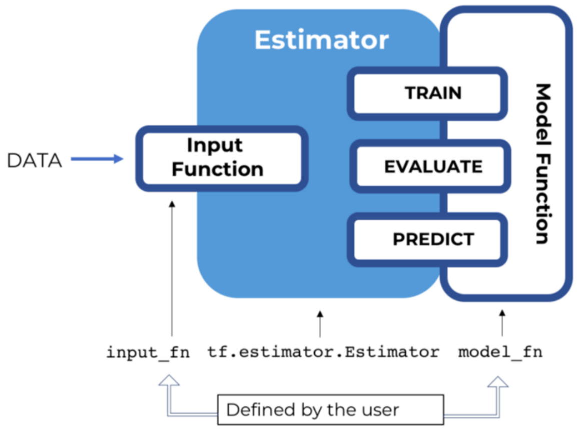


Figure 3. A schematic of Estimator

- A model function *model_fn* that is fed the features and labels and a few other parameters where the model code processes them and produces losses, metrics etc. The model function defines model, loss, optimizer, and metrics.
  - *Batch normalization* is applied after each layer to reduce the amount by what the hidden unit values shifts (covariance shift).
  - *Adam optimization algorithm* is an extension to stochastic gradient descent combining the best properties of the AdaGrad and RMSProp algorithms to provide an optimization algorithm that can handle sparse gradients on noisy problems.

```python
classifier = tf.estimator.DNNClassifier(
      hidden_units=[60, 30, 20],
      feature_columns=feature_columns,
      n_classes=len(labels),
      label_vocabulary=labels,
      batch_norm=True,
      optimizer=lambda: tf.keras.optimizers.Adam
  (
          learning_rate=tf.compat.v1.train.
  exponential_decay(
              learning_rate=0.1,
              global_step=tf.compat.v1.train.
  get_global_step(),
              decay_steps=10000,
              decay_rate=0.96)
      )
)

```

Listing 1. How our model is defined

- The input function *input_fn* to feed the Estimators that returns features and labels for train and evaluation.

```python
def input_fn(df, training, batch_size=32):
  '''
  An input function for training or evaluating
  '''
    # Convert the inputs to a Dataset
    dataset = tf.data.Dataset.from_tensor_slices((
    dict(df["features"]), df["labels"]))
    # Shuffle and repeat if you are in training
    mode
    if training:
      dataset = dataset.shuffle(1000).repeat()
    return dataset.batch(batch_size)
```

Listing 2. The function used to feed the Estimator

The resulting model is a dense, feed-forward neural network that supports all the labels previously listed for the test dataset. Hyper-parameters tuning is executed over two critical settings

- *Dropout rate*, the probability we will drop out a given coordinate. Randomly selected neurons are ignored during training, and this means that their contribution to the activation of downstream neurons is temporally removed on the forward pass and any weight updates are not applied to the neuron on the backward pass. The effect is that the network becomes less sensitive to the specific weights of neurons, resulting in a network that is capable of better generalization and is less likely to overfit the training data.
- *Learning rate*, speed at which the model learns. Specifically, it controls the amount of apportioned error that the weights of the model are updated with each time they are updated, such as at the end of each batch of training examples.
- *Number of hidden units* in the DNN

For the hyper-parameter tuning we use the holdout method with a common split using 80% of data for training and the remaining 20% of the data for evaluation.

2.3. Evaluation

Several metrics can be used to measure a model's performance in machine learning. *Accuracy* (Ac) is computed as the proportion of correct predictions for positive (TP) and negative class (TN) over the total number of observations.

$$Ac = \frac{TP + TN}{Total} \tag{1}$$

However, we need to be aware of the *"Accuracy Paradox"*: always predicting the majority class (e.g. benign traffic in a real case scenario) can in fact result in a high accuracy rate. Classification accuracy alone is usually not enough to evaluate a model, so we have to compute also the following metrics

- *Precision* (Pr), the ratio of correctly classified attacks flows (TP), in front of all the classified flows (TP+FP).
- *Recall* (Rc), the ratio of correctly classified attack flows (TP), in front of all generated flows (TP+FN).
- *F1 Score* (F1), the harmonic combination of the precision and recall into a single measure.

