ALÉM DAS FRONTEIRAS DIGITAIS:
CHIPS NEUROMÓRFICOS

LINGUAGEM SOB CONTROLE: MODELOS PROFUNDOS NO NLP

10

LISTA DE FIGURAS

LISTA DE CÓDIGOS-FONTE

LISTA DE COMANDOS DE PROMPT DO SISTEMA OPERACIONAL

SUMÁRIO

1 LINGUAGEM SOB CONTROLE: MODELOS PROFUNDOS NO NLP

1.1 O que é o Aprendizado Profundo?

O Aprendizado Profundo (ou Deep Learning, em inglês) é uma subárea do aprendizado de máquina que utiliza Redes Neurais Artificiais (RNAs) com múltiplas camadas (Deep Neural Networks) para modelar e extrair representações complexas de dados. Essas redes são compostas por camadas hierárquicas de neurônios artificiais, capazes de aprender automaticamente características de alto nível a partir de dados brutos, como imagens, texto ou áudio, sem a necessidade de engenharia manual de atributos. O treinamento é realizado por algoritmos de otimização, como o gradiente descendente, que ajustam os pesos das conexões neurais com base no erro da previsão, frequentemente utilizando grandes volumes de dados e alto poder computacional.

Uma das principais vantagens do aprendizado profundo em relação aos métodos tradicionais de aprendizado de máquina, conhecidos como aprendizado raso, é a sua habilidade de lidar com grandes volumes de dados e aprender representações complexas de forma direta, sem a necessidade de pré-processamento extensivo (engenharia de features). Isso torna o Deep Learning extremamente eficaz em tarefas como reconhecimento de imagens, Processamento de Linguagem Natural (NLP) e reconhecimento de fala, onde padrões não lineares precisam ser identificados.

Entretanto, essa capacidade de aprender a partir de dados brutos também traz desafios. Modelos de aprendizado profundo exigem grandes quantidades de dados para alcançar uma boa performance, o que pode ser um obstáculo em cenários com dados limitados. Além disso, embora esses modelos apresentem alta acurácia na tarefa específica para a qual foram treinados, sua capacidade de generalização é limitada. Ou seja, um modelo treinado para uma tarefa, como o reconhecimento de rostos, pode não se sair bem em outras tarefas sem ajustes significativos.

Nota: o Aprendizado Profundo e o Aprendizado Raso se diferenciam pela forma como lidam com a Engenharia de Features no pré-processamento. No Aprendizado Raso, é necessário realizar essa etapa manualmente, enquanto no Aprendizado Profundo, o processo é feito automaticamente pelo modelo.

1.2 Arquiteturas de RNAs

Perceptrons são um dos modelos mais simples de redes neurais artificiais, compostos por uma única camada de neurônios. Cada neurônio recebe entradas, que são multiplicadas por pesos, e, em seguida, passa por uma função de ativação para produzir uma saída. O objetivo dos perceptrons é ajustar esses pesos de maneira que a saída da rede seja o mais próxima possível do valor desejado, minimizando o erro de previsão. O treinamento de um perceptron é realizado por meio do algoritmo de retropropagação, que ajusta os pesos com base no erro entre a saída prevista e a saída esperada. Esse ajuste é feito utilizando um algoritmo de otimização, como o gradiente descendente, para minimizar a função de custo, o que permite à rede aprender a mapear as entradas para as saídas corretas, mesmo em problemas não lineares.

Multilayer feedforward networks (ou perceptrons multicamadas, do inglês Multlayer Perceptrons, MLP) são modelos de redes neurais compostos por várias camadas de neurônios organizados de forma hierárquica. Cada neurônio em uma camada recebe entradas, que são multiplicadas por pesos e, em seguida, passa por uma função de ativação para gerar uma saída. O principal objetivo dessas redes é ajustar os pesos das conexões de modo que a saída da rede seja o mais próxima possível do valor desejado, minimizando o erro de previsão. O treinamento de uma multilayer feedforward network é realizado por meio do algoritmo de retropropagação, que ajusta os pesos com base no erro entre a saída prevista e a saída esperada.

