Inteligência Computacional - Redes Neurais

MLP aplicado no CIC-IDS2017

Autores:
Gustavo D. Ventino
Arthur Faria

Universidade Federal de Uberlândia
Bacharelado em Ciências da Computação - Inteligência Computacional

Fevereiro de 2025

SUMÁRIO

0.1 Resumo

Nos últimos anos, o aumento exponencial do número de dispositivos conectados à rede, impulsionado pela expansão da Internet das Coisas (IoT), tem transformado a forma como interagimos com a tecnologia. Dispositivos como câmeras de segurança, assistentes virtuais, carros conectados e até eletrodomésticos inteligentes são cada vez mais comuns em ambientes cotidianos. No entanto, essa maior conectividade também trouxe consigo um aumento significativo nos riscos de invasão de privacidade, comprometendo dados sensíveis. A vulnerabilidade desses dispositivos a ataques cibernéticos pode resultar não apenas na perda de informações privadas, mas também na falha de funcionalidades essenciais, como a interrupção de serviços em carros conectados ou falhas em dispositivos médicos.

Em resposta a esses riscos, uma primeira abordagem é utilizar um *Network Intrusion Detection System* (NIDS), a fim de analisar o tráfego da rede e extrair dados importantes para a categorização de ataques. Entretanto, este tipo de abordagem tem duas limitações: i) os tipos de ataque já devem ser conhecidos e característicos, ii) ela gera uma alta taxa de falsos positivos. Mas não é o fim do mundo, Uma alternativa é o uso de técnicas de *Machine Learning* (ML). Essas técnicas incluem o aprendizado supervisionado para classificar fluxos de rede em diferentes categorias e o aprendizado não supervisionado para detectar anomalias.

O aprendizado supervisionado depende da disponibilidade de um conjunto de dados, neste caso, esses dados são provenientes da captura de pacotes de rede contendo tráfego normal e ataques. Para nossa sorte, conjuntos de dados comuns para detecção de intrusão em redes, como KDD-Cup99, NSL-KDD e CIC-IDS2017, que possuem diferentes tipos de ataques contra computadores ou servidores, podem ser usados para detecção e classificação de ataques. Pretendemos fazer isso neste trabalho por meio de um *Multi Layer Perceptron* (MLP) com base no artigo "Efficient MLP-Based NIDS for CICIDS2017 Dataset" (Arnaud Rosay et al; CARLIER; LEROUX, 2020) e no repositório disponibilizado pelos autores (ROSAY, 2025)

Fizemos um repositório com o código utilizado para isso, (VENTINO; ARTHUR, 2025).

0.2 Escolha do Dataset

Neste momento inicial, vamos discutir o por quê da escolha do CIC-IDS2017 como *dataset* principal do trabalho. Vamos fazer isso passando pelo histórico dos datasets alternativos que poderíamos ter utilizado para este trabalho. Vamos começar com o KDD-Cup99, este é um conjunto de dados público para NIDS's, lançado em 1999, derivado do dataset DARPA-98, que contém dados brutos de *frames* de rede capturados com *TCPDUMP*. O conjunto de treinamento inclui 24 tipos de ataques e foi processado para gerar 41 *features*. Uma primeira crítica a esse conjunto foi a proposta do NSL-KDD, uma versão aprimorada para resolver algumas limitações do KDD-Cup99. O NSL-KDD tem sido amplamente utilizado em NIDS desde 2009. Embora os ataques atuais não estejam presentes em um conjunto de dados que depende de tráfego registrado há 20 anos, tanto o KDD-Cup99 quanto o NSL-KDD ainda são estudados.

Em 2015, um novo conjunto de dados chamado UNSW-NB15 foi proposto, este é gerado por simulação e representa 31 horas de tráfego. Ao contrário do NSL-KDD, ele inclui ataques modernos de baixo impacto, retirados do site CVE, agrupados em nove famílias diferentes. O UNSW-NB15 é composto por 49 *features*, algumas extraídas dos *headers* de pacotes e outras geradas especificamente. O KDD-Cup99 e seu derivado NSL-KDD sofrem com a falta de alguns protocolos importantes, como o HTTPS, que é o protocolo de referência para sistemas atuais, amplamente utilizado.

