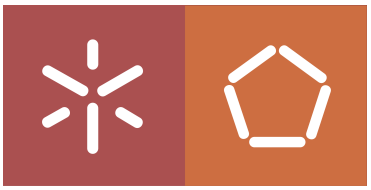


Universidade do Minho
Escola de Engenharia

Aprendizagem Profunda

Módulo 1

Relatório

MEI - 1º Ano - 2º Semestre

Trabalho realizado por:

PG55948 - Hugo Ramos
PG55951 - João Vale
PG55974 - Leonardo Barroso
PG55977 - Luís Borges
PG56015 - Tomás Oliveira

Braga, 1 de abril de 2025

1. Introdução

Este projeto foi realizado no âmbito da unidade curricular de Aprendizagem Profunda e tem como objetivo o desenvolvimento e avaliação de modelos de *Machine/Deep Learning* capazes de classificar frases em duas categorias: "AI" (texto gerado por modelos de Inteligência Artificial) e "Human" (texto escrito por humanos).

O trabalho foi desenvolvido entre várias etapas, desde a criação e preparação de *datasets* em inglês, a implementação de modelos manuais utilizando apenas bibliotecas como `numpy`, e também a implementação de modelos com arquiteturas mais complexas através do `Tensorflow/Keras`. A avaliação de cada um dos modelos foi feita com recurso a dados de treino, teste e validação (sendo esta dividida em duas abordagens, a primeira a divisão de validação do nosso *dataset* inicial e a segunda através das entradas reveladas pelo professor para as submissões).

Todo o trabalho desenvolvido está presente num repositório do `GitHub` contendo o código desenvolvido para cada um dos modelos, os datasets utilizados e os resultados das diferentes submissões.

2. Metodologias para a Criação do *Dataset*

Este relatório detalha as metodologias utilizadas na construção do dataset, abordando diferentes abordagens e os desafios encontrados em cada uma delas. Inicialmente, foi tentado construir um dataset manualmente (*Dataset* 1), mas devido a diversas dificuldades, optou-se por otimizar um dataset existente (*Dataset* 2). Por fim, decidiu-se utilizar um dataset pronto (*Dataset* 3), que apresentou melhores resultados. Além disso, ao longo do processo, levamos em consideração que as submissões seriam avaliadas com *datasets* de conteúdo científico e que os textos deveriam ter entre 100 e 120 palavras. Dessa forma, buscamos garantir que o dataset final fosse adequado para previsões precisas e alinhado com os critérios de avaliação esperados.

2.1. Dataset 1: Construção Manual do *Dataset*

A primeira tentativa para a criação do dataset envolveu a construção manual de um conjunto de dados, combinando tanto entradas humanas quanto geradas por inteligência artificial. Para obter entradas humanas, foi realizado *Web Scraping* do *website* `ARXIV.org`, utilizando a biblioteca `Beautiful Soup` para extrair o texto de artigos científicos e processar os dados extraídos. O `ARXIV.org` foi escolhido por ser um repositório amplamente reconhecido na comunidade científica, contendo uma vasta quantidade de artigos de alta qualidade em diversas áreas do conhecimento. Essa escolha garantiu que as entradas humanas fossem baseadas em conteúdos confiáveis e relevantes para os objetivos do projeto.

Além disso, também foram geradas entradas utilizando diferentes modelos de linguagem, como *ChatGPT* e *Claude*, com o objetivo de complementar o *dataset* com exemplos diversos e mais abrangentes, os prompts utilizados pediam explicitamente textos relacionados a ciências e também frases entre cem a 100 a 120 palavras.

Entretanto, essa abordagem apresentou várias dificuldades. Um dos principais problemas foi a geração de grandes quantidades de *prompts* de IA sem que houvesse repetições e garantindo contextos variados. Além disso, a dificuldade em aprender padrões a partir dos dados gerados foi significativa, pois as entradas nem sempre mantinham uma coerência adequada para a tarefa desejada. Outro desafio enfrentado foi a necessidade de filtrar e limpar os dados obtidos, visto que muitas das informações extraídas continham ruídos ou estavam mal formatadas. Diante dessas dificuldades, percebeu-se que essa metodologia não era a mais eficiente, o que levou à busca por alternativas.

