MODELOS DE DEEP LEARNING

PG55946 Guilherme Barbosa, PG55959 João Carvalho, PG55932 Diogo Ferreira,
PG55998 Rafael Peixoto, PG56005 Rodrigo Ralha, Universidade do Minho

Abstract

Neste trabalho, o objetivo é desenvolver modelos de *machine/deep learning* capazes de distinguir com precisão entre textos gerados por inteligência artificial e textos escritos por humanos na língua inglesa. Para isso, os modelos serão treinados com *datasets* criados pelo grupo e classificarão pequenos textos (com cerca de 120 palavras) em duas categorias: **AI** e **Human**.

O projeto envolve a implementação de modelos próprios, a utilização de *frameworks* como o *TensorFlow* e a aplicação de técnicas avançadas, como *embeddings*, *transformers* e *large language models (LLMs)*. Esses modelos, após passarem por várias fases de teste e validação, terão os seus resultados comparados para avaliar a sua eficácia na classificação de textos.

INTRODUÇÃO

Com o uso crescente de ferramentas como *chatbots* e sistemas de geração automática de texto, a capacidade de diferençar entre essas duas fontes torna-se um desafio muito importante e complexo. O trabalho prático proposto pelo o primeiro módulo de "Aprendizagem Profunda" tem como objetivo o desenvolvimento de modelos de *machine/deep learning* com o intuito de distinguir texto gerado por IA de texto escrito por seres humanos. Com este trabalho, busca-se treinar modelos capazes de, a partir de um texto com cerca de 120 palavras, classificá-lo corretamente como gerado por IA ou por um ser humano.

O trabalho encontra-se dividido em várias tarefas, incluindo a construção dos *datasets*, implementação de modelo de *deep learning* como as *Deep Neural Networks (DNN)*, as *Recurrent Neural Networks (RNN)* e a utilização de *frameworks* como o **TensorFlow**. Além disso, também será necessário explorar técnicas avançadas, como *embeddings*, *transformers* e *LLM's*, aplicadas ao problema de classificação de textos.

A avaliação dos modelo será feita por meio de validação interna, isto é, utilizando *datasets* criados pelo o próprio grupo, bem como uma validação externa fornecida pelo professor Miguel Rocha. Ao longo das três fases de submissão, os modelos serão avaliados e classificados, permitindo ao grupo acompanhar e melhorar/otimizar os resultados provenientes das duas abordagens.

CONTEXTUALIZAÇÃO

O avanço das tecnologias de inteligência artificial (IA), especialmente no domínio da geração de texto, tem revolucionado a forma como interagimos com sistemas computacionais. Modelos de linguagem de grande escala (*Large Language Models - LLMs*), como o *GPT*, o *BERT* ou mesmo sistemas mais recentes, têm demonstrado uma capacidade impressionante de produzir textos que, à primeira vista , podem ser indistinguíveis daqueles escritos por humanos. Esta evolução levanta questões importantes sobre a autenticidade da comunicação digital, a confiabilidade de conteúdos gerados automaticamente e os desafios éticos e técnicos associados à sua identificação.

No contexto académico e profissional, a distinção entre textos gerados por IA e textos humanos tornou-se uma área de investigação relevante, especialmente no campo da Aprendizagem Profunda (*Deep Learning*). A capacidade de desenvolver modelos capazes de classificar com precisão estas duas fontes de texto não só contribui para o avanço científico, mas também tem aplicações práticas, como a deteção de desinformação, a validação de autoria em documentos e a melhoria de sistemas de moderação de conteúdos.

Este trabalho insere-se nesta problemática, alinhando-se com os objetivos do primeiro módulo da UC de Aprendizagem Profunda. A crescente sofisticação dos LLMs exige o desenvolvimento de técnicas igualmente avançadas para os analisar, como o uso de embeddings para representar semanticamente o texto, *transformers* para capturar relações contextuais complexas e redes neuronais profundas para realizar classificações precisas. Além disso, o projeto reflete a necessidade de explorar tanto abordagens tradicionais de *machine learning*, como a regressão logística, quanto métodos mais modernos de *deep learning*, permitindo uma comparação abrangente entre diferentes paradigmas.