O CIC-IDS2017 foi lançado pelo *Canadian Institute for Cybersecurity* da Universidade de New Brunswick, com o objetivo de resolver as limitações dos conjuntos de dados anteriores. Os dados brutos estão disponíveis no formato *Package Capture* (PCAP), acompanhados por um conjunto de 84 *features* em arquivos *Comma Separated Values* (CSV). O tráfego de rede foi registrado ao longo de 5 dias, sendo o primeiro dia apenas com tráfego normal e os demais com 14 tipos de ataques. Devido às limitações do KDD-Cup99 e do NSL-KDD, a escolha acabou sendo restrita entre os conjuntos de dados UNSW-NB15 e CICIDS2017. A Figura 1 apresenta 7 parâmetros relevantes, incluindo o ano de criação de cada conjunto, o número de *features*, o número de registros e a distribuição entre tráfego normal e dados de ataque, informações cruciais para a seleção do *dataset*.

Table 1. Comparison of UNSW-NB15 and CICIDS2017.

Parameters	UNSW-NB15	CICIDS2017
Year of creation	2015	2017
Features	49 (+2 labels)	84 (+1 label)
Attack families	9	14
Duration	31 h	5 days
# of instances	2,540,044	2,830,743
# of normal instances	2,218,764	2,273,097
# of attack instances	321,283 (12.65%)	557,646 (19.70%)

Figura 1 – Tabela de comparação UNSW-NB15 x CICIDS2017
Retirada de (Arnaud Rosay et al; CARLIER; LEROUX, 2020)

0.3 Overview da Metodologia e Solução MLP

Após selecionarem um conjunto de dados, os autores do artigo propõem uma abordagem que consiste em treinar um MLP para quantificar os benefícios e as limitações potenciais do uso de aprendizado supervisionado em NIDS's. A Figura 2 fornece uma noção geral do *framework* proposto.

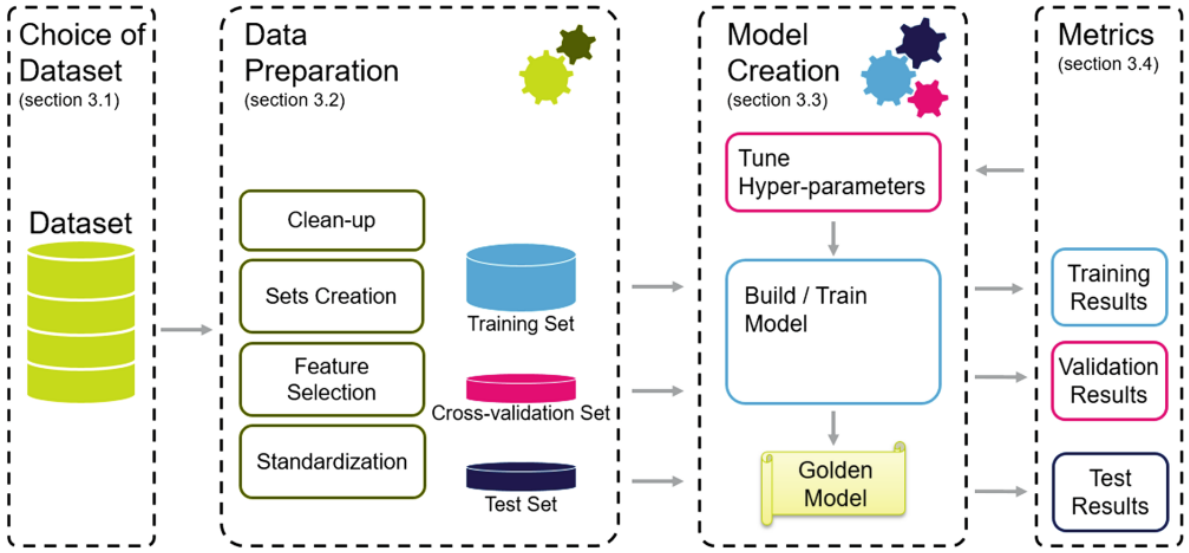

Figura 2 – Visão geral da abordagem proposta
Retirada de (Arnaud Rosay et al; CARLIER; LEROUX, 2020)

Note que, o fluxo de trabalho é dividido em quatro etapas principais, sendo elas:

1. Escolha do Dataset,
2. Preparação dos dados,
3. Criação do modelo,
4. Coleta de métricas.

Note que há um detalhamento de cada uma dessas etapas na figura, iremos passar por todas elas, a fim de alcançarmos bons resultados.