Além dos perceptrons e das MLP, existem diversas outras arquiteturas de redes neurais que são projetadas para atender a diferentes tipos de problemas. Uma dessas arquiteturas é a **Rede Neural Convolucional** (do inglês, Convolutional Neural Networks, CNN), amplamente utilizada em problemas de processamento de imagem e vídeo, reconhecimento de padrões e tarefas relacionadas. As CNNs são caracterizadas por camadas convolucionais que aplicam filtros (ou kernels) para extrair características locais das imagens. Essas redes são muito eficazes pois, ao contrário das redes tradicionais, elas são capazes de detectar padrões em regiões específicas da entrada sem a necessidade de conectar todos os neurônios da camada anterior a todos os neurônios da camada seguinte, como ocorre nas MLPs. Além disso, as CNNs costumam ter camadas de pooling, que reduzem a dimensionalidade dos dados, permitindo uma maior generalização e eficiência no treinamento.

Outro tipo importante de rede neural é a **Rede Neural Recorrente** (do inglês, Recurrent Neural Network, RNN). As RNNs são projetadas para trabalhar com dados sequenciais ou temporais, como séries temporais, texto e áudio. O diferencial das RNNs é que elas possuem conexões recorrentes, o que permite que a saída de um neurônio de uma camada seja retroalimentada como entrada para o mesmo neurônio em um momento posterior, criando assim uma “memória” que pode influenciar as saídas futuras. Isso torna as RNNs ideais para tarefas como previsão de séries temporais, tradução automática e modelagem de linguagem, onde o contexto passado influencia diretamente a previsão futura. No entanto, as RNNs tradicionais têm dificuldades em aprender dependências de longo prazo devido ao problema de desvanecimento e explosão do gradiente. Para contornar isso, foram desenvolvidas variações como as **LSTMs** (Long Short-Term Memory) e as **GRUs** (Gated Recurrent Units), que são capazes de aprender e reter dependências de longo prazo de maneira mais eficiente.

Além disso, temos as **Redes Neurais de Atenção**, que se tornaram extremamente populares especialmente em modelos de processamento de linguagem natural. Arquiteturas como o **Transformer**, baseadas em mecanismos de atenção, permitem que a rede foque em diferentes partes de uma entrada simultaneamente, sem a necessidade de processar os dados de forma sequencial, como nas RNNs. Esse tipo de rede é altamente paralelo, o que facilita o treinamento em grandes volumes de dados. Modelos baseados em Transformer, como o **BERT** e o **GPT**, têm mostrado resultados impressionantes em tarefas como tradução de idiomas, geração de texto e compreensão de contexto, devido à sua capacidade de capturar relações complexas entre palavras em uma sequência.

Ainda podemos mencionar as **Redes Neurais Generativas Adversariais** (do inglês Generative Adversarial Networks, GANs), que consistem em duas redes neurais treinadas simultaneamente: um **gerador** e um **discriminador**. O gerador tenta criar amostras realistas, enquanto o discriminador tenta distinguir entre amostras reais e geradas. Esse processo de *adversarial training* faz com que as GANs sejam particularmente poderosas em tarefas de geração de dados, como a criação de imagens, vídeos, músicas e até mesmo deepfakes. Elas também têm sido aplicadas em áreas como design de novos medicamentos, melhoria de resolução de imagens e geração de arte.

Por fim, as **Redes Neurais de Aprendizagem por Reforço** (do inglês Reinforcement Learning, RL) são projetadas para aprender políticas de ação em ambientes dinâmicos. Nesse tipo de rede, um agente interage com um ambiente e aprende a maximizar recompensas através de tentativas e erros, ajustando suas ações com base no feedback recebido. As redes de aprendizagem por reforço têm sido fundamentais para o desenvolvimento de sistemas de IA que conseguem resolver problemas complexos, como jogos (por exemplo, o AlphaGo da DeepMind) e até mesmo controle de robôs em ambientes do mundo real.