2.2. Dataset 2: Otimização de um *Dataset* Existente

Após os desafios encontrados na construção manual do *dataset*, a segunda abordagem consistiu em otimizar um dataset já existente. Para isso, foi utilizado um conjunto de dados extraído do *Kaggle*, contendo cerca de 2 milhões de entradas contendo texto gerado por IA e por humanos. O primeiro passo no processo de otimização foi a filtragem do *dataset*, mantendo apenas textos de natureza científica. Essa filtragem foi realizada com base em um campo lexical com a ajuda do spacy para filtrar as palavras principais dos vários excertos, permitindo que apenas textos relevantes fossem mantidos.

Após a filtragem, foi necessário segmentar as entradas para garantir um tamanho uniforme, restringindo cada uma delas a um intervalo entre 100 e 120 palavras. Além disso, foi implementado um balanceamento entre as entradas geradas por inteligência artificial e aquelas provenientes de textos humanos, de forma a evitar viés nos dados utilizados para o treino dos modelos.

Entretanto, essa abordagem também apresentou dificuldades. A filtragem de dados exigia a inserção manual de palavras-chave para garantir a extração correta do conteúdo relevante, tornando o processo demorado e trabalhoso, como potenciais erros, devido à possibilidade de não termos *keywords* sufucientes. Além disso, muitas entradas apresentavam uma similaridade excessiva entre si, o que reduzia a diversidade do *dataset* e impactava negativamente na aprendizagem do modelo. Assim, apesar de ter sido uma evolução em relação à abordagem anterior, essa metodologia ainda possuía desafios que dificultavam sua aplicação eficiente. Por esse motivo, foi considerada uma nova, e última, abordagem.

2.3. Dataset 3: Utilização de um Dataset Existente

Por fim, a solução adotada foi utilizar um dataset pronto, disponível na plataforma *Hugging Face*. Esse conjunto de dados era menor, contendo apenas cerca de 4000 entradas, mas apresentava algumas vantagens em relação às abordagens anteriores. Primeiramente, o dataset era variado, contendo textos com diferentes contextos e estruturas, o que contribuiu para uma melhor generalização dos modelos. Além disso, utilizava entradas geradas por múltiplas *Large Language Models (LLMs)*, permitindo um conjunto de dados mais rico e representativo das capacidades de diferentes modelos de IA. Por fim, o tamanho reduzido do dataset facilitou o treino e a realização de previsões, tornando o processo mais rápido e eficiente.

Para melhorar ainda mais a qualidade do *dataset*, foi realizada uma limpeza cuidadosa dos dados, removendo erros de formatação, como espaços desnecessários, *tabs* e caracteres especiais indesejados.

Essa abordagem trouxe benefícios significativos. Devido ao menor tamanho do dataset, os modelos conseguiram executar previsões de maneira mais rápida, sem comprometer a qualidade dos resultados. Devido ao seu alto desempenho, também foi possível testar várias combinações de hiperparâmetros. Além disso, as previsões realizadas a partir desse conjunto de dados apresentaram uma precisão superior às abordagens anteriores, demonstrando que, muitas vezes, a qualidade dos dados é mais importante do que a quantidade.

3. Implementação de modelos de raíz

3.1. Preparação dos Dados

Nesta seção, apresentamos a implementação de diferentes tokenizers, utilizados para a tokenização de textos em representações numéricas, um passo essencial no processamento de linguagem natural (NLP). Os tokenizers implementados foram: *SimpleTokenizer*, *RobustTokenizer* e *AdvancedTokenizer*.

3.1.1. SimpleTokenizer

O *SimpleTokenizer* oferece uma abordagem básica para a tokenização de textos. Ele divide o texto em palavras com base num delimitador específico (por padrão, espaço) e atribui índices únicos a cada palavra com base na sua frequência.

- Utiliza `split` para segmentar palavras.
- Mantém um vocabulário limitado pelo parâmetro *num_words*.
- Pode converter textos em sequências numéricas de forma simples.

Apesar da sua simplicidade, este tokenizer pode ser sensível à pontuação e não trata bem variações na forma das palavras.

3.1.2. RobustTokenizer

O *RobustTokenizer* é uma versão mais avançada do *SimpleTokenizer*, introduzindo uma tokenização mais robusta baseada em expressões regulares.

