Projeto 3:
Previsão de tráfego de rede com Redes Neurais Recorrentes

Tiago de Paula Alves (187679)
t187679@dac.unicamp.br

18 de junho de 2025

1 Introdução

Este projeto tem como objetivo modelar o tráfego em uma rede de computadores utilizando redes neurais recorrentes (LSTM). Para isso, utilizamos um arquivo de captura de pacotes (PCAP) da base pública MAWI/CAIDA, construímos uma série temporal de bytes por segundo e, por fim, treinamos um modelo LSTM para prever o próximo valor da série. A metodologia adotada e os resultados obtidos são detalhados nas seções subsequentes.

2 Análise Exploratória de Dados (EDA)

A primeira etapa do projeto consistiu em realizar uma análise exploratória sobre os dados de tráfego para compreender suas características fundamentais. O *dataset* original, após ser processado, resultou em uma série temporal de volume de tráfego agregado por segundo. O primeiro *dataset* escolhido para análise foi o `200701251400.dump.gz`, por conter um volume maior de dados.

Tabela 1: Estatísticas descritivas para a série temporal do `200701251400.dump.gz`.

Estatística	Valor
Quantidade	902
Média	1247.7 Kibit/s
Desvio Padrão	121.4 Kibit/s
Mínimo	143.7 Kibit/s
25° Percentil	1158.8 Kibit/s
50° Percentil	1238.7 Kibit/s
75° Percentil	1346.8 Kibit/s
Máximo	1502.9 Kibit/s

As estatísticas descritivas na Tabela 1 fornecem um resumo de alto nível da intensidade do tráfego de rede durante o período de captura de 15 minutos. O tráfego apresenta uma média de aproximadamente 1,27 MiB/s. No entanto, o desvio padrão é relativamente alto, 124,5 KiB/s, o que representa cerca de 10% da média. No entanto, podemos ver que essa instabilidade não aparece na série temporal da Figura 2, o que indica que as bordas da série causam esse alto valor de desvio padrão.

O gráfico do tráfego agregado ao longo do tempo na Figura 2 oferece uma visão intuitiva do comportamento dos dados. A série parece ter uma linha de base relativamente estável em torno de 1,2-1,4 MiB/s. Não há tendências óbvias e de longo prazo para cima ou para baixo nessa janela de 15 minutos, mas há "regimes" claros em que o tráfego é maior e mais volátil, seguido por períodos em que é menor e mais estável. As quedas repentinas no início e no final são provavelmente artefatos do início e da interrupção da captura.

A análise de decomposição na Figura 3 fornece os insights mais profundos. A componente de tendência revela as alterações subjacentes e lentas no volume de tráfego. Podemos ver uma diminuição gradual na carga geral de tráfego desde o início até por volta da marca de 05:08, após o que ela começa a se recuperar ligeiramente. Isso mostra que a série não é completamente estacionária.

O componente sazonal mostra claramente um padrão forte e repetitivo com um período de 60 segundos. Isso pode ser bem útil, pois indica um comportamento cíclico no uso da rede, talvez causado por processos automatizados em segundo plano, ferramentas de monitoramento ou comportamentos agregados de usuários que se repetem minuto a minuto. Por fim, o componente residual mostra o "ruído" ou as flutuações aleatórias deixadas após a remoção da tendência e da sazonalidade.

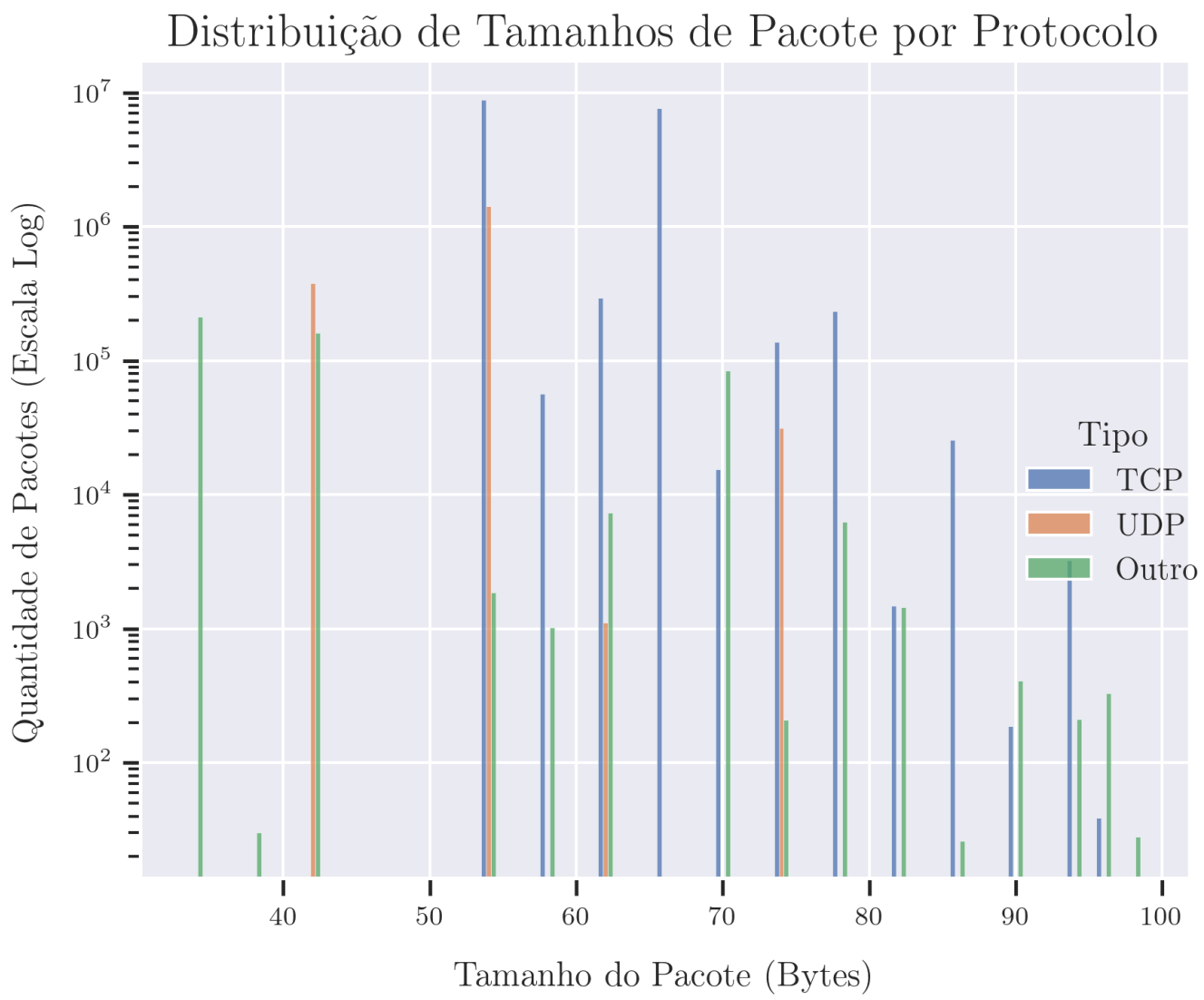

Figura 1: Distribuição de tamanhos de pacote para os protocolos TCP e UDP em `200701251400.dump.gz`. A frequência (eixo Y) está em escala logarítmica para melhor visualização.

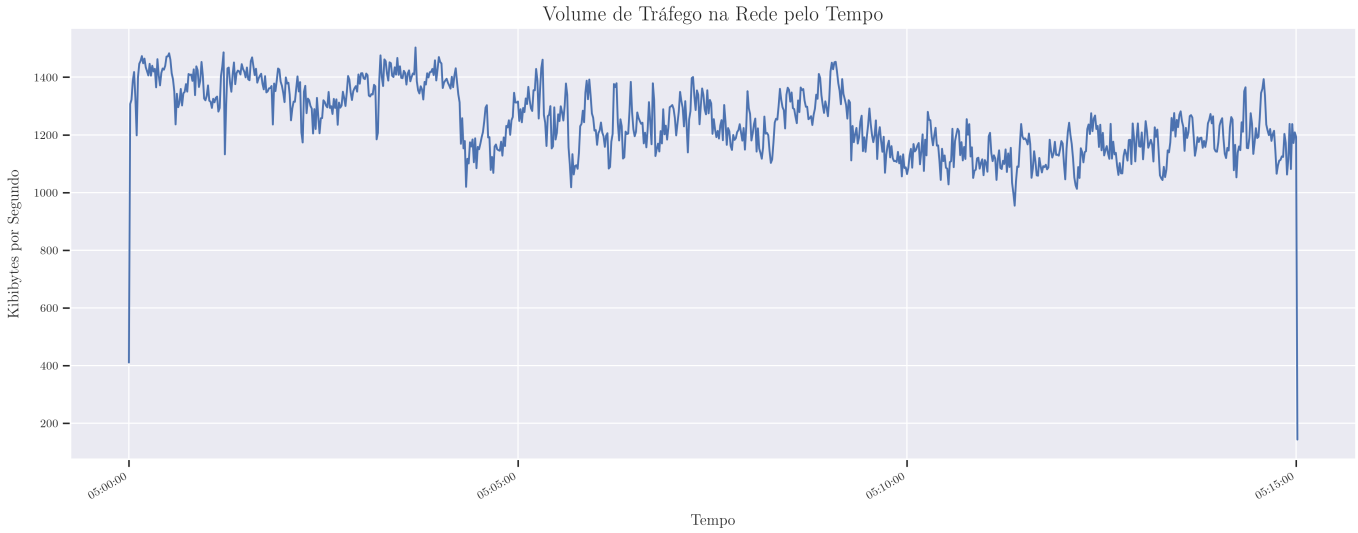

Figura 2: Série temporal do volume de tráfego agregado por segundo de `200701251400.dump.gz`.