A escolha de trabalhar com textos em inglês justifica-se pela predominância desta língua nos datasets disponíveis e na literatura científica, bem como pela ampla utilização de modelos de IA treinados neste idioma. Assim, este trabalho procura contribuir para o entendimento das diferenças subtis entre textos gerados por IA e humanos, ao mesmo tempo que desenvolve competências práticas na implementação e avaliação de modelos de *machine learning* e *deep learning*.

METODOLOGIA PARA A CRIAÇÃO DE DATASETS

Para a criação dos *datasets*, o grupo seguiu dois caminhos, dependendo se o objetivo era obter texto humano ou gerado por uma tecnologia artificial.

Dados Humanos

Para obter os textos gerados por Humanos, inicialmente, o grupo criou o objetivo de realizar uma procura por *datasets* que pudessem disponibilizar informação útil para o nosso caso. Nesta, as características mais importantes passavam por, textos relativamente curtos (100 a 120 palavras) e caratriz cientifico.

Com isto, foi encontrado o conjunto de dados *arXiv Dataset* que contém os metadados de artigos académicos. O

primeiro procedimento foi descarregar este *dataset* e criar uma base de dados "MongoDB" com os mesmos dados. Utilizando a ferramenta "MongoDB Compass", com o objetivo de estudar os temas dos artigos, o grupo desenvolveu a seguinte *query* para assim analisar os temas existentes nos artigos e a sua forma:

```
db.getCollection('arXiv').aggregate(
  [
    {
      $group: {
        _id: '$categories',
        total: { $sum: 1 }
      }
    },
    { $sort: { total: -1 } }
  ]
);
```

Posteriormente, o objetivo passou para extrair do atributo *"abstract"* a informação de forma mais útil possível, prosseguindo à seguinte *pipeline*:

- Filtrar pelo tema do nosso interesse, utilizando expressões regulares. "`.*physics.+`" e "`.*bio.*i`" foram as expressões usadas para tentar captar principalmente os temas de física e biologia.
- Limitar o número de instâncias. O grupo decidiu limitar a 2500 por tema, de modo a tentar não desequilibrar o *dataset* final.
- Descartar toda a informação desnecessária, mantendo apenas o campo *"abstract"*.
- Passando à formatação do texto, primeiro foram retirados os espaços no início e fim de cada texto e depois foram substituídos "\n" por " " (caractere espaço).
- Foram filtrados os textos que tinham entre 100 a 120 palavras.
- Por fim foi adicionada uma *Label* a cada registo com o valor *"Human"*.

Dados AI

Para os textos gerados pelas *Large Language Models (LLMs)*, o grupo desenvolveu um *script* em *Python* que integra com a *API* do *Deepinfra*. Esse *script* permite ao grupo enviar um *prompt* personalizado para a *API*, que então gera textos baseados nos temas definidos. O processo é bastante flexível, pois o grupo pode alterar o conteúdo do *prompt* para explorar diferentes tópicos e contextos, o que facilitava a criação de *datasets* variados e ricos em informações. No entanto, esse processo acabou sendo repetitivo e limitado, devido a chave da *API* disponibilizada pelo *DeepInfra*, que tem um limite máximo gratuito de requisições. Isso significava que, sempre que o limite máximo de requisições era atingido, o grupo precisava criar uma nova chave de *API* para continuar o processo de geração de textos.

IMPLEMENTAÇÃO DE MODELOS DE RAIZ

No que respeita os modelos implementados de raiz, o grupo desenvolveu o código para a regressão logística, redes neuronais densas (DNN) e redes neuronais recorrentes (RNN) e, numa fase inicial, tentou implementar *support vector machine* (SVM), ideia que acabou por descartar. No restante deste capítulo, serão descritos cada modelo implementado.

Regressão Logística

A regressão logística é um modelo de *machine learning* que utiliza a técnica de gradiente descendente para atualizar os coeficientes da reta, de forma a minimizar a função de custo (*cost function*). A implementação desenvolvida pelo grupo baseou-se bastante no código desenvolvido durante as aulas.