- Usa regex `(\w+)` para capturar palavras e ignorar pontuação.
- Mantém um vocabulário limitado pela frequência de palavras.
- Permite um processamento mais confiável e menos dependente da estrutura textual.
- Converte textos para sequências numéricas de forma similar ao SimpleTokenizer.

A remoção da pontuação melhora a qualidade da tokenização, tornando-o mais eficiente para modelos de redes neurais.

3.1.3. AdvancedTokenizer

O *AdvancedTokenizer* é a versão mais sofisticada, incluindo suporte para *n-grams*, remoção de *stopwords* e um método de *padding* para garantir sequências de tamanho fixo.

- Tokeniza palavras usando regex aprimorado `(\b\w+(?:[-']\w+)?\b)`.
- Permite a remoção de *stopwords*, tornando o vocabulário mais eficiente.
- Suporta *n-grams*, melhorando a representação contextual.
- Usa *padding* e truncamento para padronizar o tamanho das sequências.
- Mantém apenas palavras que aparecem com uma frequência mínima (*min_freq*).

Este tokenizer é ideal para aplicações de NLP mais avançadas, permitindo que redes neurais lidem melhor com dependências de contexto dentro dos textos.

3.2. Implementação da *Deep Neural NetworK* (DNN)

3.2.1. Camada Embedding

A Embedding Layer é essencial para modelos que trabalham com dados categóricos, como textos, convertendo índices de palavras em vetores densos. A implementação envolve a inicialização aleatória da matriz de embeddings, a propagação direta para mapear índices para vetores correspondentes e a retropropagação que ajusta os vetores com base nos erros calculados.

3.2.2. Camada GRU

A GRU Layer (*Gated Recurrent Unit*) é uma variante das LSTMs eficiente para modelagem de sequências, possuindo um mecanismo simplificado de memória. A implementação inclui a inicialização de pesos com Xavier/Glorot, o cálculo dos gates de atualização e *reset*, a atualização do estado oculto ao longo da sequência e a propagação direta e retropropagação parcial para aprendizado dos pesos.

3.2.3. Camada Batch Normalization

A Batch Normalization Layer melhora a convergência e estabilidade do treino, normalizando cada batch de entrada. A implementação envolve o cálculo da média e variância do batch durante o treino, o uso de estatísticas acumuladas para inferência e parâmetros treináveis para escalonamento e deslocamento.

3.2.4. GlobalAveragePooling

Global Average Pooling 1D reduz a dimensionalidade dos dados agregando informações de sequência, sendo muito usada em redes convolucionais para NLP e visão computacional. Seu funcionamento baseia-se no cálculo da média ao longo da dimensão temporal e na retropropagação que distribui o erro uniformemente pela sequência.

3.2.5. Dropout Layer

A Dropout Layer é uma técnica de regularização que reduz o sobreajuste durante o treino, desativando aleatoriamente alguns neurónios. A implementação inclui a geração de uma máscara binária para dropout, a aplicação da máscara aos inputs durante o treino e a escalagem dos valores para manter a expectativa da ativação inalterada.

3.2.6. Adam Optimizer

Para garantir um treinamento eficiente, foi implementado o AdamOptimizer, um algoritmo de otimização baseado em gradientes adaptativos que combina os métodos de Momentum e RMSProp. Ele mantém momentos de primeira e segunda ordem para ajustar dinamicamente o *learning rate* de cada parâmetro. A implementação envolve a inicialização dos momentos, a atualização dos valores conforme os gradientes das camadas e a aplicação da correção de viés. O Adam melhora a convergência e estabilidade do treinamento, sendo amplamente utilizado em redes neurais profundas devido à sua capacidade de lidar bem com diferentes tipos de dados e distribuições de gradientes.