0.4 Preparação dos dados

0.4.1 LIMPEZA DO *DATASET*

A ideia proposta pelos autores do artigo nesta etapa é realizar uma limpeza geral nos dados, a fim de facilitar a manipulação destes. Para isso, uma série de operações foram realizadas a fim de detectar linhas vazias, características redundantes e valores não numéricos em campos numéricos. Isso permitiu descartar mais de 280.000 instâncias com todas as características vazias, além de algumas colunas repetidas. Embora a maioria das características seja numérica, alguns valores não são números (NaN), aparecendo em 6 tipos de tráfego: **BENIGN**, **FTP-PATATOR**, **DoS Hulk**, **Bot**, **PortScan** e **DDoS**. Como o número de instâncias com 'NaN' ou 'Infinity' é irrelevante em cada tipo de tráfego, essas instâncias foram simplesmente removidas. Assim, conseguimos realizar uma limpeza geral dos dados, removendo tudo aquilo que poderia atrapalhar nosso processo de aprendizado supervisionado, além de reduzir a quantidade de entradas e a dimensionalidade do *dataset*. A Figura 9 mostra o número de registros para cada tipo de tráfego e o número de 'NaN'/'Infinity'.

These match the functions shown in your image, with equation numbers (5) and (6).

Traffic	Original instances	NaN/Infinity	After clean-up
BENIGN	2,273,097	1,777	2,271,320
Bot	1,966	10	1,956
DDoS	128,027	2	128,025
DoS GoldenEye	10,293	0	10,293
DoS Hulk	231,073	949	230,124
DoS Slowhttptest	5,499	0	5,499
DoS Slowloris	5,796	0	5,796
FTP-PATATOR	7,938	3	7,935
Heartbleed	11	0	11
Infiltration	36	0	36
PortScan	158,930	126	158,804
SSH-PATATOR	2,897	0	2,897
WebAttack BruteForce	1,507	0	1,507
WebAttack SQL Injection	21	0	21
WebAttack XSS	652	0	652

Figura 3 – Instâncias de tráfego do CICIDS2017
Retirada de (Arnaud Rosay et al; CARLIER; LEROUX, 2020)

No nosso trabalho realizamos os mesmo passos, mas de maneira diferente. Para isso, utilizamos artifícios da biblioteca PANDAS do python, a qual permite a manipulação de dados com muita facilidade. Esta é uma de nossas preocupações neste trabalho, manter as implementações simples e coerentes. Isso se dá pois, embora os autores do artigo tenham

disponibilizado o código implementado por eles em (ROSAY, 2025), este código ao nosso ver não atende aos padrões atuais e boas práticas para códigos voltados para *datascience*. Então buscamos realizar o mesmo processo de maneira muito simples, entretanto isto faz com que surjam sutis diferenças de implementação. Por exemplo, preferimos realizar a etapa de limpeza dos dados após a divisão destes em conjuntos de treino, *cross-validation* e teste, como é possível verificar nas Figuras 4 e 5.

Figura 4 – Data Cleanup Pt.1
Retirada de (VENTINO; ARTHUR, 2025)

```
columns = [
    'bwd psh flags',
    'bwd urg flags',
    'fwd avg bytes/bulk',
    'fwd avg packets/bulk',
    'fwd avg bulk rate',
    'bwd avg bytes/bulk',
    'bwd avg packets/bulk',
    'bwd avg bulk rate',
    'timestamp',
]

print(f"Size before dropping: {len(df_train.columns)}")
df_train = df_train.drop(columns=columns)
print(f"Size after dropping: {len(df_train.columns)}")
```

```
Size before dropping: 85
Size after dropping: 76
```

Figura 5 – Data Cleanup Pt.2
Retirada de (VENTINO; ARTHUR, 2025)

0.4.2 DIVISÃO DO *DATASET*

Como mostrado na Figura 1, o conjunto de dados é muito desequilibrado em dois níveis. Primeiro, o tráfego normal (**BENIGN**) representa 80%, e segundo, alguns ataques (**Heartbleed**, **Infiltration**, **WebAttack**, e **SQL injection**) são representados por um número muito baixo de instâncias. O que é um desafio para o aprendizado supervisionado, que funciona facilmente sobre conjuntos bem distribuídos, balanceados. A proposta para criar o conjunto de dados para treinamento e teste do MLP consiste em escolher aleatoriamente 50% de cada ataque para o conjunto de treinamento, 25% para o conjunto de *cross-validation* e 25% para o conjunto de teste, garantindo que cada instância seja usada apenas uma vez. Em seguida, cada conjunto é preenchido com instâncias de tráfego normal selecionadas aleatoriamente. Isso resulta em um conjunto de dados balanceado em termos de tráfego normal versus ataques, mas ainda desequilibrado em termos de tipos de ataque como pode ser visto na Figura 6