Cada uma dessas arquiteturas de redes neurais tem características e vantagens distintas, sendo escolhidas de acordo com o tipo de problema que se deseja resolver. Neste capítulo, focaremos nas arquiteturas que melhor se adequam para o NLP, como as LSTMs e os Transformers. No entanto, antes de prosseguirmos, é importante que estejamos familiarizados com as bibliotecas de aprendizado profundo que iremos utilizar. Por isso, na próxima seção, discutimos como treinar RNAs em Keras/TensorFlow e PyTorch para dados tabulares.

1.3 Treinando uma RNA

O uso de bibliotecas como **scikit-learn**, **Keras/TensorFlow** e **PyTorch** oferece diferentes abordagens e níveis de flexibilidade para treinar Redes Neurais Artificiais. O scikit-learn é uma excelente opção para quem está começando, pois oferece implementações simples e de alto nível de redes neurais e outros algoritmos de aprendizado de máquina. Embora menos flexível que as outras bibliotecas, ele é ideal para tarefas mais simples, como classificação e regressão.

O Keras — que agora faz parte do TensorFlow — proporciona uma interface de alto nível e fácil de usar para a construção e treinamento de redes neurais profundas, com abstrações poderosas para definir, treinar e avaliar modelos de redes neurais. Já o PyTorch, conhecido por sua flexibilidade e controle detalhado sobre o processo de treinamento, é amplamente utilizado em pesquisa e produção, permitindo a criação de redes neurais altamente customizáveis e mais eficientes em termos de desempenho, principalmente para redes profundas e complexas. Enquanto o TensorFlow/Keras foca em uma abstração maior e facilidade de uso, o PyTorch oferece maior controle e é preferido por muitos pesquisadores devido à sua dinâmica de execução, que facilita a experimentação.

Antes de realizarmos um exemplo prático, precisamos nos certificar de instalar corretamente as bibliotecas:

```
pip install tensorflow
```

Comando de prompt 1 - Instalação do TensorFlow
Fonte: Elaborado pelo autor (2025)

O código-fonte “Uso do TensorFlow para criação de uma rede neural” cria e treina uma rede neural utilizando Keras/TensorFlow para classificar o conjunto de dados Iris, que contém características de flores e suas respectivas espécies. Ele começa carregando e normalizando os dados de entrada, aplicando a técnica de one-hot encoding para as classes de saída. Em seguida, constrói um modelo de rede neural sequencial com duas camadas ocultas, utilizando a função de ativação ReLU, e uma camada de saída com softmax para gerar probabilidades das classes. O modelo é compilado com a função de perda `categorical_crossentropy` e o otimizador Adam, treinado por 100 épocas e, por fim, avaliado para verificar sua acurácia no conjunto de teste.

```
import tensorflow as tf
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.utils import to_categorical

# Carregando o conjunto de dados Iris
data = load_iris()
X = data.data
y = data.target

# Normalizar os dados de entrada
scaler = StandardScaler()
X = scaler.fit_transform(X)

# Converter as classes em variáveis categóricas
y = to_categorical(y, num_classes=3)

# Dividir em treino e teste
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)

# Criar o modelo de rede neural feedfoward
model = Sequential()

# Camada de entrada (com 4 neurônios para 4 features de entrada)
model.add(Dense(8, input_dim=4, activation='relu'))

# Camada oculta
model.add(Dense(8, activation='relu'))

# Camada de saída (com 3 neurônios para 3 classes de saída)
model.add(Dense(3, activation='softmax'))

# Compilar o modelo
model.compile(loss='categorical_crossentropy',
optimizer='adam', metrics=['accuracy'])

# Treinar o modelo
model.fit(X_train, y_train, epochs=100, batch_size=10,
validation_data=(X_test, y_test))

# Avaliar o modelo
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Loss: {loss}")
print(f"Accuracy: {accuracy}")
```

Código-fonte 1 - Uso do TensorFlow para criação de uma rede neural
Fonte: Elaborado pelo autor (2025)