3.3. Implementação da *Recurrent Neural Network* (RNN)

A implementação de raiz de um modelo RNN (*Recurrent Neural Network*) teve por base a mesma lógica aplicada à implementação do modelo DNN, sendo a maior diferença a utilização de duas novas camadas: RNN e LSTM

3.3.1. Camada RNN

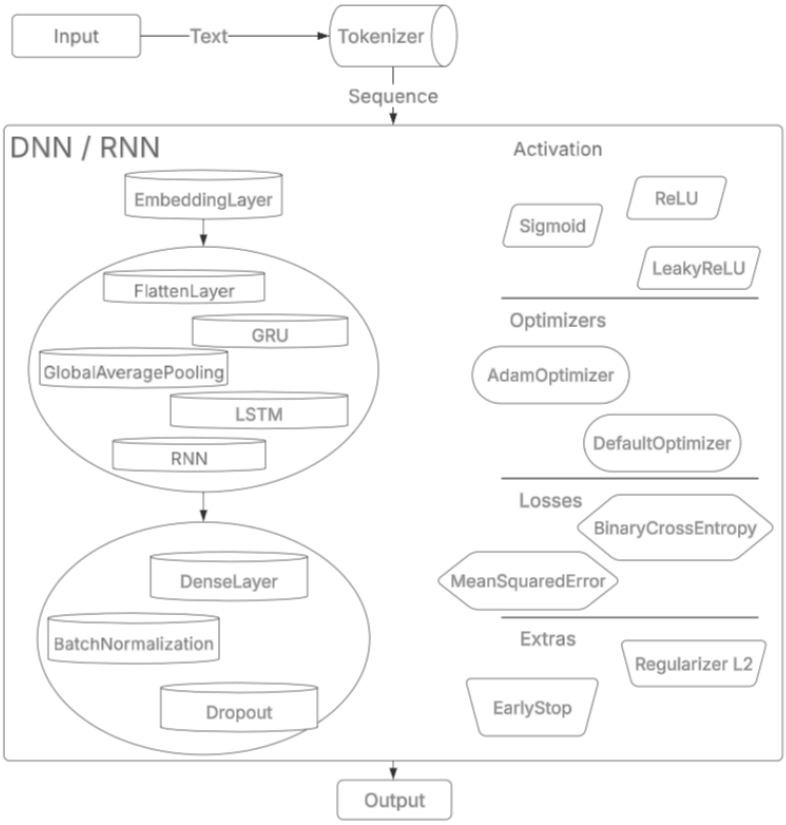
Esta é uma camada simples de uma RNN, destacando-se por fazer uso da técnica de inicialização de Xavier/Glorot, evitando assim a existência de *vanishing* e *exploding* gradients. Outro aspeto importante a mencionar é a possibilidade de usar a variável *bptt_trunc* para limitar o número de timesteps usados na parte referente à *backpropagation*, conferindo ao utilizador a possibilidade de treinar o modelo mais

rapidamente. Excluindo os aspetos mencionados acima, a implementação segue uma lógica simples, usando a função da tangente hiperbólica para fazer os cálculos necessários na parte da *forward propagation* e calculando os erros, para atualizar os gradientes, na *backward propagation*.

3.3.2. Camada LSTM

A camada LSTM representa uma implementação algo mais complexa, uma vez que faz uso de técnicas que possibilitam o armazenamento de informação a longo prazo num input de forma mais intricada. Isto é possível através da implementação de um *cell state* que, para cada frase, a cada input, guarda a informação relevante. Este vai sendo atualizado cada vez que um novo token da frase é processado, sendo para isso importantes a *input gate* (responsável por decidir que nova informação guardar no *cell state*), a *forget gate* (responsável por decidir que informação remover do *cell state*) e a *output gate*, que decide qual a informação do novo *cell state* usar como *output*. Esta implementação permite que o modelo guarde e processe apenas a informação mais relevante, providenciando, teoricamente, resultados melhores e mais eficientes.

O seguinte diagrama representa um pipeline para processamento de texto usando redes neurais profundas (DNN) e redes neurais recorrentes (RNN). O texto de entrada passa por um *tokenizer*, que converte palavras em sequências numéricas. Em seguida, essas sequências são processadas pela rede que pode usar diferentes camadas, funções de ativação (ReLU, Sigmoid), otimizadores (AdamOptimizer) e funções de perda (BinaryCrossEntropy, MeanSquaredError). Extras como Regularizer L2 e EarlyStop ajudam a evitar overfitting. O fluxo termina com uma camada de saída para a previsão final.



4. Implementação de modelos em Tensorflow e Outras Abordagens

Foram desenvolvidos diversos modelos de *Machine Learning* com recurso à biblioteca `Tensorflow/Keras`, de forma a explorar arquiteturas mais complexas. A implementação destes modelos com recurso a esta biblioteca permitiu tirar partido de ferramentas avançadas de pré-processamento e treino, bem como da utilização de *embeddings* e de camadas recorrentes.