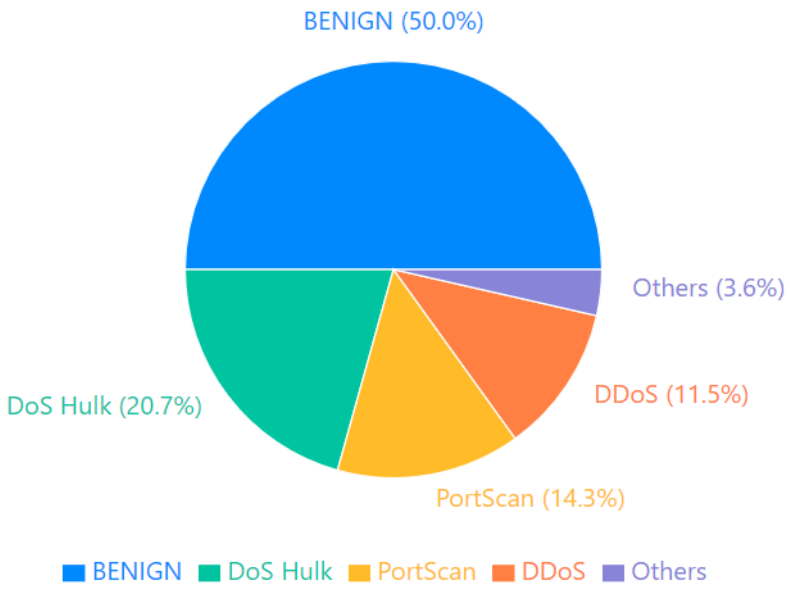

Figura 6 – Distribuição de entradas balanceadas
Elaborado pelo autor

A composição exata dos conjuntos de treinamento, cross-validation e teste é fornecida na Figura 7.

Traffic	Training set	Cross-validation set	Test set
BENIGN	278,274	139,135	139,135
Bot	978	489	489
DDoS	64,012	32,006	32,006
DoS GoldenEye	5,146	2,573	2,573
DoS Hulk	115,062	57,531	57,531
DoS Slowhttptest	2,749	1,374	1,374
DoS Slowloris	2,898	1,449	1,449
FTP-PATATOR	3,967	1,983	1,983
Heartbleed	5	2	2
Infiltration	18	9	9
PortScan	79,402	39,701	39,701
SSH-PATATOR	2,948	1,474	1,474
WebAttack BruteForce	753	376	376
WebAttack SQL Injection	10	5	5
WebAttack XSS	326	163	163
Total	**556,548**	**278,270**	**278,270**

Figura 7 – Divisão do *dataset*
Retirada de (Arnaud Rosay et al; CARLIER; LEROUX, 2020)

Seguimos a mesma divisão dos dados que os autores do artigo, inclusive fizemos uso dos arquivos *Parquet* disponibilizados por eles, os quais continham os dados divididos e não limpos, por isso que tivemos que limpar os dados divididos.

0.4.3 SELEÇÃO DE *FEATURES*

Esta selção de seleção de*features* é um dos principais conceitos de *Machine Learning* que influencia a performance do modelo. *Features* irrelevantes podem trazer um impacto no tempo de treinamento sem uma melhora suficiente na acurácia do modelo. A análise RFE realizada pelos autores do artigo revelou que oito *features* não são relevantes. Consequentemente, features 'Bwd PSH Flags', 'Bwd URG Flags', 'Fwd Avg Bytes/Bulk', 'Fwd Avg Packets/Bulk', 'Fwd Avg Bulk Rate', 'Bwd Avg Bytes/Bulk', 'Bwd Avg Packets/Bulk', 'Bwd Avg Bulk Rate', foram removidas. Uma vez que, não é desejado que o modelo aprenda quando o ataque ocorre, isto é, o *timestamp* não é considerado significativo. Além disso, cada instância é caracterizada pelos IP e Porta destino, o protocolo utilizado e um identificador de fluxo, que contém a mesma informação. Como este 'Flow ID' é redundante devido à seleção de outras *features*, este também foi removido pelos autores do artigo.

Iremos trazer mais informações sobre os conjuntos utilizados para experimentação e a quantidade de features correspondentes na seção dedicada a experimentação. O artigo prevê dois conjuntos, *Variant-1* e *Variant-2*, nós criamos novas variantes utilizando diferentes parâmetros no modelo, a fim de comparar a eficácia de métodos distintos.