No código-fonte “Uso do PyTorch para criação de uma rede neural”, veremos como implementar o mesmo modelo de rede neural utilizado no TensorFlow, mas agora com o PyTorch. O objetivo é classificar o conjunto de dados Iris, que contém características de flores e suas respectivas espécies. Assim como no exemplo com TensorFlow, é necessário carregar e normalizar os dados, convertendo as classes em formato one-hot para adaptá-las à rede neural. Em seguida, construímos uma rede com uma camada de entrada, uma camada oculta e uma camada de saída, utilizando as funções de ativação ReLU e softmax, mas, no caso do PyTorch, a função de perda `CrossEntropyLoss` já realiza o softmax internamente, simplificando o modelo. Durante o treinamento, a otimização é realizada com o otimizador Adam, e a rede é treinada por 100 épocas para classificar as amostras corretamente. A avaliação do modelo também é realizada, comparando as previsões da rede com os valores reais das classes no conjunto de teste e, por fim, calculando a precisão da rede neural. Assim, o código demonstra a flexibilidade e as semelhanças entre as duas bibliotecas, destacando a estrutura de treino e validação no PyTorch para este tipo de tarefa de classificação.

Assim como fizemos para o TensorFlow, para instalar o PyTorch, você pode executar o seguinte comando utilizando o gerenciador de pacotes de Python:

```
pip install torch
```

Comando de prompt 2 - Instalação do PyTorch
Fonte: Elaborado pelo autor (2025)

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import OneHotEncoder
from torch.utils.data import DataLoader, TensorDataset

# Carregar o conjunto de dados Iris
data = load_iris()
X = data.data
y = data.target

# Normalizar os dados de entrada
scaler = StandardScaler()
X = scaler.fit_transform(X)

# One-hot encoding para as classes
encoder = OneHotEncoder(sparse_output=False)
y = encoder.fit_transform(y.reshape(-1, 1))

# Dividir em treino e teste
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)

# Converter para tensores do PyTorch
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.float32)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.float32)

# Criar Dataset e DataLoader para treino e teste
train_data = TensorDataset(X_train_tensor, y_train_tensor)
test_data = TensorDataset(X_test_tensor, y_test_tensor)

train_loader = DataLoader(train_data, batch_size=10,
shuffle=True)
test_loader = DataLoader(test_data, batch_size=10,
shuffle=False)

# Definir o modelo de rede neural
class IrisNN(nn.Module):
    def __init__(self):
        super(IrisNN, self).__init__()
        self.fc1 = nn.Linear(4, 8)  # Camada de entrada (4
features de entrada para 8 neurônios)
        self.fc2 = nn.Linear(8, 8)  # Camada oculta
        self.fc3 = nn.Linear(8, 3)  # Camada de saída (3
classes)
```

```
    def forward(self, x):
        x = torch.relu(self.fc1(x))  # Função de ativação ReLU
        x = torch.relu(self.fc2(x))  # Função de ativação ReLU
        x = self.fc3(x)  # Saída sem ativação aqui, pois usaremos o softmax na loss function
        return x

# Instanciar o modelo, definir a função de perda e o otimizador
model = IrisNN()
criterion = nn.CrossEntropyLoss()  # Usamos CrossEntropyLoss, que aplica o softmax internamente
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Treinamento do modelo
epochs = 100
for epoch in range(epochs):
    model.train()
    running_loss = 0.0
    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, torch.max(labels, 1)[1])
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    # A cada 10 épocas, exibe o progresso
    if (epoch + 1) % 10 == 0:
        print(f'Epoch {epoch + 1}/{epochs}, Loss: {running_loss/len(train_loader)}')

# Avaliar o modelo
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for inputs, labels in test_loader:
        outputs = model(inputs)
        _, predicted = torch.max(outputs, 1)
        _, labels_max = torch.max(labels, 1)
        total += labels.size(0)
        correct += (predicted == labels_max).sum().item()