4.1. Preparação dos Dados

Foram exploradas duas abordagens principais para transformar o texto em formatos compatíveis com redes neuronais.

4.1.1. Tokenizer (Keras)

Foi utilizado o `Tokenizer` do `Keras` para converter os textos em sequências de inteiros, onde cada inteiro representa um token (palavra) atribuído com base na frequência de ocorrência.

- As palavras mais frequentes são mapeadas para índices mais baixos.
- Foi aplicado padding para garantir que todas as sequências tenham o mesmo comprimento, permitindo o processamento em batch.
- Este formato é essencial para ser utilizado em camadas de *Embedding*.

4.1.2. TF-IDF (TfidfVectorizer Sklearn)

Foi utilizado também o `TfidfVectorizer` do `Sklearn`, que representa cada documento como um vetor numérico com base na frequência e importância das palavras. Este não mantém a ordem das palavras, sendo mais adequada para arquiteturas que não dependem de sequência.

4.1.3. Embeddings

Foram utilizadas duas abordagens para representar semanticamente os textos:

- **Camada Embedding do Keras:** Transforma os índices inteiros das palavras em vetores densos de dimensão fixa, que capturam relações semânticas básicas.
- **BERT (SentenceTransformer):** Foi utilizada a biblioteca `sentence_transformers` para gerar embeddings contextuais de frases completas, baseados em modelos. Estes vetores foram utilizados como entrada para modelos densos.

4.2. Implementação da *Deep Neural NetworK* (DNN)

Para a implementação da *Deep Neural Network*, os melhores resultados obtidos foram utilizando uma arquitetura composta por:

- Camada Embedding
- Camada AveragePooling1D - Downscaling
- 4 Camadas Dropout
- BatchNormalization para estabilizar o treino
- 5 Camadas Dense

Para a compilação e treino deste modelo, utilizamos:

- Otimizador Adam
- EarlyStopping
- ReduceLROnPlateau
- 20 epochs

4.3. Implementação da *Recurrent Neural Network* (RNN)

Para a implementação da *Recurrent Neural Network*, obtivemos os melhores resultados para uma arquitetura utilizando *Long Short-Term Memory* (LSTM). A arquitetura utilizada foi constituída por:

- Camada Embedding
- 2 Camadas LSTM
- 3 Camadas Dropout
- BatchNormalization para establizar o treino
- 2 Camadas Dense

Para a compilação e treino deste modelo, utilizamos:

- Otimizador Adam
- EarlyStopping
- ReduceLROnPlateau
- 15 epochs

4.4. Implementação da *Convolutional Neural NetworK* (CNN)

Para a implementação da *Convolutional Neural Network*, os melhores resultados obtidos foram utilizando uma arquitetura composta por:

- Camada Embedding
- 2 Camadas Conv1D
- 2 Camadas MaxPooling1D
- 2 Camadas Dropout
- Camada Flatten para transformar a saída 2D em 1D para a camada Dense
- BatchNormalization para estabilizar o treino
- 3 Camadas Dense

Para a compilação e treino deste modelo, utilizamos:

- Otimizador AdamW
- EarlyStopping
- ReduceLROnPlateau
- 50 epochs

4.5. Implementação do *Zero-Shot Learning*

Para a implementação do *Zero-Shot Learning*, utilizamos diferentes *Large Language Models (LLMs)* por meio de chamadas às suas respectivas APIs.

Para garantir uma abordagem estruturada, utilizamos um *prompt* específico que definiu um especialista na distinção entre textos humanos e gerados por IA. Esse *prompt* incluía uma estrutura clara de entrada e saída de dados, estabelecendo um formato preciso para a classificação das respostas. Dessa forma, asseguramos a consistência dos resultados e a padronização das previsões, permitindo uma melhor avaliação do desempenho dos modelos. Fomos ajustando os *prompts* com base nos resultados obtidos, de modo a tornar o mais claro possível para a LLM a nossa intenção.