0.4.4 NORMALIZAÇÃO

O pré-processamento de dados é essencial para preparar o *dataset* para um treinamento eficiente. Particularmente, em nosso caso, a rede neural pode aprender melhor quando todas as características estão escalonadas dentro do mesmo intervalo. Isto é útil quando os registros que servem de entrada para o modelo estão em escalas muito diferentes, como no nosso caso. Existem diversas técnicas de normalização, na implementação do artigo foi utilizada a técnica de normalização Z-score, também chamada de padronização. Isso se dá pois, esta técnica apresenta melhores resultados do que outras técnicas quando aplicadas no *dataset* em questão.

$$x_i^{(\mathcal{F}_j)} = \frac{x_i^{(\mathcal{F}_j)} - \mu^{(\mathcal{F}_j)}}{\sigma^{(\mathcal{F}_j)}} \tag{1}$$

Para cada *feature* F_j, realizamos uma transformação de normalização em cada valor x_i usando a Equação 1. Esta transformação, ajusta os dados para que tenham média zero e desvio padrão igual a um.

A equação de padronização funciona da seguinte maneira:

- Primeiro, subtraímos de cada valor x_i a média da característica ($\mu^{(F_j)}$)
- Depois, dividimos o resultado pelo desvio padrão da mesma característica ($\sigma^{(F_j)}$)

Onde:

- $x_i^{(F_j)}$ representa o valor da *feature*
- $\mu^{(F_j)}$ é a média de todos os valores da *feature* F_j
- $\sigma^{(F_j)}$ é o desvio padrão da *feature* F_j

0.4.5 CRIAÇÃO DO MODELO

MLP's são redes neurais classificadoras, conexas e *feed-forward*. A Figura 8 mostra o design arquitetural do modelo descrito pelos autores do artigo, com uma modificação na camada de entrada que será discutida mais tarde.

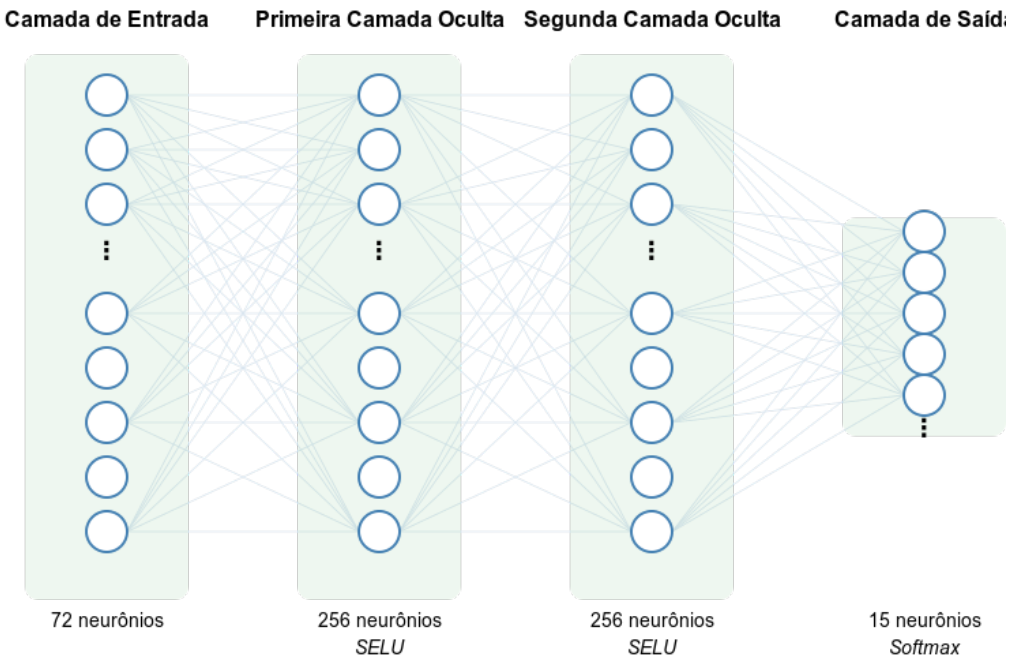

Figura 8 – Arquitetura utilizada
Elaborado pelo autor

As entradas correspondem aos valores normalizados das *features* selecionadas do *dataset* original. As duas camadas ocultas contém 256 neurônios cada. A camada de saída possui 15 neurônios, um para tipo de ataque e para o tráfego benigno. O *Dropout* é usado como uma técnica de regularização entre as camadas ocultas, a fim de prevenir *over-adjustment* nos dados do treinamento. A saída $h^{(3)}$ é calculada por meio do encadeamento das saídas de diferentes camadas, de acordo com as Eq. 2, 3 e 4 nas quais x, $W^{(i)}$ e $b^{(i)}$ é respectivamente, o vetor contendo as *features* selecionadas, a matriz de pesos e o vetor de *biases* para camada i.

