Project 1: Introduction to Deep Learning

Surnames: Humblot, Anvarov, El Belghiti

First names: Arthur, Bekhzod, Ghita

Contents

Chapter 1

Lab1:

1.1 Introduction

The objective of this lab is to develop a machine learning model capable of performing network packet analysis on a labeled dataset. The goal is to classify each network flow as either benign traffic or malicious traffic originating from an attack. By leveraging supervised learning techniques, we aim to build an intelligent system that can detect abnormal behaviors, distinguish legitimate communication from intrusion attempts, and support automated network threat detection.

1.2 Task 1: Data Preprocessing

The first task of our project was preprocessing the dataset. This included removing infinite values, handling missing entries, and eliminating duplicated samples. Here is the code used,

```
# Set random seed for reproducibility
random_seed = 110
np.random.seed(random_seed)

# Open the dataset
data = pd.read_csv("dataset_lab_1.csv")
print("Number of rows in our dataset", data.size)

# Remove rows with missing values
data = data.replace([np.inf, -np.inf], np.nan) # whith the fit_transform python was co
data = data.dropna()
print("Number of rows after missing values", data.size)
```

```
# Remove duplicates
data = data.drop_duplicates()
print("Number of rows after duplicates", data.size)

# Encode labels
label_encoder = LabelEncoder()
data['Label'] = label_encoder.fit_transform(data['Label'])
```

Q:Howmanysamplesdidyouhavebefore and after removing missing and duplicates entries?

After preprocessing, the dataset size evolved as follows:
Number of rows in the original dataset: 535,619
After removing missing values: 535,160
After removing duplicates: 499,562

Q: Split the dataset to extract a training, validation and test sets (60Remember to set the seeds if you want reproducibility. Q:Focus on the training and validation partitions. Check for the presence of outliers and decide on the correct normalization.

Next, we attempted to visualize potential outliers using boxplots. We observed that most data points are highly concentrated, as shown by the very compact interquartile ranges. However, the remaining values still span a very large range, which is expected for data originating from network traffic analysis. A typical example is the distribution of destination ports: although the majority fall between 0 and 1000, all port numbers can be targeted, leading to a naturally wide spread in the data.

In our dataset, the features present very different numerical scales, ranging from small integer values to large continuous measurements. Such discrepancies in magnitude can significantly affect the learning process of most machine learning algorithms, and neural networks in particular. When features are not normalized, the model tends to assign disproportionate importance to variables with larger numerical ranges, which leads to unstable gradients, slower convergence, and poorer overall performance.

Q: How did you normalize the data? Why did you choose it?

To mitigate this issue, we apply the standard normalization procedure introduced in the course by using the `StandardScaler`. This method centers each feature around zero and rescales it to unit variance, ensuring that all variables contribute more evenly during training. The scaler is fitted on the training set and subsequently applied to the validation

and test sets to avoid data leakage. We also convert the data to PyTorch tensors just after. The corresponding preprocessing code is shown below:

```
# Split label in Y and features in X
X = data.drop(columns=['Label']).values
y = data['Label'].values

# Train/val/test split
X_train, X_temp, y_train, y_temp = train_test_split(
    X, y, test_size=0.4, random_state=random_seed
)
X_val, X_test, y_val, y_test = train_test_split(
    X_temp, y_temp, test_size=0.5, random_state=random_seed
)

# Standardize the features
scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)
X_val = scaler.transform(X_val)
X_test = scaler.transform(X_test)

# Convert data to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.long)
X_val_tensor = torch.tensor(X_val, dtype=torch.float32)
y_val_tensor = torch.tensor(y_val, dtype=torch.long)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.long)
```

1.3 Task 2: Shallow Neural Network

In this part of the project, we first focused on generalizing and structuring the codebase to ensure clarity and reusability. Whenever possible, we created helper functions to factorize repetitive operations and maintain a modular design. Although a few sections remain less factorized due to their specificity, the core of the implementation follows this principle to keep the workflow clean and consistent.

We then implemented three shallow neural network models with 32, 64, and 128 neurons, using the hyperparameters specified in the assignment. All three models converged

quickly and without any noticeable instability. This behavior is expected given the simplicity of the architecture and the reliability of the chosen training configuration—namely: a single linear layer, the AdamW optimizer with a small learning rate (0.0005), a batch size of 64, cross-entropy loss, and standard weight initialization. Such settings generally lead to smooth optimization dynamics and stable gradient flow, especially in shallow networks.

For each architecture, the model selection process is based on the accuracy ans the F1-score obtained on the test set. The model achieving the highest with this two score is considered the best-performing configuration. After training each model for up to 100 epochs (with early stopping when appropriate), the best-performing architecture was the one with 64 neurons, achieving a classification accuracy of 89.7 percent on the validation set (all models are really close to each other). This performance are quite poor because it's not that much and there's still many noise in the loose function. Let's see with ReLu fonctions.

When replacing the linear activation with a ReLU activation function, we observed similar convergence behavior across all three architectures. However, the performance improved significantly. ReLU introduces non-linearity into the model, allowing it to capture more complex decision boundaries compared to a purely linear transformation. As a result, the best model reached a substantially higher accuracy of 95.3 percent, demonstrating the clear benefit of using non-linear activation functions even in relatively shallow neural networks.