A execução deste modelo segue a seguinte linha cronológica:

1. A função *gradient_descent* é chamada, recebendo como parâmetros os dados de treino, os de validação (opcional), o número de *epochs*/iterações e o parâmetros *alpha*, que atua como o *learning rate* do modelo
2. Em cada *epoch*
    1. Calculamos as previsões para os dados de treino
    2. Calculamos o custo (através da função de custo)
    3. Atualizamos os coeficientes da reta com base no custo obtido e no *learning rate*
    4. Para desenhar posteriormente as curvas de aprendizagem
        1. Calculamos a precisão (*accuracy*) dos dados de treino
        2. Calculamos a precisão e o custo dos dados de validação, se existirem
        3. Guardamos os valores obtidos num dicionário

Pseudo-código do gradiente descendente:

```
def gradient_descent(self, X, y, X_val, y_val,
alpha=0.01, iters=10000):
  # Theta represents the relation between inputs
and outputs
  theta = zeros(X_n_cols)

  for its in range(iters):
    # Update theta in order to better define the
relation between inputs and outputs
    delta = X.T * (sigmoid(X * (theta)) - y)
    theta = theta - (alpha / X_n_rows * delta)

    J = cost_function(X, y)
    val_loss = cost_function(X, y)

    train_acc = accuracy_score(predict(X), y)
    val_acc = accuracy_score(predict(X_val),
y_val)

    store train and validation history values to
later draw learning curves
```

No fim, caso o utilizador deseje, utilizamos os dados obtidos para desenhar as curvas de aprendizagem, de forma a obtermos uma visualização gráfica do processo de aprendizagem e percebermos qual a condição atual do modelo (*overfit*, *underfit*, entre outros). De seguida apresentamos o pseudo-código usado para desenhar as curvas.

```
def plot_train_curves(self):
  plt.figure()
  plt.plot(epochs, training_accuracy, 'r',
```

```
label='t')
  plt.plot(epochs, validation_accuracy, 'b',
label='v')
  plt.legend()
  plt.xlabel('Epoch')
  plt.ylabel('Accuracy')
  plt.title('Accuracy curves')
  plt.show()

  plt.figure()
  plt.plot(epochs, training_loss, 'r',
label='t')
  plt.plot(epochs, validation_loss, 'b',
label='v')
  plt.legend()
  plt.xlabel('Epoch')
  plt.ylabel('Loss')
  plt.title('Loss curves')
  plt.show()
```

Para obter as previsões do *dataset* de teste, o utilizador recorre à função ***predict_many***, tal como demonstrado a seguir.

```
predictions = model.predict_many(X_test)
```

DNN

As DNN, por outro lado, são modelos de *deep learning*, constituídos por múltiplas camadas, cada uma com vários neurónios e estes, por sua vez, mantém uma relação com todos os neurónios da camada posterior. Recorrem também métodos de gradiente descendente, implementado através da *forward propagation* e da *backward propagation* para aprender as relações presentes nos dados. O código implementado pelo grupo foi baseado no código fornecido pelo docente durante as aulas, sendo melhorado através de otimizações de regularização, da implementação da camanda de *Dropout* e de *early stopping* do treino.

A regularização L1 atualiza o valor do erro com a penalização dos pesos de elevados valores com base na fórmula $\left(\frac{\lambda}{2*n}\right) * \sum(|w|)$, onde λ é o termo de regularização indicado pelo utilizador, n é o número de exemplos do *input* e w é o vetor de pesos. A regularização L2 funciona da mesma forma, mudando apenas a fórmula para $\frac{\lambda}{2*n}) * \sum(w^2)$. No que respeita à camada de *Dropout*, os pesos são colocados a 0 de acordo com uma probabilidade definida pelo utilizador. Isto permite que as ligações sejam vistas como desligadas, contribuindo para evitar o *overfit* da rede. Por fim, o *early stopping* foi implementado na função *fit*, do ficheiro *network.py*, sendo que, com base no nível de paciência e da descida mínima do custo da iteração, dados fornecidos pelo utilizador, termina o treino da rede mais cedo.

Pseudo-código do cálculo do *Dropout*:

```
m = weights.shape[0]
n = weights.shape[1]

if drop_rate == 1:
  return np.zeros(m, n)

mask = np.random.rand(m, n) > self._drop_rate
dropout_weights = mask * self.weights / (1.0 -
self._drop_rate)
```

Estes *dropout_weights* são guardados na camada para serem novamente utilizados durante a fase de *back propagation*.