$$Pr = \frac{TP}{TP+FP}, Rc = \frac{TP}{TP+FN}, F1 = \frac{2*(Pr*Rc)}{Pr+Rc} \tag{2}$$

```
              precision  recall  f1-score  support

BENIGN        0.00       0.00    0.00            1
LDAP          0.94       0.96    0.95           96
MSSQL         0.91       0.94    0.93          270
NetBIOS       1.00       0.97    0.99          183
Syn           1.00       0.96    0.98          243
UDP           0.93       0.92    0.92          207
UDPLag        0.00       0.00    0.00            0

accuracy                         0.95         1000
macro avg     0.68       0.68    0.68         1000
weight. avg   0.95       0.95    0.95         1000
```

Listing 3. Example of a typical report executed over 1000 samples

3. Kubeflow

3.1. Resources

Kubeflow requires a Kubernetes cluster to run, access to an OpenStack dashboard has been granted to setup the following virtual machines

- *kubeflow-master*, with 4 VCPUs, 8GB RAM, 100GB of storage
- *kubeflow-slave-cloud-1*, with 4 VCPUs, 16GB RAM, 80GB of storage
- *kubeflow-slave-cloud-2*, with 4 VCPUs, 16GB RAM, 80GB of storage

All the machines run Ubuntu 16.04 LTS, Kubernetes v1.15.3 and Kubeflow v1.0. While the installation of Kubernetes was straightforward, some issues related to the storage systems and configuration emerged during the Kubeflow installation.

3.2. Dashboard

The master node hosts the Kubeflow Dashboard, displaying one of the possibly many workspaces. For our work we focused on the Pipeline section, that includes

- *Pipelines* page, that allows to manage existing pipelines or upload a new one
- *Experiments* page, that allows to manage past runs or trigger a new run

3.3. Pipelines

Kubeflow Pipelines is a platform for building and deploying portable, scalable machine learning (ML) workflows based on Docker containers.

- A *pipeline* is a description of an ML workflow, including all of the components in the workflow and how they combine in the form of a graph. The pipeline includes the definition of the inputs (parameters) required to run the pipeline and the inputs and outputs of each component. After developing a pipeline, it can be uploaded and shared on the Kubeflow Pipelines UI.
- A *pipeline component* is a self-contained set of user code, packaged as a Docker image, that performs one step in the pipeline. For example, a component can be responsible for data preprocessing, data transformation, model training, and so on.

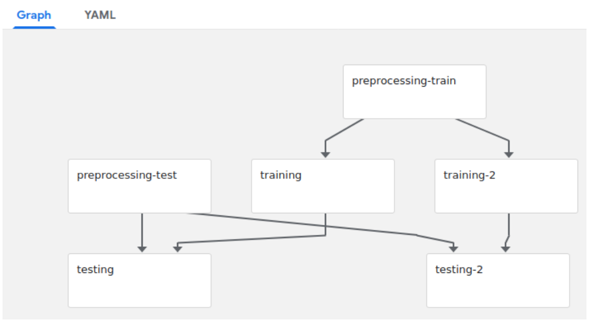


Figure 4. Pipelines are described by YAML configuration files, that can be displayed both as human-readable data or as a graph

The Kubeflow Pipelines SDK provides a set of Python packages to specify and run any machine learning (ML) workflow, including all of the components that make up the steps in the workflow and how the components interact with each other.

3.4. Components

Each component requires its own container image to package the program. Components can create outputs that the downstream components can use as inputs, each output must be a string and the container image must write each output to a separate local text file. For this project we built the following the docker images

- *ddos-classifier-base* describes the base image for all the other images. The image packages all the dependencies shared by the components (e.g. python, conda, tensorflow)
- *ddos-classifier-preprocess-train* packages the training dataset and the source code to preprocess it
- *ddos-classifier-preprocess-test* packages the test dataset and the source code to preprocess it
- *ddos-classifier-train* packages the source code to train the model
- *ddos-classifier-test* packages the source code to evaluate the model

3.5. Experiments

An experiment is a workspace to try different configurations of pipelines and organize runs into logical groups. Experiments can contain arbitrary runs, including recurring runs. A run is a single execution of a pipeline. Runs comprise an immutable log of all experiments, and are designed to be self-contained to allow reproducibility. Progress of a run can be tracked by looking at its details page on the Kubeflow Pipelines UI: the runtime graph, output artifacts, and logs for each step are available.