accuracy = correct / total
print(f'Accuracy on test data: {accuracy * 100:.2f}%')
```

Código-fonte 2 - Uso do PyTorch para criação de uma rede neural
Fonte: Elaborado pelo autor (2025)

2 REDES NEURAIS RECORRENTES

2.1 O que é uma RNN?

O uso de Redes Neurais Recorrentes em NLP tem sido uma abordagem fundamental para tarefas que envolvem dados sequenciais como tradução automática, geração de texto, análise de sentimentos, entre outras. As RNNs são capazes de manter informações de contextos anteriores dentro de sua estrutura de rede, o que as torna ideais para lidar com sequências de texto ou fala.

Uma RNN é composta por células de memória que retem um estado ao longo do tempo, permitindo que a saída em um momento t dependa não apenas da entrada x_t naquele instante, mas também do estado anterior h_{t-1}. Isso permite à RNN "lembrar" informações de passos anteriores na sequência.

A figura ilustra um exemplo esquemático de RNN:

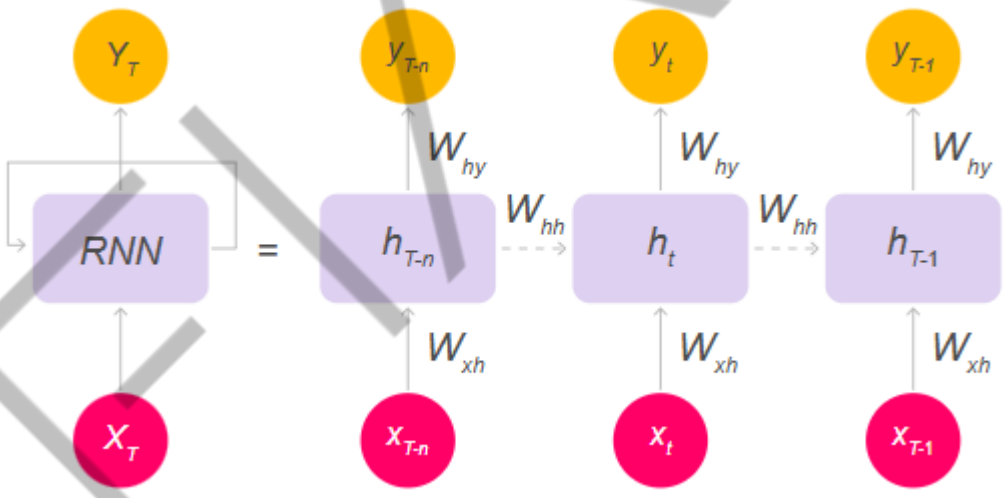

Figura 1 - Esquema de arquitetura de uma RNN
Fonte: Google imagens (2025), adaptado por FIAP (2025)

Matematicamente, a atualização de uma célula de memória pode ser descrita da seguinte forma:

$$h_t = f(W_h h_{t-1} + W_x x_t + b)$$

Onde h_t é o estado oculto no tempo t; h_{t-1} é o estado oculto no tempo anterior; x_t é a entrada no tempo t; W_h e W_x são os pesos sinápticos das conexões entre os estados ocultados e as entradas, respectivamente; f é uma função de ativação, geralmente uma função sigmoide ou tangente hiperbólica; e b é o viés.

A saída y_t da rede é descrita pelos pesos sinápticos W_y que mapeiam o estado oculto para a saída:

$$y_t = W_y h_t + b_y$$

2.2 Criando uma RNN para processar uma série temporal

A seguir, vamos entender como construir uma RNN. O código-fonte "Criando uma RNN para processar sinais usando o PyTorch" utiliza uma RNN para prever o valor futuro de um sinal AM (modulado em amplitude) ruidoso. Inicialmente, o código gera um sinal portador de 50 Hz, que é modulado por um sinal de 5 Hz com uma profundidade de modulação de 0,5. O sinal modulado é então somado a um ruído gaussiano, criando o sinal ruidoso que será utilizado nas etapas seguintes. Para alimentar a rede neural, o código utiliza uma abordagem baseada em janelas de amostras, onde são selecionadas sequências de 10 amostras consecutivas do sinal ruidoso como entrada (X), e a amostra subsequente é utilizada como saída (Y). As janelas de entrada e saída são transformadas em tensores do PyTorch. O dataset é dividido em 80% para treino e 20% para teste. Essa estrutura é típica em problemas de previsão de séries temporais, onde a rede aprende a prever o próximo valor a partir dos valores anteriores.

A arquitetura da rede neural é simples, composta por uma camada RNN que processa as sequências temporais. O número de unidades na camada oculta é 5, e a camada de saída é uma camada totalmente conectada (densa), que mapeia a saída da RNN para a previsão do próximo valor do sinal. O modelo é treinado utilizando o algoritmo de otimização Adam e a função de perda escolhida é o erro quadrático médio (MSE). O treinamento ocorre por 100 épocas, e a cada 10 épocas, a perda atual é exibida para monitorar o progresso.