$$h^{(1)} = g_1\left(W^{(1)T}x + b^{(1)}\right) \tag{2}$$

$$h^{(2)} = g_1\left(W^{(2)T}h^{(1)} + b^{(2)}\right) \tag{3}$$

$$h^{(3)} = g_2\left(W^{(3)T}h^{(2)} + b^{(3)}\right)' \tag{4}$$

As funções de ativação g_1 e g_2 introduzem não linearidade na rede neural. Como a MLP não fornece uma normalização intrínseca de suas saídas, a unidade linear exponencial escalonada, dada na Eq. 5, é utilizada como função de ativação para as camadas ocultas, com os valores com $\lambda = 1.0507$ e $\alpha = 1.6733$ para *self normalization*.

$$g_1(z) = \lambda \cdot \begin{cases} \alpha \cdot (e^z - 1), & \text{para } z < 0 \\ z, & \text{para } z \geq 0 \end{cases} \tag{5}$$

A camada de saída utiliza a função de ativação softmax g_2, conforme definido na Eq. 6, de modo que cada saída possa ser interpretada como a probabilidade de prever uma determinada classe. O rótulo previsto $\hat{y}$ é dado por:$\hat{y} = \arg\max h^{(3)}$

$$g_2(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{N} e^{z_k}}, \quad \text{para } j \in [1; 15] \tag{6}$$

0.4.6 TREINAMENTO

Para a implementação e treinamento do modelo, os autores do artigo utilizaram Python e TensorFlow, nós usamos o Python com o Keras. O conjunto de treinamento é dividido em pequenos *batches* de 32 instâncias. A MLP aprende a classificação ajustando os pesos w entre os nós da rede neural para reduzir a *cross-entropy loss function* $L(w)$, conforme definida na Eq. 7:

$$\mathcal{L}(w) = -\left[y \cdot \log(\hat{y}) + (1-y) \cdot \log(1-\hat{y})\right] \tag{7}$$

onde y é o rótulo real e $\hat{y}$ é a classe prevista. A função $L(w)$ é otimizada com o algoritmo Adam. Dois modelos diferentes, correspondentes à duas variantes citadas anteriormente foram treinadas.

0.5 Experimentos

A arquitetura dos autores do artigo foi seguida com algumas mudanças. Mantivemos iguais as camadas de saídas e camadas ocultas, tanto em número de neurônios quanto em funções de ativação. Porém distoamos em número de *features* por motivos que serão explicados futuramente. Nossa arquitetura está na Eq. 8

$$SELU(z) = \lambda \begin{cases} \alpha(e^z - 1) & \text{for } z < 0 \\ z & \text{for } z \geq 0 \end{cases} \tag{8}$$

$$SOFTMAX(z) = \frac{e^{z_j}}{\sum_{k=1}^{N} e^{z_k}} \text{ for } j \in [1:15] \tag{9}$$

Softmax: É usado para transformar as saídas de uma rede neural em probabilidades, tornando-o ideal para problemas de classificação multiclasse.

Os autores do trabalho original usaram duas arquiteturas, que se distinguem na camada de entrada. A primeira com 73 neurônios comporta *features* como endereços IP e *src port*, na segunda arquitetura essas *features* são removidas.

De acordo com o método de *feature selection Recursive feature elimination* (RFE) *Source IP* é a *feature* mais importante do *dataset*. Porém usá-la como paramêtro de treinamento não é realista. Afinal, o modelo aprenderia o endereço IP dos atacantes durante a simulação e passaria a classificar ataques por autoria do pacote em vez de suas características, isso se dá pois os dados desses ataques são simulados durante a criação do *dataset*.

Mas diferente dos autores, não retiramos a classe *src port* porque a porta geralmente está associada a um serviço, consequentemente é uma característica do pacote e não do autor. Acabamos assim com 72 *features*, por isso, temos 72 neurônios nas camadas de entrada.