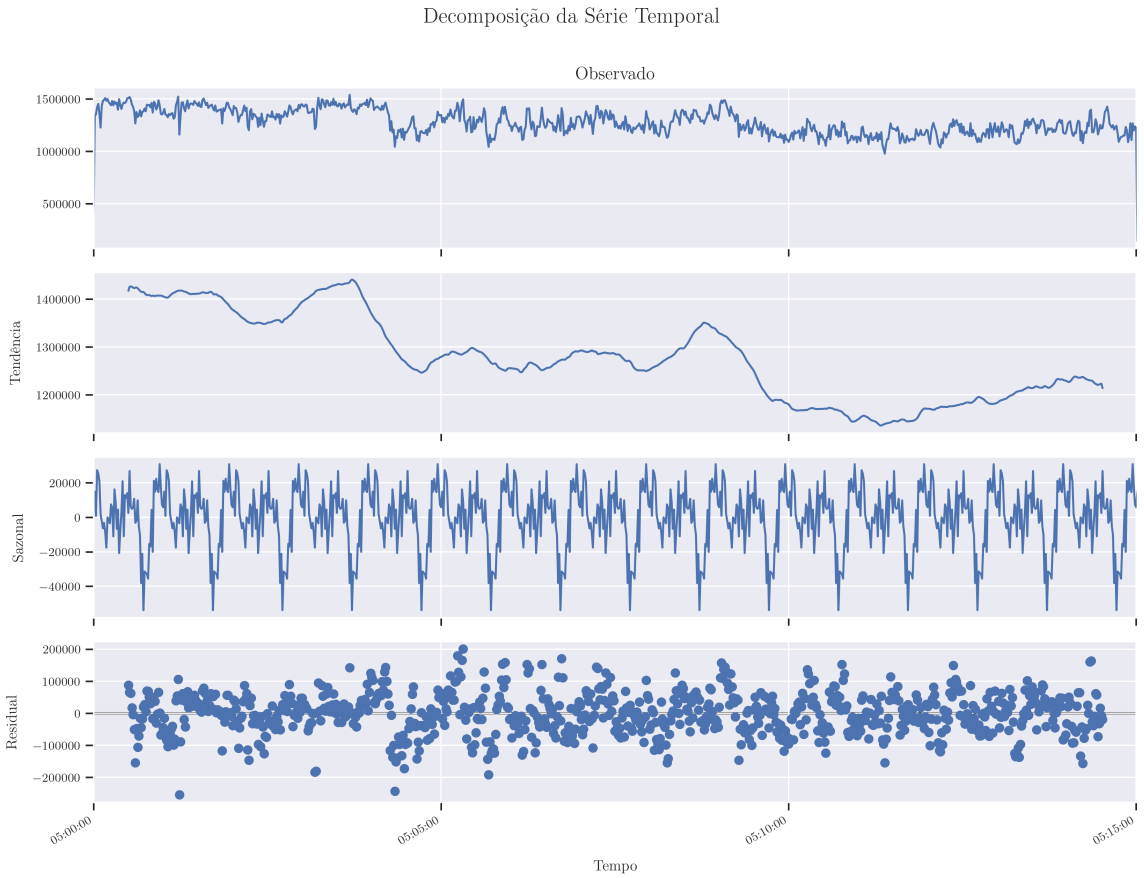

Figura 3: Decomposição da série temporal em tendência, sazonalidade e resíduos, com um período sazonal de 60 segundos para o *dataset* `200701251400.dump.gz`.

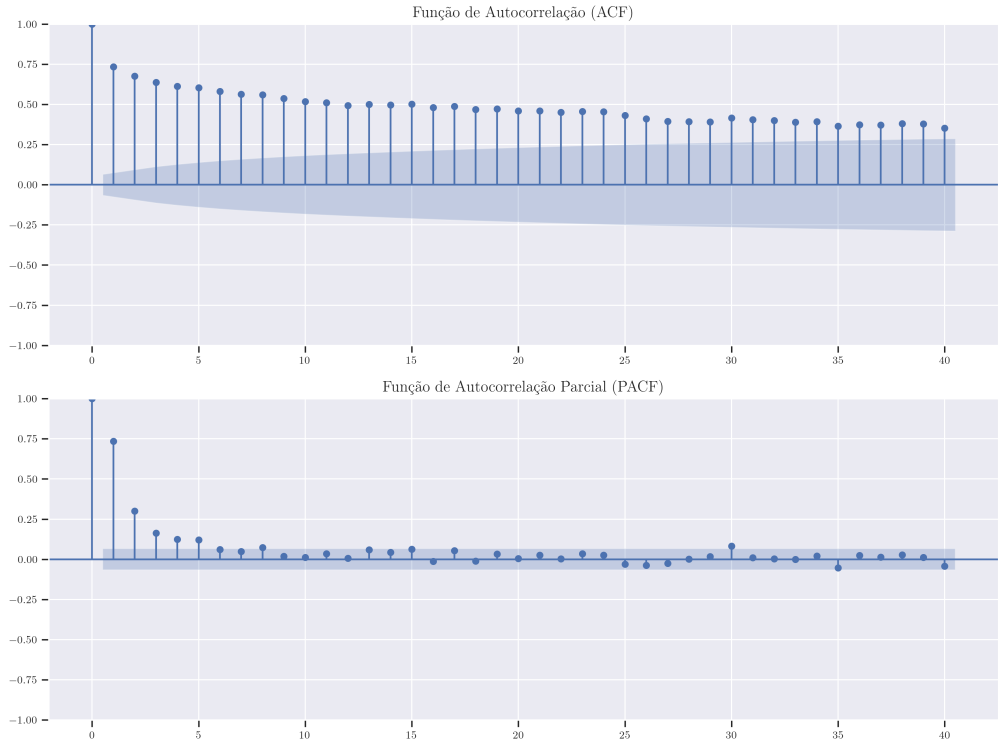

Figura 4: Funções de Autocorrelação (ACF) e Autocorrelação Parcial (PACF) para a série temporal de `200701251400.dump.gz`.

Por fim, os gráficos ACF e PACF na Figura 4 fornecem uma confirmação estatística dos padrões observados anteriormente. O gráfico da ACF mostra um decaimento muito lento. A correlação com valores passados permanece alta mesmo depois de alguns pontos, o que é um sinal clássico de uma não estacionariedade nos dados. Ela também confirma que o valor do tráfego em um segundo é altamente dependente do valor do segundo anterior.

O gráfico PACF é crucial para a escolha da janela look_back. Ele mostra um pico muito grande na defasagem 1, indicando uma forte correlação direta com o valor imediatamente anterior. Em seguida, a correlação cai drasticamente, tornando-se estatisticamente insignificante após cerca de 10 a 12 defasagens. Isso fornece uma forte justificativa estatística para experimentar uma janela look_back de 10, pois ela representa a memória de curto prazo mais significativa da série.

2.1 Aumentando a Quantidade de Pontos

Apesar do *dataset* `200701251400.dump.gz` ser composto de mais de 19 milhões de pacotes, a distribuição se dá em apenas 15 minutos de coleta. Para um análise mais aprofundada, foi utilizado também um *dataset* com 1515492 pacotes coletados em uma sessão de 9 horas, o `200701011800.dump.gz`.

Tabela 2: Estatísticas descritivas para a série temporal do `200701011800.dump.gz`.

Estatística	Valor
Quantidade	36815
Média	3653.0 bit/s
Desvio Padrão	5161.4 bit/s
Mínimo	428.0 bit/s
25° Percentil	1722.0 bit/s
50° Percentil	2304.0 bit/s
75° Percentil	3582.0 bit/s
Máximo	136 182.0 bit/s

A análise deste novo conjunto de dados revela características de tráfego em uma escala de tempo muito mais ampla, que são fundamentais para um modelo de previsão robusto.

A Tabela 2 mostra que média de tráfego neste dataset mais longo é significativamente menor, em torno de 120.4 KiB/s, comparada com os 1.27 MiB/s do dataset anterior. Mais importante, o desvio padrão é quase o dobro da média. Isso indica que, ao longo de um dia, o tráfego não é apenas volátil, mas também apresenta regimes de operação muito distintos (períodos de alta e baixa atividade), em vez da aparente estabilidade do tráfego no dataset de 15 minutos.

Apesar de conter muito ruído, o gráfico da Figura 6 é bem revelador. Ao contrário da série de 15 minutos, a captura de 9 horas mostra claramente padrões macroscópicos. Vemos períodos de baixa atividade (provavelmente durante a madrugada ou fora do horário comercial) e períodos de tráfego intenso e sustentado. Essa visão em escala maior é crucial, pois um modelo treinado apenas no período de alta atividade falharia em prever os períodos de baixa, e vice-versa.

Sobre a decomposição na Figura 7, a componente de tendência agora mostra um padrão diário muito claro, com o tráfego começando baixo, aumentando para um pico e depois diminuindo. Com um período de uma hora (3600 segundos), a componente sazonal é muito mais pronunciada e regular do que a sazonalidade de 60 segundos observada no dataset menor. Isso sugere que, além dos padrões minuto a minuto, existem padrões de comportamento que se repetem a cada hora.