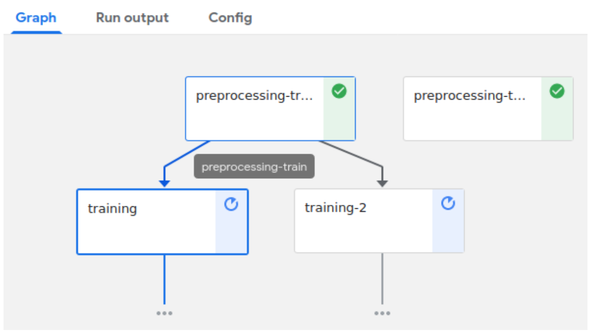


Figure 5. Depending on the availability of the underlying resources, Kubeflow (and Kubernetes) executes independent components concurrently. The pipeline runner service analyzes the pipeline configuration file to find an optimal scheduling.

Considering components scheduling, we have also tested the behaviour of Kubeflow by recreating possible scenarios on the cluster

- If enough memory is available on a single host, concurrent components are scheduled on the same host. If this is not the case, components are distributed over different hosts depending on the memory availability and the requirements of the component (and queued if these are not met)
- Failing of a master or slave node causes an interruption of any experiment, resumed as soon as the node is made available again

3.6. Results

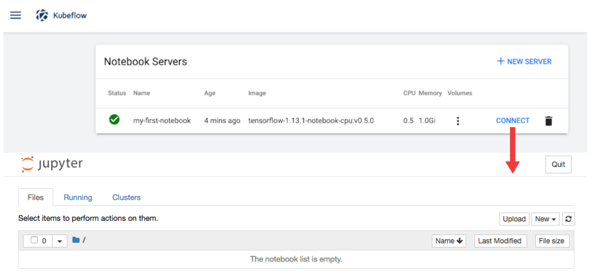
We consider now the usual task, involving preprocessing, hyper-parameter optimization, training, testing over the previously described DNN. Our model search will be executed over the following set of values, representing 8 possible configurations.

Parameter	Possible values
Dropout rate	0.1, 0.2
Learning rate	0.1, 0.3
Number of hidden units	[60, 30, 20], [60, 40, 30, 20]

TABLE 1. SET OF POSSIBLE VALUES FOR THE HYPER-PARAMETER OPTIMIZATION

For the same task we will showcase the times required by the following implementations. The code is almost the same except few adjustments required in the case of code containerization.

1) *Jupyter notebook* implementing all the phases, run on a notebook server (2CPU, 10GB)
2) *Kubeflow Pipeline*
    a) Concurrent, with two branches: branch 1 (with *training* and *testing* components) works on a [60, 30, 20] structure, branch 2 (with *training-2* and *testing-2*) works on a [60, 40, 30, 20] structure. Each branch executes the hyper-parameter tuning for dropout rate and learning rate
    b) Non-concurrent, with just one branch that executes hyper-parameter tuning on number of hidden units, learning rate and dropout rate



Status	Name	Age	Image	CPU	Memory	Volumes
✓	my-first-notebook	4 mins ago	tensorflow-1.13.1-notebook-cpu:v0.5.0	0.5	1.0Gi	⋮

Select items to perform actions on them.

The notebook list is empty.

Figure 6. Kubeflow allows to launch one or more instances of notebook servers (1a) and run Jupyter notebooks. In our case we execute a run of a notebook that includes all the phases previously described.

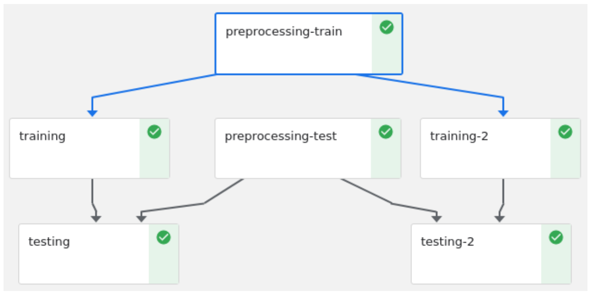


Figure 7. Kubeflow pipeline (2a) with concurrency extended to the training and testing phases

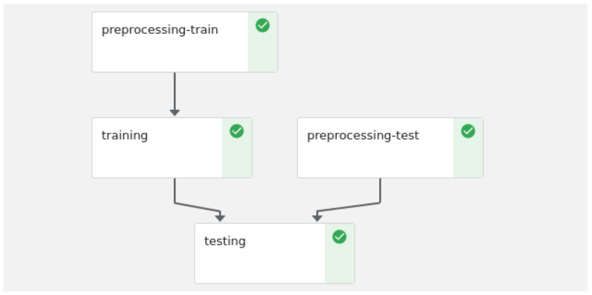


Figure 8. Kubeflow pipeline (2b) with concurrency only on the processing phase

Below we list some useful considerations on the resulting times listed on TABLE 2.