Na tabela 1 estão alguns testes realizados. Procuramos adicionar técnicas como *Learning Rates* (LR) variáveis, *Cross Validation* (CV), *Batch Normalization* (BN) e condições de parada. Testamos diferentes *dropouts* (DO), e número de *Epochs*. A tabela não é muito extensa porque nos casos em que parâmetros geravam *Accuracy (Acc)* menor que 0.97 após 30 *epochs* o processo era interrompido e o modelo não era salvo.

Tabela 1 – Performance de alguns modelos salvos

Modelo	Acc	Loss	Epochs	CV	LR	BN	DO
0	0.9874	0.0456	45	No	0.001	No	0.5
102	0.9924	0.0312	80	25%	0.001	No	0.5
479746	0.9943	0.0197	50*	25%	0.01→0.0001	Yes	0.5
	0.9847	0.0609	40*	25%	0.01→0.0001	Yes	0.8

*Limite: 100 Epochs

0.5.1 CROSS ENTROPY

A *Cross Entropy* refere-se à diferença entre duas distribuições de probabilidade sobre o mesmo conjunto de eventos/dados. A *loss function* utilizada foi a *Cross Entropy* Eq. 10.

$$H(y, \hat{y}) = -\sum_{i=1}^{C} y_i \log(\hat{y}i) \tag{10}$$

Onde:

C é o número de classes (15 no nosso caso)

y_i é a probabilidade real da classe i

$\hat{y}i$ é a probabilidade prevista pelo modelo para a classe i

0.5.2 DROPOUT

O *Dropout* é uma técnica de regularização que ajuda a prevenir o *overfitting* em redes neurais. Durante o treinamento, neurônios são temporariamente "desligados"(junto com suas conexões) com uma probabilidade específica (taxa de *dropout*).

Funcionamento: Em cada iteração de treinamento, uma porcentagem de neurônios é aleatoriamente desativada. Isso força a rede a não depender excessivamente de neurônios específicos.

0.5.3 ADAM OPTIMIZATION

"*computationally efficient, has little memory requirement, invariant to diagonal rescaling of gradients, and is well suited for problems that are large in terms of data/parameters*".(KINGMA; BA, 2014)

O *Adaptive Moment Estimation* é um algoritmo de otimização que combina as vantagens de dois outros métodos: *RMSProp* e *Momentum*. Explicaremos o passo a passo simplificado:

- A rede faz uma previsão.
- Compara a previsão com os valores reais utilizando a entropia cruzada (cross entropy).

- Calcula os gradientes através da *backpropagation*.

Esses gradientes são recebidos pelo *Adam* que cria dois momentos, um m com a média móvel do gradiente, e um v com a média móvel do quadrado do gradiente. Ele então atualiza os pesos com os dois novos momentos. O cálculo da atualização é dado por:

$$atualização = \frac{LR * m}{\sqrt{v} + \epsilon} \tag{11}$$

E então o peso θ é calculado usando $\theta_{novo} = \theta_{antigo} + atualização$

0.5.4 BATCH NORMALIZATION

A *Batch Normalization* é uma técnica que normaliza os dados que servirão de entrada para a próxima camada, fazendo isso por meio de *mini-batches* durante o treinamento. No nosso caso temos batch size igual a 32. Suponha que estamos na primeira camada oculta de 256 neurônios. Cada neurônio faz seu processamento mas antes de aplicar a função de ativação o *batch normalization* é aplicado.

Ou seja, para cada um dos 256 neuronios teremos 1 *batch* com 32 valores cada. Calculamos a média e variância dentro desses batches e normalizamos suas saídas antes de passar estes valores para a próxima camada.

0.5.5 REDUCE LR ON PLATEAU

ReduceLROnPlateau é uma estratégia que reduz a taxa de aprendizado quando uma métrica monitorada para de melhorar.

Esta estratégia monitora uma métrica (geralmente *loss* de validação). Se a métrica não melhorar por um número específico de épocas (paciência), a taxa de aprendizado é reduzida por um fator. Continua reduzindo a taxa de aprendizado até atingir um valor mínimo.