Pseudo-código do *early stop*:

```
if current_loss < current_minimal_loss -
min_delta:
  current_minimal_loss = current_loss
  patience_counter = 0
else:
  patience_counter += 1

  if patience_counter >= max_patience_counter:
    stop training
```

RNN

Para a implementação das Redes Neuronais Recorrentes, o grupo baseou-se no código da camada *RNN_layer* fornecido pelo docente. Este código serviu como ponto de partida para a construção do modelo, que foi posteriormente adaptado e otimizado para o problema de classificação de textos gerados por IA e humanos. As *RNN* são particularmente adequadas para este tipo de tarefa, uma vez que conseguem capturar dependências temporais e contextuais em sequências de dados, como é o caso do texto.

A implementação seguiu uma estrutura semelhante aos restantes modelos desenvolvidos de raiz, utilizando o método de gradiente descendente com *forward propagation* e *backward propagation* para ajustar os pesos da rede durante o treino. O processo pode ser descrito da seguinte forma:

- Inicialização: Os pesos e os bias da camada recorrente foram inicializados de forma aleatória, mas com valores controlados para tentar evitar problemas de explosão ou desaparecimento do gradiente, comuns em *RNN* tradicionais.
- *Forward Propagation* e *Backwards Propagation Through Time* são usadas para utilizadas para atualizar os pesos de acordo com o custo calculado em cada *epoch*.
- Otimização: Para melhorar o desempenho do modelo e evitar *overfitting*, foi implementada a técnica de *Early Stopping*.

Uma das maiores dificuldades do grupo foi a implementação da camada *Dense* final, principalmente na fase de *forward propagation*. O código final concretizado pelo grupo segue uma estrutura semelhante ao seguinte pseudo-código:

```
for t in range(timesteps):
  timestep_input = inputs.get_timestep(t)
  timestep_output = timestep_input * weights +
biases
  output_sequence.set(t) = timestep_output
```

Por fim, para conseguir calcular o erro para cada caso, o grupo recorreu a métodos estatísticos, tais como a média, para reduzir os erros de cada *timestep*.

O código foi estruturado de forma a permitir ajustes em hiperparâmetros, como o tamanho da camada oculta, o *learning rate* e o número de *epochs*. Apesar de a implementação inicial ter sido bem-sucedida, os resultados obtidos (como visíveis na seção de Resultados) indicaram que o modelo não alcançava o mesmo desempenho das *DNN* em

algumas métricas, possivelmente devido à simplicidade da arquitetura ou à ausência de técnicas mais avançadas, como *LSTM* ou *GRU*, que poderiam melhorar a capacidade de retenção de contexto em sequências mais longas.

SVM

A nível do modelo de *Support Vector Machine* (*SVM*), o grupo tentou numa fase inicial implementar para obter comparação com outros resultados obtidos. No entanto, a versão implementada pelo grupo estava muito simplista, sem recorrer a *kernels* e a parâmetros como o *gamma* e, por isso, os resultados obtidos não eram os melhores. Assim, o grupo abandonou este modelo, deixando o código desenvolvido inicialmente no repositório.

IMPLEMENTAÇÃO DE MODELOS COM TENSORFLOW

A implementação de modelos com a *framework TensorFlow* permitiu que o grupo ganhasse experiência com ferramentas utilizadas diariamente na indústria. Além do mais, oferece uma forma de comparação dos modelos desenvolvidos na primeira fase com os modelos de referência do mercado.

DNN

As DNN implementadas com *TensorFlow* recorreram a 3 tipos de camandas: *Input*, *Dense* e *Dropout*. Enquanto que as últimas duas dispensão apresentações, a camada de *Input* apareceu aqui pela primeira vez, e é responsável por definir o formato dos dados de entrada, especificar o seu tipo e fazer algum pré-processamento dos mesmos. Uma das principais diferenças sentidas em relação à nossa especificação de *DNNs* da primeira fase é a presença de uma grande variedade de funções de ativação, cada uma com as suas vantagens e desvantagens.