Phases	1a	2a	2b
Preprocessing			
- Training dataset	0:03:36	0:09:19	0:08:34
- Test dataset	0:07:08	0:11:26	0:09:10
Training	4:42:25		9:10:22
- [60, 30, 20]		3:45:04	
- [60, 40, 30, 20]		3:41:46	
Testing	0:23:06		0:42:57
- [60, 30, 20]		0:35:58	
- [60, 40, 30, 20]		1:06:37	
Overall	5:22:04	5:40:21	10:17:47

TABLE 2. TIMES COMPARISON BETWEEN DIFFERENT SOLUTIONS (EXPRESSED IN H:MM:SS)

- By comparing implementations 2a and 2b we can notice a significant reduction of times, justifying the concurrency of some tasks as a mean to decrease the overall time. While preprocessing and testing are splitted in two, the main time advantage is due to the splitting of the training task (since this is by far the longest among all). The concurrent execution of components is obviously limited by the amount of resources available, and in our tests we were able to have at most two concurrent components.
- The pipeline implementations are both affected by a small time overhead (in the order of few minutes) caused by the management of each component and related pod initialization executed by Kubeflow. While more components lead to a growth of this overhead, the resulting delay is minimal compared to the usual overall execution time.
- The pipeline implementations fire the following warning on the training phase

```
Your CPU supports instructions that this
  TensorFlow binary was not compiled to
  use: SSE4.1 SSE4.2

```

  The warning is not fired by 1a and fixing this issue may increase the training speed of solutions 2a and 2b.

3.7. Katib

Katib[6] is a Kubeflow component used for hyperparameter tuning and neural architecture search. Currently hyperparameter tuning and neural architecture search are respectively in beta and alpha version. At the moment this component seems to be independent from the Pipeline component, but as soon as Katib will exit the beta status it is recommended to further investigate its potential.

4. Conclusions

4.1. Dataset

The development of our model was a useful chance to dive into the CICDDoS2019 dataset and discover interesting details about how the data is organized. We have shown the highly imbalanced nature of the dataset, something that can have a great impact on its effectiveness. Our focus was on the csv archives, where the network traffic is described by a large set of pre-selected features. However a more fine-grained approach would certainly benefit from the raw data traces: extracting arbitrary features from the dataset boosts the complexity of the data preprocessing, but also allows to develop a wider range of solutions. Further work along this path is recommended to reveal the true value of this dataset.

6. https://www.kubeflow.org/docs/components/hyperparameter-tuning/

4.2. Kubeflow

This was also a chance to have a first taste of the Kubeflow toolkit. While the framework documentation gives a great introduction to the core ideas behind this environment, many concepts are roughly (or not at all) explained. However we have to keep in mind that v1.0 is the first major release, with many changes and improvements from previous releases, and with v1.1 scheduled fro June 2020 more changes are yet to come. Only few examples are available, often requiring a Google Cloud infrastructure to work and making the job harder for in-house hosting.

Using the sandbox environment hosted on DiVINE, we managed to translate a Jupyter notebook to a ML pipeline using the Kubeflow Pipeline SDK. In terms of resources utilization, this implementation is certainly an overkill and the overhead provided by Kubeflow is hardly justifiable for the job. While the magnitude of tasks that can really justify the use of this toolkit is certainly beyond this project, some of the benefits are clear:

- The use of Kubeflow components maximizes portability and reusability of code, especially if compared with environments such as notebook servers
- Dividing the workflow in distinct components highlights independent tasks, allowing to execute multiple independent jobs (e.g. preprocess, train, test) at the same time.
- If a pipeline run fails on a component, the results of previous components are safe and the next retry will start from failed component only

Also all the main benefit of using Kubernetes are inherited by Kubeflow, enabling orchestration, scaling and management over a cloud environment. A ML project requiring massive resources can certainly benefit from Kubeflow and further work along this path is recommended to reveal all real benefit of using such a tool.

References

[1] Sharafaldin, Iman Habibi Lashkari, Arash Hakak, Saqib Ghorbani, Ali. (2019). Developing Realistic Distributed Denial of Service (DDoS) Attack Dataset and Taxonomy. 1-8. 10.1109/CCST.2019.8888419.