O decaimento da autocorrelação na Figura 8 é ainda mais lento do que antes, o que é esperado para uma série com uma tendência diária tão forte. Já o gráfico da PACF agora mostra uma história diferente. Além do forte pico no *lag* 1, vemos outros picos menores e significativos em lags maiores. Isso indica que, para prever o tráfego, o modelo pode se beneficiar de olhar para o passado em diferentes escalas de tempo (o último minuto, a última hora, etc.), reforçando a importância de testar janelas de `look_back` maiores e mais variadas.

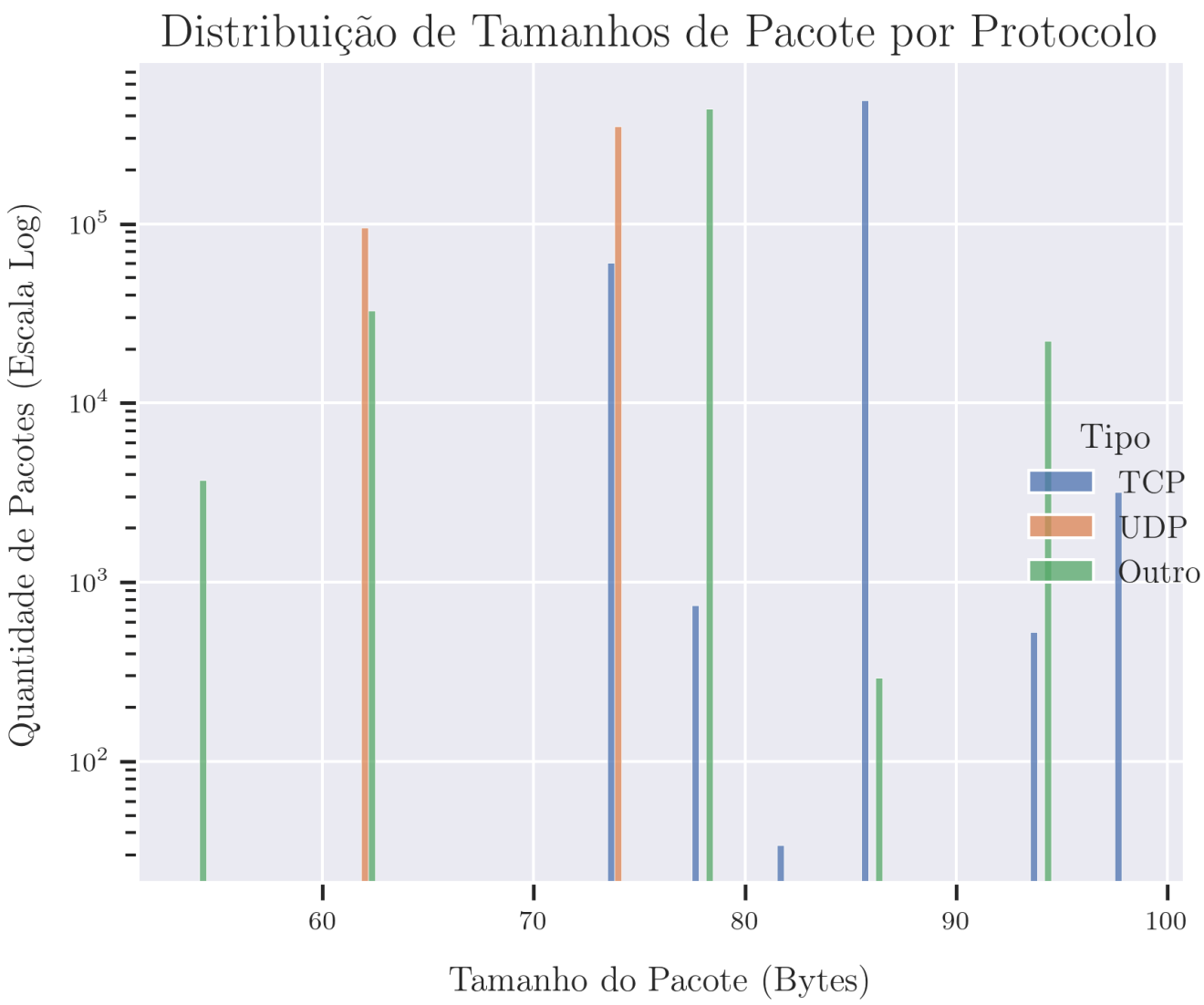

Figura 5: Distribuição de tamanhos de pacote para os protocolos TCP e UDP em `200701011800.dump.gz`. A frequência (eixo Y) está em escala logarítmica para melhor visualização.

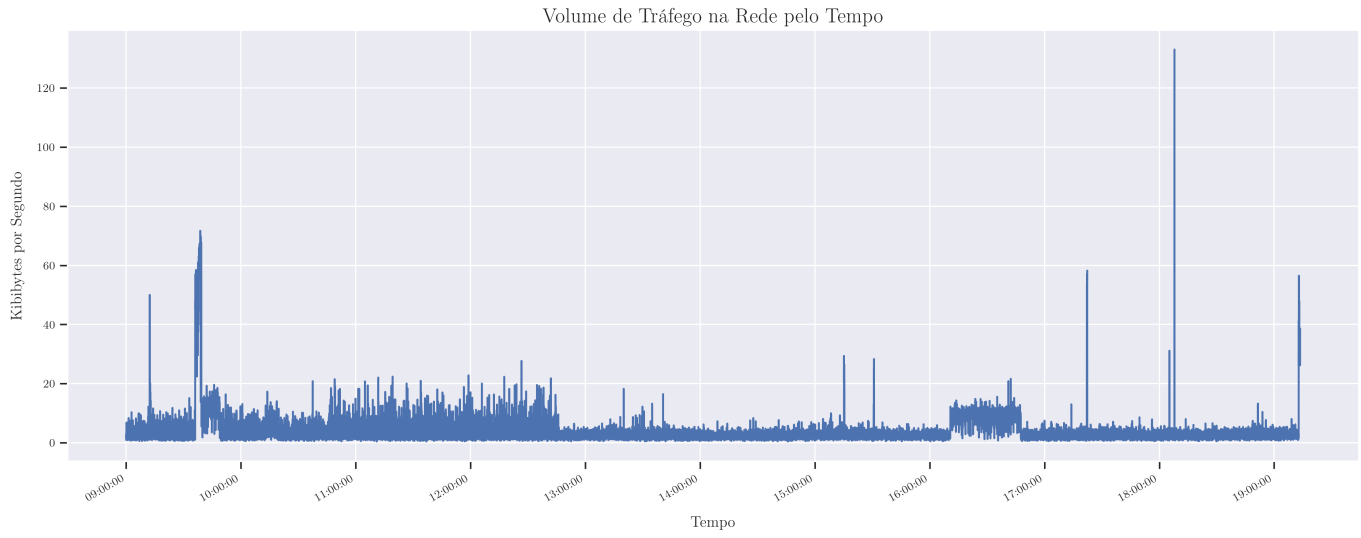

Figura 6: Série temporal do volume de tráfego agregado por segundo em `200701011800.dump.gz`.

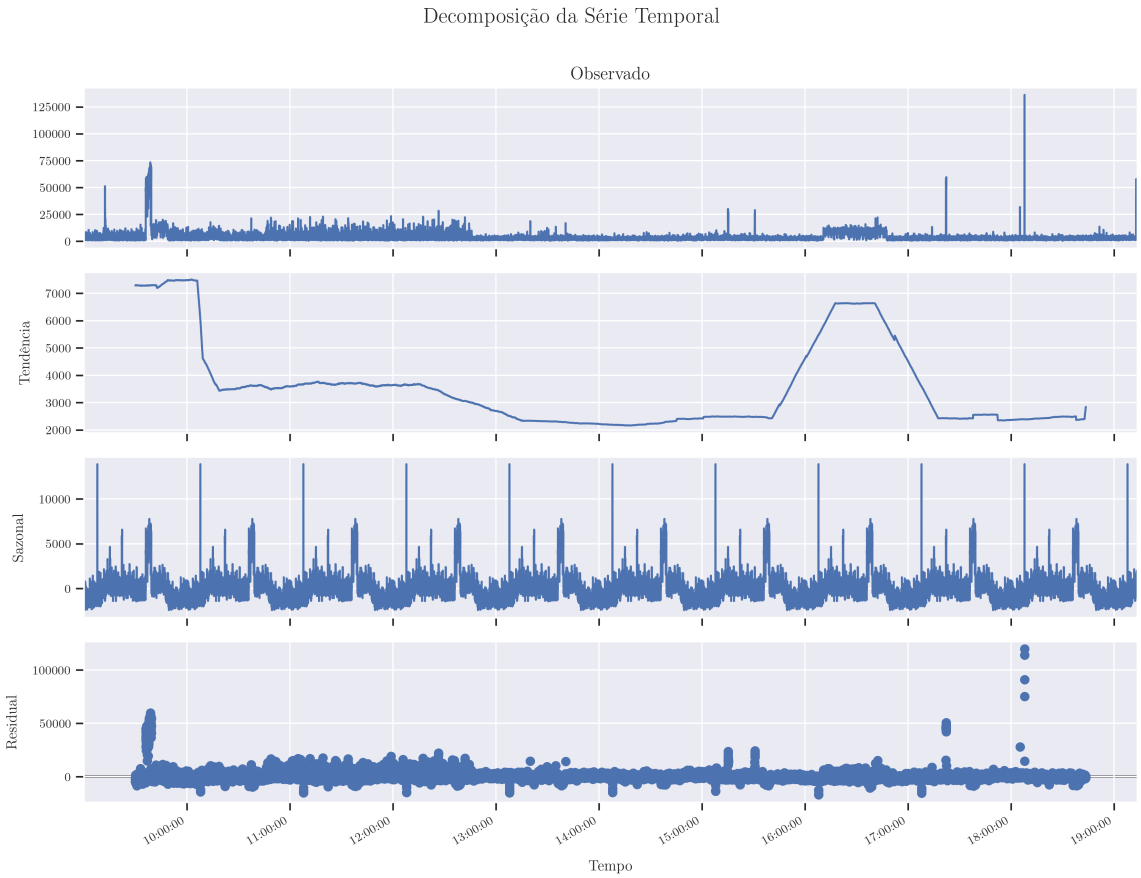

Figura 7: Decomposição da série temporal em tendência, sazonalidade e resíduos, com um período sazonal de 3600 segundos para o *dataset* `200701011800.dump.gz`.