Topologia

Após otimizar o modelo *DNN*, a topologia final escolhida foi a seguinte:

```
n_features = X_train.shape[1]

model = models.Sequential()
model.add(Input((n_features,)))
model.add(Dense(38, activation='relu'))
model.add(Dense(27, activation='sigmoid'))
model.add(Dense(31, activation='relu'))
model.add(Dense(21, activation='sigmoid'))
model.add(Dense(1, activation='sigmoid'))

optimizer =
optimizers.Adam(learning_rate=0.0001)
model.compile(optimizer=optimizer,
loss='binary_crossentropy')

history = model.fit(X_train, y_train, epochs=20,
batch_size=24, validation_data=(X_validation,
y_validation))
```

Esta topologia foi alcançada após vários testes que englobaram o uso de *keras.tuner* e de análise dos resultados obtidos.

Um exemplo do código de procura de hiperparâmatros com o *keras.tuner* é:

```
def build_model(hp):
  min_hidden = 20
  max_hidden = 40

  n_features = X_train.shape[1]

  model = models.Sequential()
  model.add(Input((n_features,)))
  hp1_units = hp.Int('hidden1_units',
min_value=min_hidden, max_value=max_hidden,
step=1)
  model.add(layers.Dense(hp1_units,
activation='relu'))

  hp_dropout1 = hp.Float('dropout1',
min_value=0.2, max_value=0.8, step=0.1)
  model.add(layers.Dropout(rate=hp_dropout1))

  hp2_units = hp.Int('hidden2_units',
min_value=min_hidden, max_value=max_hidden,
step=1)
  model.add(layers.Dense(hp2_units,
activation='sigmoid'))

  hp_dropout2 = hp.Float('dropout2',
min_value=0.2, max_value=0.8, step=0.1)
  model.add(layers.Dropout(rate=hp_dropout2))

  hp3_units = hp.Int('hidden3_units',
min_value=min_hidden, max_value=max_hidden,
step=1)
  model.add(layers.Dense(hp3_units,
activation='relu'))
  hp4_units = hp.Int('hidden4_units',
min_value=min_hidden, max_value=max_hidden,
step=1)
  model.add(layers.Dense(hp4_units,
activation='sigmoid'))
  model.add(layers.Dense(1,
activation='sigmoid'))

  # Tune the learning rate
  learning_rate_options = [0.0001, 0.0005,
0.001, 0.005, 0.01, 0.1, 0.15, 0.2]
  momentum_options = [0.8, 0.85, 0.9, 0.95]
  hp_learning_rate = hp.Choice('learning_rate',
values=learning_rate_options)
  #hp_momentum = hp.Choice('momentum',
values=momentum_options)

  optimizer =
optimizers.Adamax(learning_rate=hp_learning_rate)
  model.compile(optimizer=optimizer,
loss='binary_crossentropy', metrics=['acc'])

  return model

# Start tuner

tuner = kt.Hyperband(
  build_model,
  objective='acc',
  max_epochs=20,
  factor=3,
  directory='./KerasTuner',  # Directory to
```

```
store results
  project_name='DNN_Tuning',
  seed=42
)

tuner.search(X_train, y_train, epochs=20,
batch_size=24, validation_data=(X_validation,
y_validation))
```

RNN

As *RNN* são redes neuronais projetadas para processar dados sequenciais, como textos, sendo capaz de capturar dependências temporais através de um mecanismo de memória interna, o que as torna adequadas para distinguir textos gerados por inteligência artificial de textos humanos.

Topologia

A topologia final escolhida para este modelo foi a seguinte:

```
model = models.Sequential()
model.add(Input(shape=(X_train.shape[1],
X_train.shape[2])))
model.add(SimpleRNN(hidden1_units,
activation='sigmoid'))
model.add(Dense(1, activation='sigmoid'))
optimizer = SGD(learning_rate=learning_rate,
momentum=momentum)
model.compile(optimizer=optimizer,
loss='binary_crossentropy')
history = model.fit(X_train, y_train,
epochs=epochs_train, batch_size=batch_size,
validation_data=(X_validation, y_validation),
callbacks=[early_stopping])
```