```
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import numpy as np

#-------------------- Dados ----------------
fs = 1000  # Frequência de amostragem (Hz)
t = np.linspace(0, 1, fs)  # Intervalo de tempo de 1 segundo
f_carrier = 50  # Frequência da portadora (Hz)
f_mod = 5  # Frequência da modulação (Hz)
depth = 0.5  # Profundidade de modulação
carrier = np.cos(2 * np.pi * f_carrier * t)
modulating = 1 + depth * np.sin(2 * np.pi * f_mod * t)
am_signal = modulating * carrier
noise = np.random.normal(0, 0.2, t.shape)
am_signal_noisy = am_signal + noise

# Usando janelas de 10 amostras como entrada (X) e a próxima
amostra como saída (Y)
window_size = 10
X = []
y = []

for i in range(len(am_signal_noisy) - window_size):
    X.append(am_signal_noisy[i:i+window_size])
    y.append(am_signal_noisy[i+window_size])

# Convertendo para tensores do PyTorch
X  =  torch.tensor(X,  dtype=torch.float32).unsqueeze(-1)    #
Shape (batch_size, seq_len, input_size)
y  =  torch.tensor(y,  dtype=torch.float32).unsqueeze(-1)    #
Shape (batch_size, output_size)

# Dividindo os dados em treino e teste
train_size = int(len(am_signal_noisy) * 0.8)
X_train = X[:train_size]
X_test = X[train_size:]
y_train = y[:train_size]
y_test = y[train_size:]

#-------------------- RNN ----------------
class SimpleRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SimpleRNN, self).__init__()
        self.hidden_size = hidden_size
        self.rnn     =     nn.RNN(input_size,     hidden_size,
batch_first=True) # Camada RNN
        self.fc = nn.Linear(hidden_size, output_size) # Camada
de saída
```

```
    def forward(self, x):
        # Inicializando o estado oculto (h0) com zeros
        h0 = torch.zeros(1, x.size(0),
self.hidden_size).to(x.device)
        # Passar os dados pela RNN
        out, hn = self.rnn(x, h0)
        # Passar o resultado da RNN pela camada de saída
        out = self.fc(out[:, -1, :])
        return out

# Parâmetros do modelo
input_size = 1  # Entrada unidimensional
hidden_size = 5  # Tamanho da camada oculta
output_size = 1  # Saída unidimensional

# Inicializar o modelo, critério e otimizador
model = SimpleRNN(input_size, hidden_size, output_size)
criterion = nn.MSELoss()  # Erro quadrático médio
optimizer = optim.Adam(model.parameters(), lr=0.01)

# Treinamento do modelo
epochs = 100
for epoch in range(epochs):
    # Zerar os gradientes dos parâmetros
    optimizer.zero_grad()
    # Passar os dados de treino pela rede
    outputs = model(X_train)
    # Calcular a perda
    loss = criterion(outputs, y_train)
    # Backpropagation e otimização
    loss.backward()
    optimizer.step()
    # Exibir o progresso
    if (epoch + 1) % 10 == 0:
        print(f"Epoch [{epoch+1}/{epochs}], Loss:
{loss.item():.4f}")

#------------------- TESTE ----------------
# Predição com o modelo treinado (usando X_test)
with torch.no_grad():
    y_pred = model(X_test)

# Plotando o gráfico de comparação em subplots
fig, axs = plt.subplots(1, 2, figsize=(15, 6))

axs[0].plot(t[:train_size], am_signal_noisy[:train_size],
color='blue', label='Treinamento')
axs[0].plot(t[train_size:], am_signal_noisy[train_size:],
color='orange', label='Teste')
axs[0].set_title('Sinal Original (Treinamento e Teste)')
axs[0].set_xlabel('Tempo (s)')
```