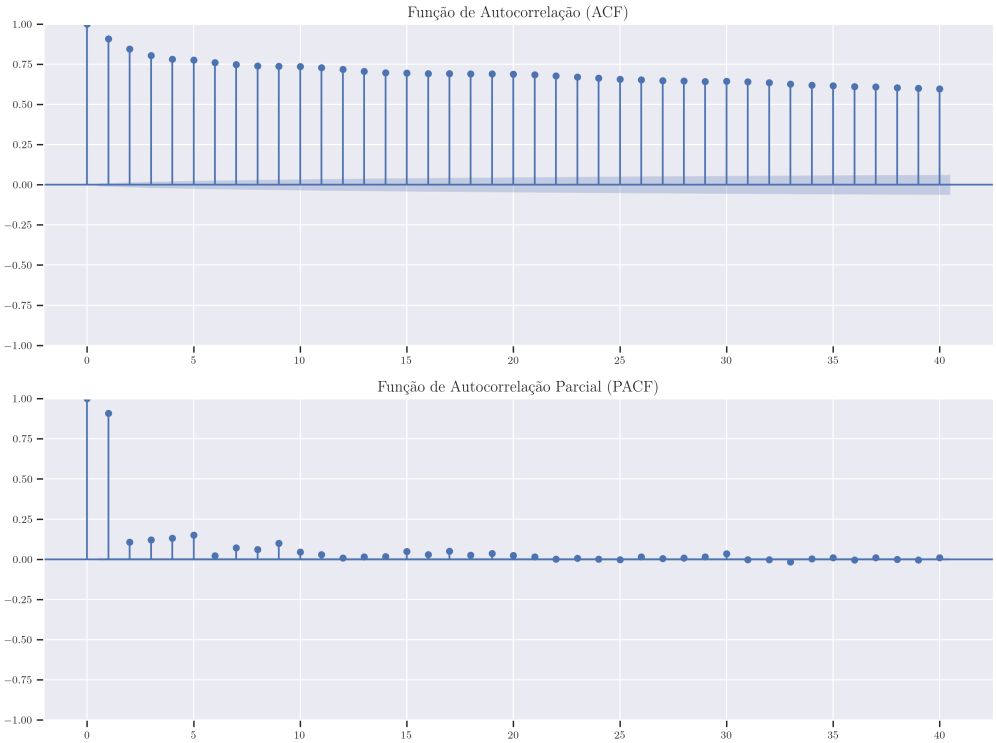

Figura 8: Funções de Autocorrelação (ACF) e Autocorrelação Parcial (PACF) para a série temporal de `200701011800.dump.gz`.

3 Modelamento Preditivo

Para a previsão do tráfego, foi implementado um modelo de Redes Neurais Recorrentes do tipo LSTM (*Long Short-Term Memory*), conforme especificado. A arquitetura adotada foi enxuta, consistindo em uma camada LSTM com 50 unidades, seguida por uma camada Densa com uma única saída para prever o valor do próximo segundo. A preparação dos dados envolveu a normalização da série de `bytes_per_second` para o intervalo [0, 1] e a criação de sequências de entrada utilizando uma janela deslizante (*look-back*).

3.1 Resultados com o Dataset Curto (15 minutos)

Inicialmente, foram conduzidos experimentos com o *dataset* `200701251400.dump.gz`, que abrange um período de 15 minutos de captura. Foram testados quatro tamanhos de janela distintos: 10, 20, 30 e 60 segundos, com 50 épocas de treinamento e um *batch size* de 64. A performance de cada modelo foi avaliada no conjunto de teste utilizando a métrica de Erro Quadrático Médio (MSE).

Tabela 3: Resultados de MSE para diferentes configurações de *look-back* com o *dataset* de 15 minutos. O modelo com `look_back=20` apresentou a melhor performance.

Look Back	MSE
10	11242733286.49
20	10432004668.86
30	11655084093.13
60	11419691841.78

Os resultados na Tabela 3 indicam que a janela de `look_back=20` segundos obteve o menor MSE. Isso sugere que, para esta série temporal curta e relativamente estável, um histórico de 20 segundos forneceu o melhor equilíbrio entre contexto e ruído para a previsão.

A Figura 9 compara visualmente a previsão do melhor modelo com os dados reais. O modelo consegue capturar a tendência geral e a sazonalidade de baixa frequência do tráfego. No entanto, as previsões são notavelmente mais suaves que os dados reais, evidenciando uma dificuldade em modelar os picos de alta frequência, o que é um comportamento esperado para uma arquitetura simples em uma série com ruído.

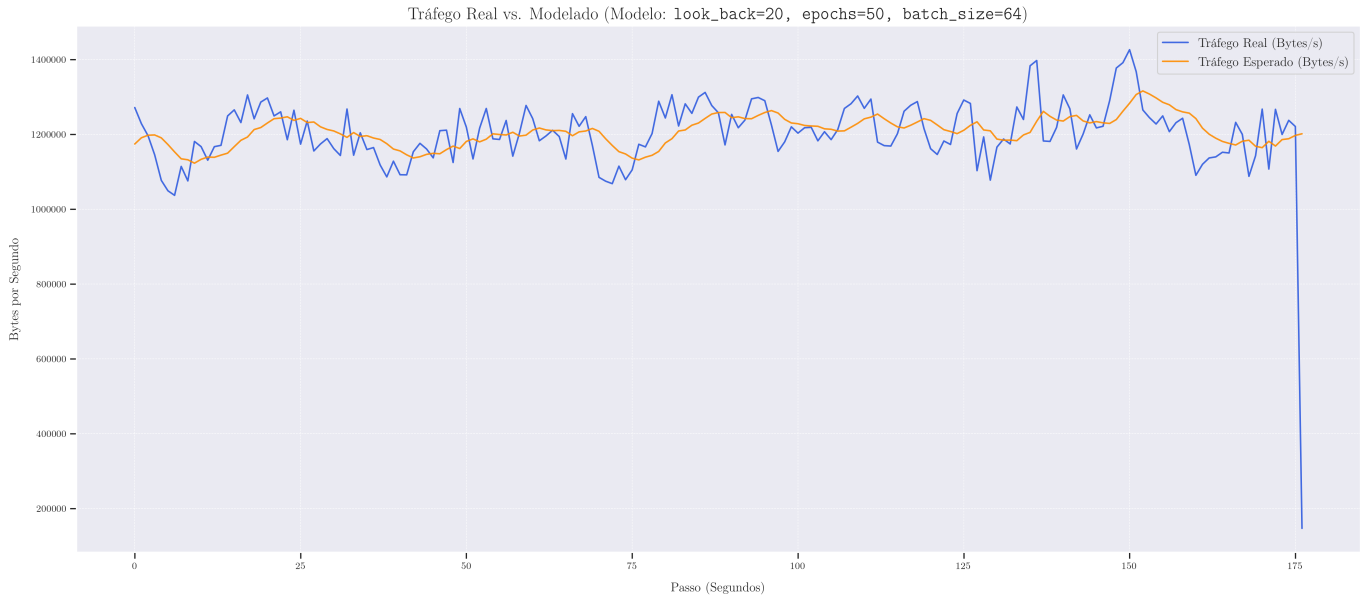

Figura 9: Comparação entre os valores reais e previstos pelo melhor modelo (`look_back=20`) no *dataset* de 15 minutos. O modelo segue a tendência, mas suaviza as flutuações de alta frequência.

3.2 Resultados com o Dataset Longo (9 horas)

Para avaliar o modelo em um cenário mais realista com padrões de longo prazo, os mesmos experimentos foram replicados no *dataset* `200701011800.dump.gz`, que abrange quase 9 horas.

Tabela 4: Resultados de MSE para o *dataset* de 9 horas. Novamente, a janela de `look_back=20` se mostrou a mais eficaz.

Look Back	MSE
10	6087128.23
20	5999701.95
30	6006869.72
60	6141613.94

Como mostra a Tabela 4, a janela de `look_back=20` novamente apresentou o melhor desempenho. Mais notavelmente, os valores de MSE são ordens de magnitude menores em comparação com o dataset curto. Isso ocorre porque o tráfego no dataset longo possui uma linha de base muito mais baixa e estável, com picos de tráfego sendo eventos mais raros, embora de altíssima magnitude.