No nosso projeto, realizamos chamadas a múltiplas LLMs para obter respostas variadas e diversificadas. Os modelos utilizados foram Gemini *(gemini-2.0-flash)*, Claude *(claude-3-7-sonnet-20250219)* e DeepSeek *(DeepSeek-V3)*. A única *API* paga utilizada foi a do modelo *Claude*, enquanto as demais foram acessadas por meio de versões gratuitas. O Gemini fornece acesso gratuito direto às suas APIs enquanto que para o DeepSeek utilizamos uma aplicação terceira, a *TogetherAI* que nos fornece créditos iniciais para correr o modelo.

5. Análise de Resultados Obtidos

Este capítulo apresenta a análise dos resultados obtidos a partir da aplicação de diferentes modelos de aprendizado de máquina, incluindo redes neurais profundas (DNN), redes recorrentes (RNN), redes convolucionais (CNN) e abordagens de aprendizado *zero-shot*. A avaliação dos modelos foi realizada considerando métricas de *accuracy* em teste, validação e uma *accuracy* final consolidada, definida com base no conjunto de dados de entradas e saídas reveladas pelo professor. Essa métrica é especialmente relevante, pois a prioridade não é apenas obter modelos com alto desempenho, mas também garantir que sejam eficazes na previsão desse tipo específico de entrada, devido às várias submissões que foram feitas ao longo do trabalho com o intuito de tentar prever corretamente as várias entradas.

5.1. Modelos de Raíz

Os modelos seguintes foram implementados manualmente, o que resultou em um desempenho inferior e dificuldades significativas na previsão do conjunto de dados final. Como consequência, os resultados obtidos demonstraram limitações na capacidade de generalização dos modelos.

5.1.1. DNN (*Deep Neural Network*)

Modelo	Accuracy de Teste	Accuracy de Validação	Accuracy Final
DNN	0.67	0.6815	0.4750

5.1.2. RNN (*Recurrent Neural Network*)

Modelo	Accuracy de Teste	Accuracy de Validação	Accuracy Final
RNN Simples	0.6831	0.6705	0.5750
RNN LSTM	0.6573	0.7028	0.5375

5.2. Modelos em Tensorflow e Outras Abordagens

Nesta seção, são apresentados os resultados obtidos com modelos desenvolvidos em TensorFlow e outras abordagens modernas. Contrariamente aos modelos de raiz, estes modelos demonstraram um desempenho muito superior, tanto nas redes previamente desenvolvidas quanto nas novas arquiteturas exploradas. Além disso, os modelos de aprendizagem *zero-shot* apresentaram resultados excelentes, destacando-se pela capacidade de realizar previsões precisas mesmo sem qualquer treino anterior necessário.

5.2.1. DNN (*Deep Neural Network*)

Modelo	Accuracy de Teste	Accuracy de Validação	Accuracy Final
DNN	0.9618	0.9676	0.65

5.2.2. RNN (*Recurrent Neural Network*)

Modelo	Accuracy de Teste	Accuracy de Validação	Accuracy Final
RNN Simples	0.9630	0.9753	0.6250
RNN LSTM	0.9433	0.9522	0.7375
RNN LSTM (BERT)	0.8187	0.8336	0.5875
RNN GRU	0.8841	0.8891	0.7125

5.2.3. CNN (*Convolutional Neural Network*)

Modelo	Accuracy de Teste	Accuracy de Validação	Accuracy Final
CNN	0.9766	0.9661	0.6125

5.2.4. *Zero-Shot Learning*

Modelo (LLM)	Accuracy de Teste	Accuracy de Validação	Accuracy Final
Gemini	N/A	N/A	0.7231
Claude	N/A	N/A	0.9923
DeepSeek	N/A	N/A	0.8375

6. Conclusão

O presente trabalho permitiu alcançar resultados significativos, evidenciando a eficácia da abordagem adoptada. A análise realizada demonstrou que os objectivos propostos foram cumpridos, permitindo uma compreensão mais aprofundada do tema e fornecendo insights valiosos para futuras investigações.

Um dos aspectos mais relevantes identificados foi a importância do *dataset* utilizado, dos modelos aplicados e da escolha cuidadosa dos hiperparâmetros. Verificámos que a qualidade dos dados tem um impacto direto no desempenho dos modelos, reforçando a necessidade de um tratamento criterioso. Além disso, a parametrização adequada dos modelos revelou-se essencial para otimizar os resultados obtidos.

Foram também testados vários modelos diferentes, analisando o seu desempenho em distintos cenários, o que nos permitiu compreender melhor as suas vantagens e limitações.