Esta topologia, à semelhança da topologia da ***DNN***, foi obtida através do *keras.tuner* e de uma análise de resultados. De seguida, apresentamos a criação de um modelo *RNN* com o *keras.tuner*:

```
def build_model(hp):
  min_hidden = 2
  max_hidden = 16

  model = models.Sequential()
  model.add(Input(shape=(X_train.shape[1],
X_train.shape[2])))

  hp1_units = hp.Int('hidden1_units',
min_value=min_hidden, max_value=max_hidden,
step=2)
  model.add(SimpleRNN(hp1_units,
activation='sigmoid', return_sequences=True))

  hp2_units = hp.Int('hidden2_units',
min_value=min_hidden, max_value=max_hidden,
step=2)
  model.add(SimpleRNN(hp2_units,
activation='sigmoid'))

  model.add(Dense(1, activation='sigmoid'))

  #compilar modelo
  hp_learning_rate = hp.Choice('learning_rate',
values=[0.001, 0.005, 0.01])
  hp_momentum = hp.Choice('momentum',
values=[0.001, 0.005, 0.01])
  optimizer =
SGD(learning_rate=hp_learning_rate,
momentum=hp_momentum)

  model.compile(optimizer=optimizer,
loss='binary_crossentropy', metrics=['acc'])

  return model
```

EMBEDDING

A camada de *Embedding* é uma técnica fundamental em tarefas de processamento de linguagem natural que transforma palavras ou *tokens* em representações vetoriais densas de dimensão fixa. Estas representações capturam relações semânticas entre palavras, permitindo que os modelos de *deep learning* processem textos de forma mais eficiente.

Topologia

```
dim_embed = 100
model = Sequential()
model.add(Input((X_train.shape[1],)))
model.add(Embedding(max_words, dim_embed,
embeddings_initializer=GlorotUniform(seed=44)))
model.add(Flatten())
model.add(Dense(8, activation='relu'))
model.add(Dense(16, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
loss='binary_crossentropy', metrics=['acc'])
```

Esta topologia, e a dos modelos que se seguem, ao contrário do que ocorreu com a ***DNN*** e a ***RNN***, foram obtidos através de uma sequência de tentativas de topologias, originadas através da análise dos diversos resultados obtidos.

LSTM

A *LSTM* é uma variante avançada de redes neuronais recorrentes, projetada para modelar dependências de longo prazo em sequências de dados, como textos, sendo especialmente eficaz na captura de contextos complexos para a classificação de textos gerados por inteligência artificial ou humanos.

Topologia

```
# Build model
model = models.Sequential()
model.add(Embedding(max_words, epochs))

model.add(LSTM(hidden1_units,
return_sequences=True,
kernel_initializer=GlorotUniform(seed=seed),
activation='tanh',
recurrent_activation='hard_sigmoid'))
model.add(Dropout(dropout))

model.add(LSTM(hidden2_units,
kernel_initializer=GlorotUniform(seed=seed),
activation='tanh',
recurrent_activation='hard_sigmoid'))
model.add(Dropout(dropout1))

model.add(Dense(1, activation='sigmoid'))
```

```
optimizer = SGD(learning_rate=learning_rate,
momentum=momentum)

model.compile(optimizer=optimizer,
loss='binary_crossentropy', metrics=['acc'])

model.summary()

history = model.fit(X_train, y_train,
epochs=epochs, batch_size=batch,
validation_data=(X_validation,y_validation),
callbacks=[early_stopping])
```

Os valores utilizados na topologia foram escolhidos através do *keras.tuner*. De seguida apresentamos o *keras.tuner* utilizado.

```
epochs = 10
def build_model(hp):
    min_hidden = 2
    max_hidden = 16

    hp1_units = hp.Int('hidden1_units',
min_value = min_hidden,max_value = max_hidden,
step = min_hidden)
    hp2_units = hp.Int('hidden2_units',
min_value = min_hidden,max_value = max_hidden,
step = min_hidden)
    hp_learning_rate =
hp.Choice('learning_rate', values = [0.1, 0.01,
0.001])
    hp_momentum = hp.Choice('momentum', values =
[0.5, 0.9, 0.95])
    hp_batch = hp.Choice('batch', values = [4,
8, 16])
    hp_dropout = hp.Choice('dropout', values =
[0.2, 0.3, 0.4])
    hp_dropout1 = hp.Choice('dropout1', values =
[0.2, 0.3, 0.4])

    model = models.Sequential()
    model.add(Embedding(max_words, epochs))

    model.add(LSTM(hp1_units,
return_sequences=True,
kernel_initializer=GlorotUniform(seed=seed),
activation='tanh',
recurrent_activation='hard_sigmoid'))
    model.add(Dropout(hp_dropout))

    model.add(LSTM(hp2_units,
kernel_initializer=GlorotUniform(seed=seed),
activation='tanh',
recurrent_activation='hard_sigmoid'))
    model.add(Dropout(hp_dropout1))

    model.add(Dense(1, activation='sigmoid'))

    optimizer =
SGD(learning_rate=hp_learning_rate,
momentum=hp_momentum)

    model.compile(optimizer=optimizer,
loss='binary_crossentropy', metrics=['acc'])

    return model
```