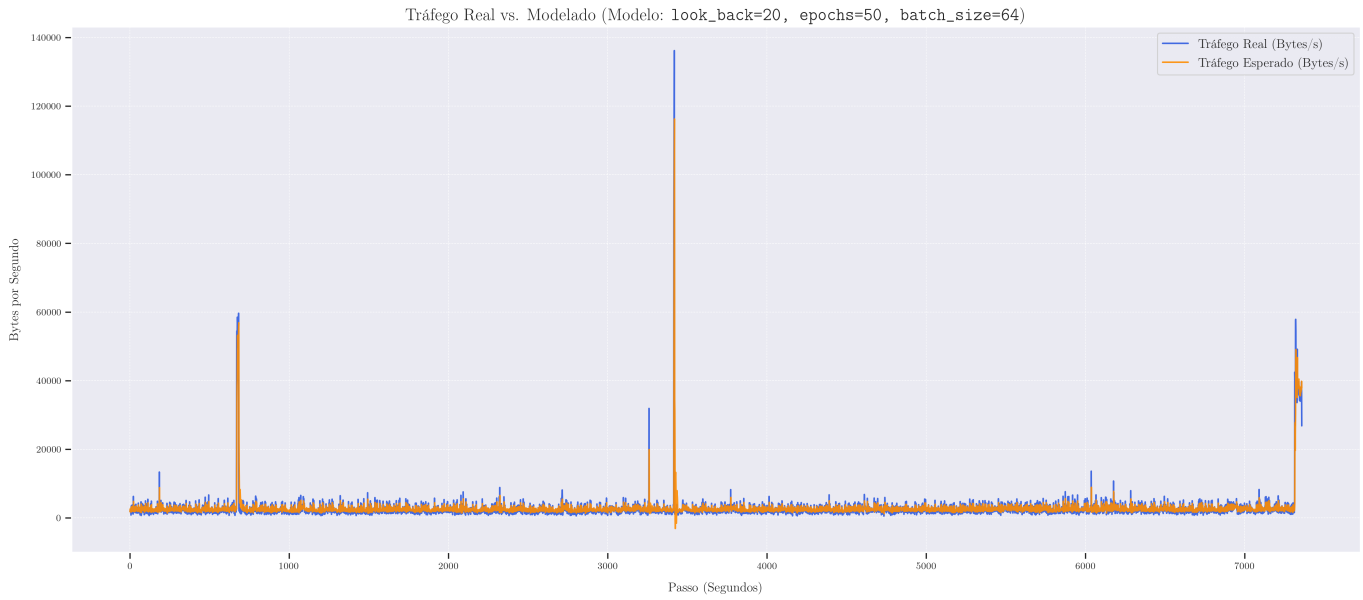

Figura 10: Comparação entre os valores reais e previstos pelo melhor modelo (`look_back=20`) no *dataset* de 9 horas. O modelo prevê a linha de base com alta precisão, mas falha completamente em antecipar os picos de tráfego.

A Figura 10 revela a principal característica do modelo neste cenário: ele se torna um excelente preditor da linha de base do tráfego. O modelo aprende a ignorar a variância extrema e prevê com precisão o comportamento do tráfego durante os longos períodos de baixa atividade. No entanto, ele falha completamente em prever os picos súbitos e massivos de tráfego. O modelo não aprendeu a antecipar esses eventos de "cisne negro", tratando-os como ruído imprevisível e mantendo sua previsão próxima da média histórica.

3.3 Resultados com Tamanho de Batch Reduzido

Adicionalmente, foi realizado um experimento no *dataset* curto com hiperparâmetros reduzidos (20 épocas e *batch size* de 32) para avaliar a sensibilidade do modelo.

Conforme observado na Tabela 5 e na Figura 11, a redução dos hiperparâmetros de treinamento não resultou em uma diferença significativa na performance ou no comportamento do modelo em comparação com a execução principal. Isso reforça a conclusão de que a performance do modelo, neste caso, é mais limitada pela sua arquitetura simples do que pela otimização fina dos hiperparâmetros de treinamento.

Tabela 5: Resultados de MSE para o *dataset* `200701251400.dump.gz` com 20 épocas e *batch size* de 32.

Look Back	MSE
10	11814694510.86
20	10691583376.90
30	11310507183.37
60	13703889350.77

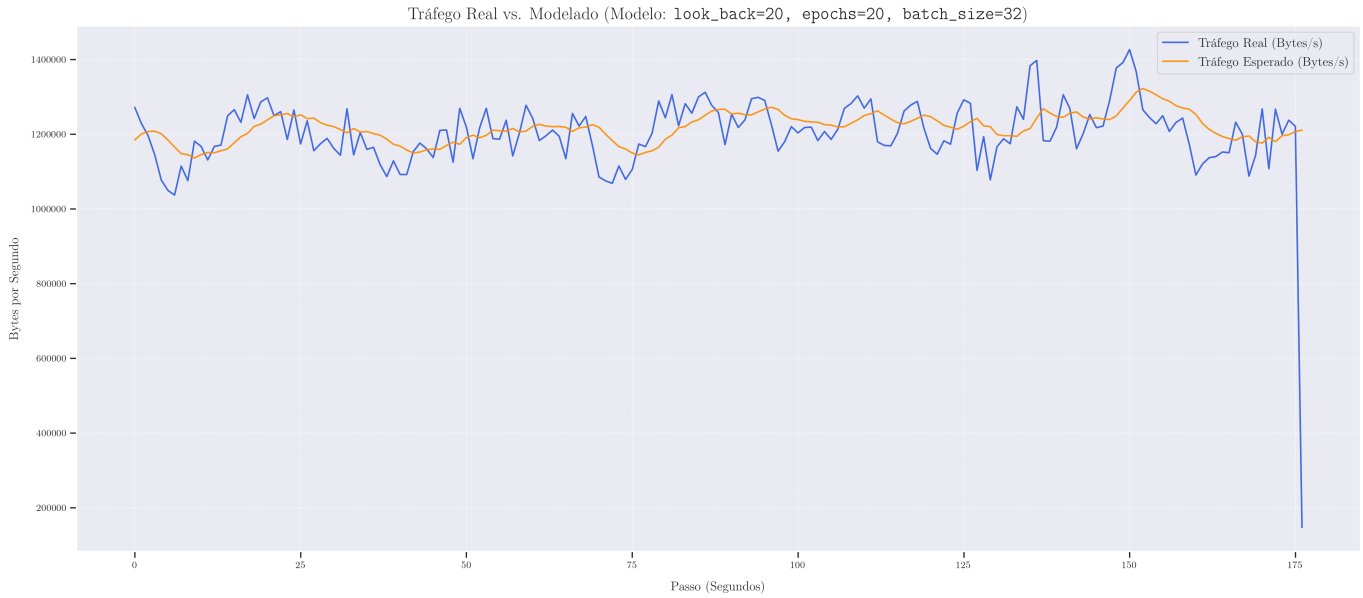

Figura 11: Previsão do modelo com *batch size* de 32 no *dataset* curto.

4 Discussão Crítica

A análise dos resultados revela um comportamento dual do modelo LSTM, que varia drasticamente com a natureza do *dataset* utilizado.

No *dataset* de 15 minutos, que exibe um tráfego relativamente estável, o modelo foi capaz de aprender a tendência geral e a sazonalidade de 60 segundos identificada na EDA. A previsão, embora mais suave que os dados reais, seguiu a trajetória da série. Isso demonstra a capacidade da arquitetura LSTM de capturar padrões de curto e médio prazo quando o comportamento da rede não apresenta variações extremas.

Em contraste, no *dataset* de 9 horas, o desempenho do modelo foi paradoxal. Por um lado, ele se tornou um preditor extremamente preciso da linha de base do tráfego. Os valores de MSE foram ordens de magnitude menores, pois o modelo aprendeu a prever com exatidão os longos períodos de baixa atividade. Por outro lado, ele falhou completamente em antecipar os *bursts* de tráfego súbitos e de alta magnitude. O modelo tratou esses picos como ruído imprevisível, mantendo sua previsão próxima da média histórica e não conseguindo se adaptar a esses eventos de "cisne negro".

Essa limitação é uma consequência direta da simplicidade da arquitetura do modelo. Uma única camada LSTM, embora eficaz para padrões regulares, não possui a capacidade de modelar a natureza inerentemente "ruidosa"e multifacetada do tráfego de internet, que é influenciado por uma infinidade de fatores, como o controle de congestionamento do TCP, a agregação de milhares de fluxos independentes e eventos externos imprevisíveis.

4.1 Conclusão

Em síntese, o projeto demonstrou com sucesso a viabilidade de utilizar uma arquitetura LSTM simples para modelar e prever o volume de tráfego de rede. A análise exploratória foi fundamental para guiar a seleção de hiperparâmetros, como a janela de *look-back*, e para interpretar o comportamento do modelo.

A principal vantagem da abordagem foi a capacidade do modelo de aprender tendências e padrões sazonais de longo prazo, resultando em uma previsão muito precisa da linha de base do tráfego. No entanto, sua principal limitação foi a incapacidade de prever os *bursts* de curta duração, que são de grande importância para aplicações como detecção de anomalias ou planejamento de capacidade em tempo real.