1.4 Task 3: The impact of Specific Features

No, it is incorrect to assume that all brute-force attacks occur only on port 80. For example, brute-force attacks using tools like Hydra can target port 22 (SSH) or any other high-numbered port hosting a service or web application. This assumption is therefore too simplistic and does not accurately reflect the diversity of potential attack vectors in network traffic.

After replacing all occurrences of port 80 with port 8080 in the test set, we observed nearly the same results as before. This indicates that the port number is not a determining factor in identifying these brute-force attacks. It is likely that other features, such as the number of packets or similar traffic characteristics, are the real discriminative factors. Therefore, we can safely remove the port feature from the dataset, at least for the brute-force attack data, without affecting model performance.

The Brute Force class is the most affected because attackers typically repeat the exact same scan across many ports to determine whether a particular service is available on a target machine. When we remove the Destination Port feature and drop duplicates, all these nearly identical scans collapse into a much smaller number of unique entries. As a

result, the number of samples drops from around 5000 before removing duplicates to only 285 afterward.

Consequently, the classes remain clearly unbalanced, as shown below.

```
Class distribution after preprocessing:
 Label
Benign          16889
DoS Hulk         3868
Brute Force      1427
PortScan          285
```

1.5 Task 4: The impact of the Loss Function

After removing the port information, our overall accuracy increased significantly. This confirms that the port number was not a truly relevant feature for our model. However, the PortScan class was drastically reduced during preprocessing, meaning that the model now has far fewer examples to learn from. As a result, its performance on this specific class likely degraded, but this drop does not appear in the overall accuracy because the class has become heavily under-represented. In practice, the model is no longer able to correctly classify PortScan samples (only 30 percent accuracy), even though the global accuracy seems high.

To estimate the class weights, we use only the training set to avoid any information leakage from the validation or test sets.

After recalculating the class weights and retraining the model using the weighted loss, we observe that recall improves for the minority classes, while overall precision decreases slightly. Although recall increases significantly, the noticeable drop in precision indicates that the chosen weighting strategy may not be appropriate. Despite integrating class weights into the loss function, the model still performs worse in terms of precision and F1-score compared to the unweighted version. This suggests that the weights generated by computeclassweight("balanced") are not well suited to this specific dataset. In theory, properly tuned weights should enhance the performance of the minority PortScan class, resulting in a higher F1-score for that class and only a moderate reduction in overall accuracy. The larger-than-expected decline in performance implies that the automatically computed weights may be too strong or poorly aligned with the true data distribution. So we tried an other version of this wheight version. This new version have the same accuracy as the unweighted version but a better recall. The global accuracy is still the same. We'll use the unweighted version for the rest of the Lab.

1.6 Task 5: Deep Neural Network

All models converge on this task, indicating that the training process is stable and that each architecture is able to optimize the loss function effectively.

We select the best-performing architecture by applying the same criterion as before: choosing the model that provides the best balance between accuracy and F1-score on the validation set, ensuring both overall correctness and good performance across all classes. Below is the report of the best model we choose:

By changing the batch size effectively the performance changes. The weighted loss improves recall for minority classes but can slightly reduce overall precision and F1-score, as the model focuses more on rare classes. Validation results reflect higher recall but slightly lower overall accuracy. Larger batch sizes reduce the number of weight updates per epoch, so training is faster per epoch but may require more memory. Smaller batch sizes increase computation per epoch and take longer due to more frequent updates. We find that the best batch size for our model is 64

The different optimizers show different convergence behaviors. AdamW converges faster and more smoothly, while SGD without momentum is slower and more oscillatory. Adding momentum to SGD improves stability and speed of convergence. We also observe that Adamw train faster than the other optimizes. We observe that AdamW is the best optimizer here.

Higher learning rates can speed up convergence but may overshoot the minimum, while lower rates converge more slowly. More epochs allow better learning but can lead to overfitting. The best model achieves high test accuracy and F1-score with a balanced learning rate and sufficient epochs. Below is the report of the best model we choose:

```
Classification Report per class:
              precision    recall  f1-score   support

           0     0.9691    0.9873    0.9781      3397
           1     0.9242    0.9343    0.9292       274
           2     0.9914    0.8926    0.9394       773
           3     0.5333    0.6400    0.5818        50

    accuracy                         0.9640      4494
   macro avg     0.8545    0.8636    0.8571      4494
```

```
weighted avg     0.9653    0.9640    0.9641      4494

The function took 0.0334 seconds to execute.
96.39519359145527
```

1.7 Task 6: Overfitting and Regularization

The purpose of this section appears to be to examine overfitting in our model and explore correction methods through normalization techniques. Interestingly, our first model does not overfit, which might seem surprising at first. However, this is theoretically reasonable since we only trained for 50 epochs, which is relatively few, and the model uses AdamW with a very low learning rate, both factors that work against overfitting. Nevertheless, we will investigate the impact of applying normalization practices on this model.

The first method involves using batch normalization, while the second involves noise injection. We observe that with batch normalization, overfitting is introduced in our model, causing its performance to decrease. In the case of noise injection, overfitting is not introduced, but the model's performance is still degraded. Therefore, we can conclude that normalization techniques can sometimes harm the model rather than improve it if they are not applied appropriately.