GRU

A *GRU (Gated Recurrent Unit)* é um tipo de rede neural recorrente *(RNN)* projetada para processar dados sequenciais, como textos, séries temporais e áudio. Neste contexto, faz sentido o uso dela para a classificação dos textos, para detetar foram gerados por inteligência artificial ou escritos por humanos.

Topologia

```
embeddings_initializer=initializers.GlorotUniform(seed=666)

dim_embed = 20

model = models.Sequential()
model.add(Embedding(max_words, dim_embed))
model.add(Dropout(0.1, seed=666))
model.add(GRU(dim_embed, activation='tanh',
kernel_initializer=GlorotUniform(seed=48)))
model.add(Dropout(0.2, seed=666))
model.add(Dense(1, activation='sigmoid'))

optimizer =
optimizers.RMSprop(learning_rate=0.005)
model.compile(optimizer='rmsprop',
loss='binary_crossentropy', metrics=['acc'])
```

O grupo decidiu incorporar a camada de *dropout*, com o intuito de mitigar o risco de *overfit*, um problema comum em redes neurais profundas, especialmente quando os dados de treino são limitados ou ruidosos. A escolha da função de ativação 'tanh' na camada GRU foi feita para garantir uma boa propagação dos gradientes durante o treino. Além disso, o otimizador 'rmsprop' foi escolhido pela sua capacidade de adaptar a taxa de aprendizagem ao longo do treino, o que pode acelerar a convergência em problemas com dados sequenciais.

Transformer

O *transformer* é uma arquitetura de *deep learning* que utiliza *self-attetion* em vez de estruturas recorrentes permitindo assim ao modelo ponderar a importância da sequencia das diferentes palavras e capturar dependências de longo alcance em paralelo.

Topologia

```
x = PositionalEmbedding(max_len, max_words,
embed_dim)(inputs)

for _ in range(1):
    x = TransformerEncoder(
        embed_dim,
        dense_dim,
        num_heads,
    )(x)

x = layers.GlobalMaxPooling1D()(x)
x = layers.Dropout(0.8)(x)

model.compile(optimizer=AdamW(learning_rate=0.001,
weight_decay=0.01), loss="binary_crossentropy",
metrics=["acc"])
```

Foi utilizado tanto *dropout* como *weight_decay* pois como temos poucos dados de treino o modelo está muito propenso a *overfit*.

O grupo optou por integrar tanto o *dropout* com uma taxa elevada de 0.8 como o *weight_decay* no otimizador AdamW para reforçar a regularização do modelo e minimizar o risco de *overfitting*, especialmente dado o tamanho reduzido do conjunto de dados de treino utilizado.

O *TransformerEncoder*, com múltiplas cabeças de atenção (num_heads), permite capturar relações complexas entre as palavras, enquanto a camada *GlobalMaxPooling1D* reduz a dimensionalidade da saída para uma representação fixa tornando possível a classificação. O otimizador AdamW foi escolhido por combinar adaptação eficiente dos gradientes com penalização dos pesos, para combater o *overfit*.

PREVISÃO ATRAVÉS DE LLM

O grupo, nesta fase, utilizou duas *APIs* distintas, cada uma oferecendo modelos diferentes. A escolha recaiu sobre essas duas opções, pois ambas permitem o uso gratuito de pelo menos 1 modelo.