Para trabalhos futuros, a performance poderia ser aprimorada com a exploração de arquiteturas mais complexas. Modelos com múltiplas camadas LSTM, LSTMs bidirecionais, ou a incorporação de mecanismos de atenção poderiam capacitar o modelo a aprender relações temporais mais sofisticadas. Além disso, a engenharia de *features*, incluindo variáveis como a distribuição de protocolos por segundo ou a hora do dia como uma entrada explícita, poderia fornecer ao modelo um contexto mais rico, melhorando sua capacidade de prever os picos de tráfego e tornando-o uma ferramenta mais robusta e prática para o gerenciamento de redes.

Apêndice A: Scripts de Coleta e Processamento

```
"""
Download files listed in data sources.
"""

import asyncio
import glob
import gzip
import re
import struct
import sys
from argparse import ArgumentParser
from collections.abc import AsyncIterator, Callable, Iterable, Iterator
from concurrent.futures import ThreadPoolExecutor
from contextlib import AbstractContextManager, asynccontextmanager
from io import IOBase
from pathlib import Path
from signal import SIGINT
from traceback import print_exception
from typing import Final, Literal
from urllib.parse import ParseResult, urlparse
from urllib.request import urlopen

from tqdm import tqdm

from network_traffic_predictor.utils import cli_colors
from network_traffic_predictor.utils.plotting import colormap

# --- Core I/O Operations ---

_ = tqdm.get_lock()  # type: ignore[reportUnknownMemberType]

_TAB10_COLORS = colormap('tab10', cycle=True)


def _write_output[T: IOBase](
    *,
    operation: Literal['Downloading', 'Extracting'],
    input: Callable[[], AbstractContextManager[T]],
    input_size: Callable[[T], int | None],
    output: Path,
    quiet: bool,
    enable_colors: bool,
) -> Path:
    """
    Write input to output, updating a progress bar for the user when requested.
    """
    progress = tqdm(
        desc=f'{operation} {output.name}',
        disable=quiet,
        colour=next(_TAB10_COLORS) if enable_colors else None,
        unit='bytes',
```

```
        unit_scale=True,
    )
    try:
        with input() as input_file, open(output, 'wb') as output_file:
            progress.reset(total=input_size(input_file))

            CHUNK_SIZE: Final = 4096
            while chunk := input_file.read(CHUNK_SIZE):
                _ = output_file.write(chunk)
                _ = progress.update(len(chunk))

        return output

    except Exception as error:
        progress.display(f'{error}')
        output.unlink(missing_ok=True)
        raise error

    finally:
        progress.refresh()
        progress.close()


def _download_dump(dump_url: ParseResult, output_dir: Path, *, quiet: bool, enable_colors:
↪   bool) -> Path:
    """
    Download compressed dump file.
    """
    return _write_output(
        operation='Downloading',
        input=lambda: urlopen(dump_url.geturl()),
        input_size=lambda response: int(response.info().get('Content-Length', -1)),
        output=output_dir / Path(dump_url.path).name,
        quiet=quiet,
        enable_colors=enable_colors,
    )


def _get_gzip_uncompressed_size(gzip_path: Path) -> int | None:
    """
    Reads the last 4 bytes of a gzip file to get the uncompressed size.
    This should work for files under 4GB, as per the gzip format spec (ISIZE).
    """
    try:
        with open(gzip_path, 'rb') as file:
            _ = file.seek(-4, 2)
            return struct.unpack('<I', file.read(4))[0]
    except (struct.error, OSError):
        return None


def _extract_dump(dump_gz_path: Path, *, quiet: bool, enable_colors: bool) -> Path:
    """
```

```
    Uncompress downloaded dump file.
    """
    return _write_output(
        operation='Extracting',
        input=lambda: gzip.open(dump_gz_path, 'rb'),
        input_size=lambda _: _get_gzip_uncompressed_size(dump_gz_path),
        output=dump_gz_path.with_suffix(''),
        quiet=quiet,
        enable_colors=enable_colors,
    )


def _process_single_dump(dump_url: ParseResult, output_dir: Path, *, quiet: bool,
↪   enable_colors: bool) -> Path:
    """
    Download and extract a PCAP dump.
    """
    gzip_path = _download_dump(dump_url, output_dir, quiet=quiet,
    ↪   enable_colors=enable_colors)
    return _extract_dump(gzip_path, quiet=quiet, enable_colors=enable_colors)


# --- Data Source Parsing ---

# URLs to extract from data sources
_URL_PATTERN: Final =
↪   re.compile(r'\bhttp[s]?://([a-zA-Z0-9]+[.])+([a-zA-Z0-9_-]+[/])+[0-9]+\.dump\.gz\b')


def _resolve_data_sources(sources: Iterable[Path], *, recursive: bool = False) ->
↪   Iterator[Path]:
    """
    Markdown files with download URLs in them.
    """
    for source in sources:
        if source.is_dir():
            for file in glob.iglob('*.md', root_dir=source, recursive=recursive):
                yield source / file
        else:
            yield source


def _get_dump_urls(source_file: Path) -> Iterator[ParseResult]:
    """
    List markdown files with download URLs in them.
    """
    with open(source_file) as file:
        for line in file:
            for url_match in _URL_PATTERN.finditer(line):
                yield urlparse(url_match.group(0))


# --- Main Asynchronous Logic ---
```

```python
@asynccontextmanager
async def _with_no_wait_task_group() -> AsyncIterator[asyncio.TaskGroup]:
    """
    Create a task group that exits immediately on exceptions.
    """
    with ThreadPoolExecutor() as executor:
        loop = asyncio.get_running_loop()
        loop.set_default_executor(executor)

        try:
            async with asyncio.TaskGroup() as task_group:
                yield task_group
        finally:
            executor.shutdown(wait=True, cancel_futures=False)


async def _process_all_dumps(sources: Iterable[Path], *, quiet: bool, enable_colors: bool =
↪   True) -> int:
    """
    Download and extract all dump files from all sources listed.
    """
    tasks: list[asyncio.Task[Path]] = []

    async with _with_no_wait_task_group() as task_group:
        for source in sources:
            output_dir = source.parent
            for url in _get_dump_urls(source):
                coro = asyncio.to_thread(
                    _process_single_dump,
                    url,
                    output_dir,
                    quiet=quiet,
                    enable_colors=enable_colors,
                )
                tasks.append(task_group.create_task(coro))

    error_count = 0
    for task in tasks:
        if (error := task.exception()) is not None:
            error_count += 1
            if not quiet:
                print_exception(error)

    return -error_count


def main() -> int:
    """
    Download and extract PCAP files from specified sources.
    """
```

```
    parser = ArgumentParser('download', description='Download and extract PCAP files from
    ↪ specified sources.')
    _ = parser.add_argument('source', nargs='+', type=Path, help='File or directory to
    ↪ search for data urls.')
    _ = parser.add_argument('-r', '--recursive', action='store_true', help='Recurse into the
    ↪ SOURCE directories.')
    _ = parser.add_argument('-q', '--quiet', action='store_true', help="Don't display
    ↪ progress.")
    _ = cli_colors.add_color_option(parser)

    args = parser.parse_intermixed_args()

    data_sources = _resolve_data_sources(args.source, recursive=args.recursive)
    if not data_sources:
        print('No valid source files found.', file=sys.stderr)
        return 1

    try:
        return asyncio.run(_process_all_dumps(data_sources, quiet=args.quiet,
        ↪ enable_colors=args.color))
    except KeyboardInterrupt:
        return SIGINT


if __name__ == '__main__':
    sys.exit(main())
```