0.5.6 EARLY STOPPING

Early Stopping é uma técnica de regularização que interrompe o treinamento de uma rede neural quando o desempenho no conjunto de validação para de melhorar. No nosso código a métrica verificada foi *validation loss*

0.5.7 CÓDIGO

```python
y_onehot = to_categorical(y_numeric)
y_crossval_onehot = to_categorical(y_crossval_numeric)

model = Sequential([
    Dense(256, input_dim=X_scaled.shape[1], activation='selu'),
    BatchNormalization(),
    Dropout(0.5),

    Dense(256, activation='selu'),
    BatchNormalization(),
    Dropout(0.5),

    Dense(len(label_encoder.classes_), activation='softmax')
])

optimizer = Adam(
    learning_rate=0.01,
    beta_1=0.9,
    beta_2=0.999
)
model.compile(
    optimizer=optimizer,
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

model.summary()

early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=10,
    restore_best_weights=True
)

reduce_lr = ReduceLROnPlateau(
    monitor='val_loss',
    factor=0.2,
    patience=5,
    min_lr=0.0001
)

history = model.fit(
    X_scaled.values, y_onehot,
    epochs=100,
    batch_size=32,
    verbose=1,
    validation_data=(X_crossval_scaled.values, y_crossval_onehot),
    callbacks=[early_stopping, reduce_lr]
)

y_pred = model.predict(X_scaled.values)
accuracy = tf.keras.metrics.categorical_accuracy(y_onehot, y_pred)
print(f"\nAccuracy: {tf.reduce_mean(accuracy).numpy():.4f}")
```

Figura 9 – Definição do modelo
Elaborado pelo autor

0.6 Resultados

0.6.1 RECALL (SENSIBILIDADE):

O recall mede a proporção de positivos reais que foram corretamente identificados:

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{138434}{138434 + 701} = 0,9950 = 99,50\% \tag{12}$$

0.6.2 PRECISÃO:

A precisão mede a proporção de identificações positivas que estavam corretas:

$$\text{Precisão} = \frac{TP}{TP + FP} = \frac{138434}{138434 + 796} = 0,9943 = 99,43\% \tag{13}$$

0.6.3 ACURÁCIA

A acurácia mede a proporção de predições corretas (tanto positivas quanto negativas):

$$\text{Acurácia} = \frac{TP + TN}{TP + TN + FP + FN} = \frac{138434 + 138339}{138434 + 138339 + 796 + 701} = 0,9946 = 99,46\% \tag{14}$$

0.6.4 F1-SCORE

O F1-Score é a média harmônica entre precisão e recall:

$$\text{F1-Score} = 2 \times \frac{\text{Precisão} \times \text{Recall}}{\text{Precisão} + \text{Recall}} = 2 \times \frac{0,9943 \times 0,9950}{0,9943 + 0,9950} = 0,9946 = 99,46\% \tag{15}$$

0.6.5 TABELAS

Tabela 2 – Métricas de desempenho do modelo

Métrica	**Valor**
TP	138434
FP	796
FN	701
TN	138339
Taxa TN (%)	99,43
Taxa FP (%)	0,57
Recall (%)	99,50
Precisão (%)	99,43
Acurácia (%)	99,46
F1-score (%)	99,46

Tabela 3 – Análise de classificação por tipo de tráfego

Classe	**BENIGN**	**Correto**	**Outro ataque**	**Total**
BENIGN	-	138339	796	139135
Bot	30	459	0	489
DDoS	26	31976	4	32006
DoS GoldenEye	47	2524	2	2573
DoS Hulk	37	57494	0	57531
DoS Slowhttptest	16	1342	16	1374
DoS slowloris	10	1428	11	1449
FTP-Patator	5	1975	3	1983
Heartbleed	0	2	0	2
Infiltration	7	2	0	9
PortScan	18	39664	19	39701
SSH-Patator	17	1454	3	1474
Web Attack Brute Force	330	22	24	376
Web Attack Sql Injection	4	0	1	5
Web Attack XSS	154	0	9	163

REFERÊNCIAS

Arnaud Rosay et al; CARLIER, F.; LEROUX, P. *MLP4NIDS: An Efficient MLP-Based Network Intrusion Detection for CICIDS2017 Dataset*. Cham: Springer International Publishing, 2020. 240–254 p. (Lecture Notes in Computer Science). Citado 5 vezes nas páginas 2, 4, 5, 6 e 10.

KINGMA, D. P.; BA, J. Adam: A method for stochastic optimization. dez. 2014. Citado na página 17.

ROSAY, A. *MLP4NIDS: Multi-Layer Perceptron for Network Intrusion Detection Systems*. 2025. <https://github.com/ArnaudRosay/mlp4nids>. Accessed: 2025-02-20. Citado 2 vezes nas páginas 2 e 7.

VENTINO, G.; ARTHUR. *MLP_CIC-IDS2017*. 2025. Acesso em: 20 fev. 2025. Disponível em: <https://github.com/gventino/MLP_CIC-IDS2017>. Citado 3 vezes nas páginas 2, 7 e 8.