- A primeira *API* utilizada foi a da *Deepinfra*. Conforme mencionado anteriormente, ela exige a geração de uma nova chave sempre que o limite é atingido. Nesta *API*, foram testadas as configurações *zero-shot*, *one-shot* e até com 6 exemplos, e os resultados apresentaram uma variação aleatória, situando-se sempre entre 50% e 62%.
- A segunda *API* utilizada foi a disponibilizada pelo *Together AI*. Esta *API* permite o uso "ilimitado" do modelo *LLama-3.3-70B*, com a pequena limitação de 6 requisições por minuto. Nessa plataforma, foram testadas as configurações *zero-shot*, *one-shot* e até com 6 exemplos, e os resultados obtidos apresentaram uma variação aleatória, situando-se consistentemente entre 78% e 84%.

Esses resultados evidenciam a robustez do segundo modelo, que demonstrou uma performance superior em relação à primeira *API* utilizada. A partir do uso desses dois modelos, o grupo concluiu que o fator que mais influencia a performance de previsão é o próprio modelo.

RESULTADOS

Neste tópico são apresentados os resultados obtidos a cada submissão. A coluna "resultados locais " corresponde aos exemplos disponibilizados pelo professor para cada submissão: a primeira submissão contém 30 exemplos, a segunda 80 exemplos e a terceira 130 exemplos. Por outro lado, a coluna "Resultados do Professor" apresenta os resultados obtidos em cada submissão realizada pelo professor.

Primeira Submissão

Na primeira submissão, o grupo utilizou exclusivamente os modelos que implementou recorrendo somente à biblioteca *numpy*. nesta fase, os seguintes resultados foram alcançados:

Tabela 1: Resultados Primeira Submissão

Modelo	**Resultados locais**	**Resultados Professor**
DNN	73.(3)%	80.0%
RNN	70.0%	63.0%
Reg. Logistica	70.0%	-

Segunda Submissão

Na segunda submissão, foram empregados exclusivamente os modelos desenvolvidos com o *framework TensorFlow*. Durante essa fase, o grupo obteve os seguintes resultados:

Tabela 2: Resultados Segunda Submissão

Modelo	**Resultados locais**	**Resultados**	**Professor**
DNN	82.0%		78.0%
Embedding	72.0%		78.0%
RNN	68.0%		–
GRU	72.0%		-

Terceira Submissão

Na terceira fase, o grupo continuou utilizando os modelos implementados com *TensorFlow*, mas passou a incorporar o uso de *LLMs* disponibilizadas por *APIs*. Durante essa etapa, os seguintes resultados foram obtidos:

Tabela 3: Resultados Terceira Submissão

Modelo	**Resultados locais**	**Resultados Professor**
LLM - LLama	82.0% (best one)	80.0%
Transformers	80.0%	69.0%
Gru	79.2%	–
LSTM	75.3%	–
LLM - Mistril	$\approxeq$ 57.5%	–

CONCLUSÕES

Este projeto foi fundamental para consolidar as competências práticas dos diversos membros do grupo no que respeita a construção de *datasets* e a implementação de modelos de *machine* e *deep learning*. O trabalho permitiu também perceber como é que a teoria dos diversos modelos é implementada em prática, visto a primeira fase requirir modelos implementados pelo grupo recorrendo somente a *numpy*. Para além do mais, o grupo ganhou experiência a utilizar ferramentas presentes no dia a dia da indústria, tais como o *TensorFlow*.

Durante todo o trabalho, foi evidente a complexidade de distinguir entre textos gerados por inteligência artificial e

textos escritos por humanos. Com recurso a algumas técnicas, tais como *Bag of words* ou *Embedding*, e a diferentes redes neuronais e topologias, podemos observar como é que cada uma destas variáveis se comportam nesta tarefa, desde a sua performance até à sua capacidade de capturar mais ou menos detalhes nos dados.

Por fim, este projeto realçou a importância da adaptação do *dataset* ao cenário de utilização do modelo, garantindo que os dados são adequados mas não são tão idênticos ao ponto do modelo se adaptar demasiado aos dados. Nesta questão de *overfit*, além do tratamento do *dataset*, o trabalho prático forneceu a oportunidade do grupo implementar técnicas que mitigam este risco, tais como regularização L1 e L2, *Dropout* ou *Early Stopping*, técnicas estas usadas tanto nos modelos implementados pelo grupo como nos modelos implementados recorrendo ao *TensorFlow*.