Trecho de Código A.1: Script `download.py` para download e extração dos arquivos PCAP.

```
"""
Generate parquet file from raw PCAP data.
"""

import socket
import sys
from argparse import ArgumentParser
from io import BytesIO
from pathlib import Path
from signal import SIGINT
from typing import Literal
from warnings import warn

import dpkt
import polars as pl
from pcap_parallel import PCAPParallel

from network_traffic_predictor.utils import cli_colors
from network_traffic_predictor.utils.schemas import PACKET_WITHOUT_ID_SCHEMA


def _worker_process_chunk(file_handle: BytesIO) -> pl.DataFrame:
    """
    Transform a chunk of the PCAP file into structured data via Polars.
```

```
    """
    family: list[Literal['IPv4', 'IPv6']] = []
    timestamps: list[int] = []
    source_ips: list[str] = []
    dest_ips: list[str] = []
    protocols: list[int] = []
    sizes: list[int] = []
    source_ports: list[int | None] = []
    dest_ports: list[int | None] = []
    types: list[Literal['TCP', 'UDP', 'Outro']] = []

    for timestamp, buf in dpkt.pcap.Reader(file_handle):
        try:
            eth = dpkt.ethernet.Ethernet(buf)
        except (dpkt.dpkt.UnpackError, AttributeError) as error:
            warn(f'{error}', stacklevel=1)
            continue

        if isinstance(eth.data, dpkt.ip.IP):
            address_family = socket.AF_INET
            family.append('IPv4')
        elif isinstance(eth.data, dpkt.ip6.IP6):
            address_family = socket.AF_INET6
            family.append('IPv6')
        else:
            continue

        ip = eth.data
        timestamps.append(int(timestamp * 1e9))
        source_ips.append(socket.inet_ntop(address_family, ip.src))
        dest_ips.append(socket.inet_ntop(address_family, ip.dst))
        protocols.append(ip.p)
        sizes.append(len(buf))

        match ip.data:
            case dpkt.tcp.TCP() as tcp:
                types.append('TCP')
                source_ports.append(tcp.sport)
                dest_ports.append(tcp.dport)
            case dpkt.udp.UDP() as udp:
                types.append('UDP')
                source_ports.append(udp.sport)
                dest_ports.append(udp.dport)
            case _:
                types.append('Outro')
                source_ports.append(None)
                dest_ports.append(None)

    return pl.from_dict(
        {
            'Timestamp': timestamps,
            'Source IP': source_ips,
            'Destination IP': dest_ips,
```

```
            'Protocol': protocols,
            'Size (bytes)': sizes,
            'Source Port': source_ports,
            'Destination Port': dest_ports,
            'Type': types,
            'IP Family': family,
        },
        schema=PACKET_WITHOUT_ID_SCHEMA,
        strict=True,
    )


def _process_pcap_parallel(pcap_path: Path, *, quiet: bool) -> pl.DataFrame:
    """
    Parses a PCAP file in parallel using the pcap-parallel library.
    WARNING: This loads the entire PCAP file into memory.
    """
    if not quiet:
        print(f'Processing {pcap_path.name} with pcap-parallel...')

    ps = PCAPParallel(str(pcap_path), callback=_worker_process_chunk)
    df = pl.concat(future.result() for future in ps.split())
    return df.with_row_index('Packet', offset=1)


def main() -> int:
    """
    Generate parquet file from raw PCAP data.
    """
    parser = ArgumentParser('process', description='Generate parquet file from raw PCAP
    ↪ data.')
    _ = parser.add_argument('pcap_file', type=Path, help='Raw file to be processed into
    ↪ structured parquet.')
    _ = parser.add_argument('-q', '--quiet', action='store_true', help="Don't display
    ↪ progress.")
    _ = cli_colors.add_color_option(parser)

    args = parser.parse_intermixed_args()
    try:
        pcap_file: Path = args.pcap_file
        output_file = pcap_file.parent / f'{pcap_file.stem}.parquet'

        df = _process_pcap_parallel(pcap_file, quiet=args.quiet)
        df.write_parquet(output_file)
        print(df)

        return 0
    except KeyboardInterrupt:
        return SIGINT


if __name__ == '__main__':
    sys.exit(main())
```

Trecho de Código A.2: Script `preprocess.py` para processamento paralelo do arquivo PCAP e geração do dataset em formato Parquet.

Apêndice B: Scripts de Análise e Treinamento

```
"""
Perform Exploratory Data Analysis (EDA) on the preprocessed network data.
"""

import os
import sys
from argparse import ArgumentParser
from collections.abc import Iterator
from contextlib import contextmanager
from pathlib import Path
from signal import SIGINT
from typing import IO, Final

# pyright: reportUnknownMemberType=false, reportUnknownArgumentType=false
import matplotlib.dates as mdates
import matplotlib.pyplot as plt
import polars as pl
import seaborn as sns
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.seasonal import seasonal_decompose

from network_traffic_predictor.utils import cli_colors
from network_traffic_predictor.utils.plotting import set_theme
from network_traffic_predictor.utils.schemas import PACKET_SCHEMA

set_theme()


def _extract_traffic(df: pl.DataFrame, *, log: IO[str]) -> pl.DataFrame:
    """
    Aggregate all traffic in each second and sum the trafficked bytes.
    """
    print('Aggregating data to create bytes_per_second time series...', file=log)
    return (
        df.lazy()
        .select('Timestamp', 'Size (bytes)')
        .group_by_dynamic('Timestamp', every='1s')
        .agg(bytes_per_second=pl.col('Size (bytes)').sum())
        .collect()
    )


def _descriptive_stats(traffic_per_sec: pl.DataFrame, basename: Path, *, log: IO[str]) ->
↪   None:
    """
    Generates a bytes_per_second time series and calculates its descriptive statistics.
    """
    print('--- Descriptive Statistics for Bytes per Second ---', file=log)
```

```
    stats = traffic_per_sec.select(bps=pl.col('bytes_per_second')).describe()
    print(stats, file=log)
    stats.to_pandas().to_latex(
        buf=basename.parent / f'{basename.stem}.describe.tex',
        float_format='%.1f',
        index=False,
        escape=True,
    )


def _protocol_histogram(df: pl.DataFrame, basename: Path, *, log: IO[str]) -> None:
    """
    Creates and saves a histogram of packet sizes for TCP and UDP protocols.
    """
    _ = plt.figure(figsize=(12, 7))

    _ = sns.displot(
        df.select('Size (bytes)', Tipo='Type').filter(pl.col('Size (bytes)') <=
        ↪  100).to_pandas(),
        x='Size (bytes)',
        hue='Tipo',
        kind='hist',
        discrete=True,
        binwidth=1,
        multiple='dodge',
    )
    plt.gca().set_yscale('log')

    _ = plt.title('Distribuição de Tamanhos de Pacote por Protocolo', fontsize=16)
    _ = plt.xlabel('Tamanho do Pacote (Bytes)', fontsize=12)
    _ = plt.ylabel('Quantidade de Pacotes (Escala Log)', fontsize=12)
    plt.tight_layout()

    output_path = basename.parent / f'{basename.stem}.protocol_dist.png'
    plt.savefig(output_path, dpi=300)
    print(f'Protocol distribution hstogram saved to {output_path}.', file=log)
    plt.close()


def _time_series(traffic_per_sec: pl.DataFrame, basename: Path, *, log: IO[str]) -> None:
    """
    Simple time series plot.
    """
    fig = plt.figure(figsize=(15, 6))
    _ = plt.plot(traffic_per_sec['Timestamp'], traffic_per_sec['bytes_per_second'] / 1024)
    _ = plt.title('Volume de Tráfego na Rede pelo Tempo', fontsize=16)
    _ = plt.xlabel('Tempo', fontsize=12)
    _ = plt.ylabel('Kibibytes por Segundo ', fontsize=12)

    fig.axes[-1].xaxis.set_major_formatter(mdates.DateFormatter('%H:%M:%S'))
    # fig.axes[-1].xaxis.set_major_locator(mdates.SecondLocator(bysecond=(0, 30)))
    fig.autofmt_xdate()  # auto-rotate date labels
```

```
    output_path = basename.parent / f'{basename.stem}.time_series.png'
    plt.savefig(output_path, dpi=300)
    print(f'Time series saved to {output_path}', file=log)
    plt.close()


def _time_series_decomposition(traffic_per_sec: pl.DataFrame, basename: Path, *, log:
↪   IO[str]) -> None:
    """
    Decomposition of the time series using `statsmodel`.
    """
    PERIOD: Final = 60 * 60
    print(f'Time series decomposition: assuming period of {PERIOD} seconds.', file=log)
    if len(traffic_per_sec) < 2 * PERIOD:
        print(f'Warning: Time series is too short for seasonal decomposition with
        ↪   period={PERIOD}. Skipping.', file=log)
        return

    traffic_pd = traffic_per_sec.select('Timestamp',
    ↪   'bytes_per_second').to_pandas().set_index('Timestamp')
    decomposition = seasonal_decompose(traffic_pd['bytes_per_second'], model='additive',
    ↪   period=PERIOD)
    fig = decomposition.plot()

    fig.set_size_inches(12, 9)
    _ = fig.suptitle('Decomposição da Série Temporal', y=1.01, fontsize=16)
    _ = fig.axes[0].set_title('Observado')
    _ = fig.axes[1].set_ylabel('Tendência')
    _ = fig.axes[2].set_ylabel('Sazonal')
    _ = fig.axes[3].set_ylabel('Residual')

    _ = fig.axes[-1].set_xlabel('Tempo')
    fig.axes[-1].xaxis.set_major_formatter(mdates.DateFormatter('%H:%M:%S'))
    # fig.axes[-1].xaxis.set_major_locator(mdates.SecondLocator(bysecond=(0, 30)))
    fig.autofmt_xdate()  # auto-rotate date labels
    plt.tight_layout()

    output_path = basename.parent / f'{basename.stem}.decomposition.png'
    plt.savefig(output_path, dpi=300)
    print(f'Time series decomposition saved to {output_path}', file=log)
    plt.close()


def _autocorrelation(traffic_per_sec: pl.DataFrame, basename: Path, *, log: IO[str]) ->
↪   None:
    """
    Generate ACF and PACF plots to understand correlations.s
    """
    traffic_pd = traffic_per_sec.select('Timestamp',
    ↪   'bytes_per_second').to_pandas().set_index('Timestamp')

    fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 9))
    _ = plot_acf(traffic_pd['bytes_per_second'], ax=ax1, lags=40)
```

```
    ax1.set_title('Função de Autocorrelação (ACF)')
    _ = plot_pacf(traffic_pd['bytes_per_second'], ax=ax2, lags=40)
    ax2.set_title('Função de Autocorrelação Parcial (PACF)')
    plt.tight_layout()

    output_path = basename.parent / f'{basename.stem}.autocorrelation.png'
    plt.savefig(output_path, dpi=300)
    print(f'Autocorrelation saved to {output_path}', file=log)
    plt.close(fig)


def _run_all_analysis(*, input_file: Path, output_dir: Path, log: IO[str]) -> None:
    """
    Apply all proposed analysis on the data.
    """
    output_dir.mkdir(parents=True, exist_ok=True)
    basename = output_dir / input_file.name

    print(f'Analyzing {input_file.name}:', file=log)
    df = pl.read_parquet(input_file, schema=PACKET_SCHEMA)

    traffic_per_sec = _extract_traffic(df, log=log)
    _descriptive_stats(traffic_per_sec, basename, log=log)

    _protocol_histogram(df, basename, log=log)
    _time_series(traffic_per_sec, basename, log=log)
    _autocorrelation(traffic_per_sec, basename, log=log)
    _time_series_decomposition(traffic_per_sec, basename, log=log)


@contextmanager
def _open_log_file(*, quiet: bool) -> Iterator[IO[str]]:
    """
    Handle the log file for verbose and quiet output.
    """
    if quiet:
        with open(os.devnull, 'w') as null:
            yield null
    else:
        yield sys.stderr


def main() -> int:
    """
    Generate and visualize descriptive statistics from a processed parquet file.
    """
    parser = ArgumentParser(
        'analysis',
        description='Generate and visualize descriptive statistics from a processed parquet
        ↪  file.',
    )
    _ = parser.add_argument('parquet_file', type=Path, help='Generated by the preprocess
    ↪  script.')
```

```
    _ = parser.add_argument(
        '-o',
        '--output-dir',
        type=Path,
        default=Path('results'),
        help='Directory to save the output plots.',
    )
    _ = parser.add_argument('-q', '--quiet', action='store_true', help="Don't display
    ↪ progress.")
    _ = cli_colors.add_color_option(parser)

    args = parser.parse_intermixed_args()
    try:
        with _open_log_file(quiet=args.quiet) as log:
            _run_all_analysis(input_file=args.parquet_file, output_dir=args.output_dir,
            ↪  log=log)

        return 0
    except KeyboardInterrupt:
        return SIGINT


if __name__ == '__main__':
    sys.exit(main())
```

Trecho de Código B.1: Script `analysis.py` para a Análise Exploratória dos Dados.

```
"""
Trains an LSTM model to forecast network traffic based on the preprocessed data.
This script handles data preparation, model training, evaluation, and result generation.
"""

import os
import random
import sys
from argparse import ArgumentParser
from pathlib import Path
from signal import SIGINT
from typing import Any, TypedDict

# pyright: reportMissingModuleSource=false
# pyright: reportUnknownArgumentType=false
# pyright: reportUnknownMemberType=false
# pyright: reportUnknownVariableType=false
import matplotlib.pyplot as plt
import numpy as np
import polars as pl
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras import Model
from tensorflow.keras.layers import LSTM, Dense, Input
from tensorflow.keras.models import Sequential, save_model
from tensorflow.random import set_seed
```

```
from network_traffic_predictor.utils import cli_colors
from network_traffic_predictor.utils.plotting import set_theme

os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
set_theme()

random.seed(0x80251BC46E68743C)
np.random.seed(0x8A8EAB5A)
set_seed(0x985E79749991772482D363C286ADBD2A)


def _extract_traffic_series(df: pl.DataFrame) -> pl.Series:
    """
    Aggregates packet data into a bytes_per_second time series.
    """
    print('Aggregating data to create bytes_per_second time series...')
    return (
        df.lazy()
        .group_by_dynamic('Timestamp', every='1s')
        .agg(bytes_per_second=pl.col('Size (bytes)').sum())
        .collect()['bytes_per_second']
    )


def _create_sequences(data: np.ndarray, look_back: int) -> tuple[np.ndarray, np.ndarray]:
    """
    Creates sliding window sequences for the LSTM model.
    """
    xx: list[np.ndarray] = []
    y: list[np.ndarray] = []
    for i in range(len(data) - look_back):
        xx.append(data[i : (i + look_back), 0])
        y.append(data[i + look_back, 0])
    return np.array(xx), np.array(y)


class _Results(TypedDict):
    """
    Traning and evaluation results for the model.
    """

    look_back: int
    mse: float
    model: 'Model[Any, Any]'
    y_test_orig: np.ndarray
    test_predict_orig: np.ndarray


def _train_and_evaluate_model(traffic_series: pl.Series, look_back: int, epochs: int,
↪   batch_size: int) -> _Results:
    """
    Handles the full pipeline for a single look_back configuration:
```

```
    scaling, splitting, model building, training, and evaluation.
    """
    print(f'--- Processing with look_back = {look_back} ---')
    # Scale the data
    scaler = MinMaxScaler(feature_range=(0, 1))
    scaled_data = scaler.fit_transform(traffic_series.to_numpy().reshape(-1, 1))

    # Create sequences
    xx, y = _create_sequences(scaled_data, look_back)
    xx = np.reshape(xx, (xx.shape[0], xx.shape[1], 1))

    # Split into 80% training / 20% test sets
    train_size = int(len(xx) * 0.8)
    X_train, X_test = xx[:train_size], xx[train_size:]
    y_train, y_test = y[:train_size], y[train_size:]
    print(f'Train/Test split: {len(X_train)} / {len(X_test)} samples.')

    # Build LSTM model
    model = Sequential([
        Input((look_back, 1)),
        LSTM(50),
        Dense(1),
    ])
    model.compile(optimizer='adam', loss='mean_squared_error')

    # Training
    print(f'Training model for look_back={look_back}...')
    _ = model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size, verbose=0)

    # Evaluation
    test_predict_scaled = model.predict(X_test, verbose=0)
    test_predict_orig = scaler.inverse_transform(test_predict_scaled)
    y_test_orig = scaler.inverse_transform(y_test.reshape(-1, 1))

    mse = mean_squared_error(y_test_orig, test_predict_orig)
    print(f'Evaluation complete. Test MSE: {mse:.2f}')

    return {
        'look_back': look_back,
        'mse': mse,
        'model': model,
        'y_test_orig': y_test_orig,
        'test_predict_orig': test_predict_orig,
    }


def _find_best_model(*, input_file: Path, output_dir: Path, epochs: int, batch_size: int) ->
↪   None:
    output_dir.mkdir(parents=True, exist_ok=True)
    basename = output_dir / input_file.name

    df = pl.read_parquet(input_file)
    traffic_series = _extract_traffic_series(df)
```

```
    # --- Experiment Loop ---
    look_back_configs = (10, 20, 30, 60)
    results = [
        _train_and_evaluate_model(traffic_series, look_back, epochs, batch_size) for
        ↪ look_back in look_back_configs
    ]
    best_model_info = min(results, key=lambda result: result['mse'])

    # --- Generate Final Report Assets ---
    best_model_path = basename.parent /
    ↪ f'{basename.stem}.best_model.e{epochs}.b{batch_size}.keras'
    save_model(best_model_info['model'], best_model_path)
    print(f'Best model (look_back={best_model_info["look_back"]}) saved to
    ↪ {best_model_path}')

    results_df = pl.from_dict({
        'Look Back': [result['look_back'] for result in results],
        'MSE': [result['mse'] for result in results],
    })

    print('--- MSE for Different Window Configurations ---')
    print(results_df)
    results_df.to_pandas().to_latex(
        buf=basename.parent / f'{basename.stem}.mse_results.e{epochs}.b{batch_size}.tex',
        column_format='cc',
        float_format='%.2f',
        index=False,
        escape=True,
    )

    # Prediction plot
    _ = plt.figure(figsize=(18, 8))
    _ = plt.plot(best_model_info['y_test_orig'], label='Tráfego Real (Bytes/s)',
    ↪ color='royalblue')
    _ = plt.plot(
        best_model_info['test_predict_orig'], label='Tráfego Esperado (Bytes/s)',
        ↪ color='darkorange', alpha=0.9
    )
    params = f'look_back={best_model_info["look_back"]}, {epochs=}, {batch_size=}'
    _ = plt.title(rf'Tráfego Real vs. Modelado (Modelo: \texttt{{{params}}})', fontsize=16)
    _ = plt.xlabel('Passo (Segundos)', fontsize=12)
    _ = plt.ylabel('Bytes por Segundo', fontsize=12)
    _ = plt.legend(fontsize=12)
    plt.grid(True, which='both', linestyle='--', linewidth=0.5)

    plot_path = basename.parent / f'{basename.stem}.prediction.e{epochs}.b{batch_size}.png'
    plt.savefig(plot_path, dpi=300)
    print(f'Prediction plot saved to {plot_path}')
    plt.close()


def main() -> int:
```

```
    """
    Train and evaluate an LSTM model for network traffic forecasting.
    """
    parser = ArgumentParser('train', description='Train and evaluate an LSTM model for
    ↪ network traffic forecasting.')
    _ = parser.add_argument('parquet_file', type=Path, help='Path to the preprocessed
    ↪ parquet file.')
    _ = parser.add_argument(
        '-o',
        '--output-dir',
        type=Path,
        default=Path('results'),
        help='Directory to save the model, plots, and results.',
    )
    _ = parser.add_argument('-e', '--epochs', type=int, default=20, help='Number of epochs
    ↪ to run.')
    _ = parser.add_argument('-b', '--batch-size', type=int, default=32, help='Number of
    ↪ samples per epoch.')
    _ = cli_colors.add_color_option(parser)

    args = parser.parse_args()
    try:
        _find_best_model(
            input_file=args.parquet_file,
            output_dir=args.output_dir,
            epochs=args.epochs,
            batch_size=args.batch_size,
        )
        return 0
    except KeyboardInterrupt:
        return SIGINT


if __name__ == '__main__':
    sys.exit(main())
```

Trecho de Código B.2: Script `train.py` para o treinamento e avaliação do modelo LSTM.