```
axs[0].set_ylabel('Amplitude')
axs[0].grid(True)
axs[0].legend()

axs[1].plot(t[train_size + window_size:],
y_test.numpy().flatten(), label="Valores Reais (Teste)",
color='orange', alpha=0.7)
axs[1].plot(t[train_size + window_size:],
y_pred.numpy().flatten(), label="Predições da RNN",
color='green', linestyle='--')
axs[1].set_title('Predições vs Valores Reais (Teste)')
axs[1].set_xlabel('Tempo (s)')
axs[1].set_ylabel('Amplitude')
axs[1].grid(True)
axs[1].legend()

plt.tight_layout()
plt.show()
```

Código-fonte 3 - Criando uma RNN para processar sinais usando o PyTorch
Fonte: Elaborado pelo autor (2025)

Uma vez treinado, o modelo é avaliado no conjunto de teste, onde realiza a previsão do sinal a partir das sequências de entrada fornecidas. Para visualizar o desempenho da rede, são gerados dois gráficos (ilustrados na figura “Saída do Código-fonte 3, processando série temporal com RNN”). O primeiro gráfico exibe o sinal original (com ruído), mostrando as partes de treino (azul) e teste (amarelo). O segundo gráfico compara as previsões feitas pela RNN com os valores reais do sinal no conjunto de teste.

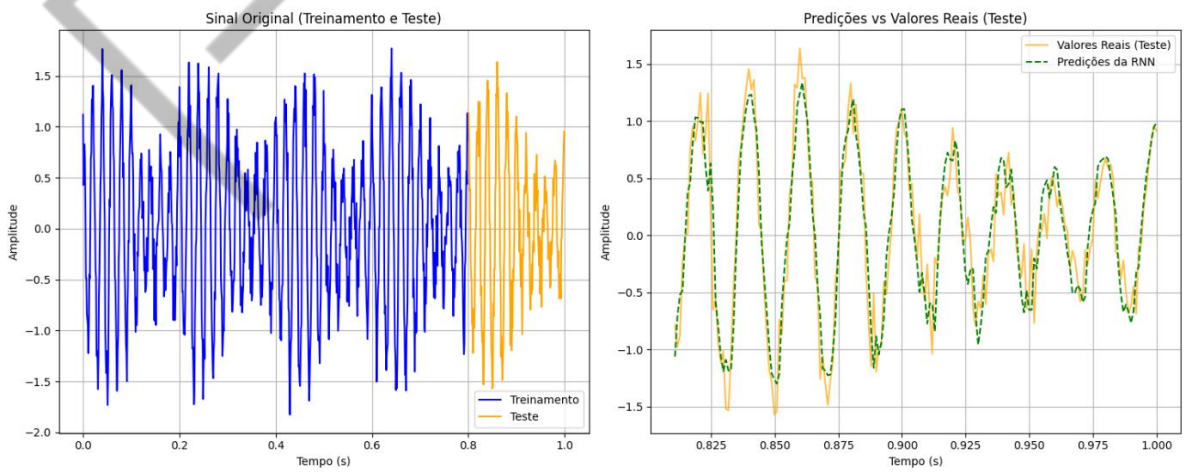

Figura 2 - Saída do Código-fonte 3, processando série temporal com RNN
Fonte: Elaborado pelo autor (2025)

Podemos visualizar os pesos sinápticos da RNN construída no código-fonte “Criando uma RNN para processar sinais usando o PyTorch” através do código-fonte

a seguir. Nele, plotamos a figura “Saída do Código-fonte 4, pesos sinápticos da RNN do Código-fonte 2”: a camada de entrada tem 5 pesos aprendidos; a camada oculta tem 25 pesos que correspondem a memória da RNN; a camada de saída tem mais 5 pesos aprendidos. As entradas são multiplicadas pelos pesos, conforme discutido na seção anterior.

```
import matplotlib.pyplot as plt
import numpy as np

fig, axs = plt.subplots(1, 3, figsize=(18, 5))

cax0 = axs[0].imshow(model.rnn.weight_ih_l0.detach().numpy(),
aspect='auto', cmap='viridis')
axs[0].set_title('Pesos de Entrada ($W_x$)')
axs[0].set_xlabel('Neurônios de Saída')
axs[0].set_ylabel('Neurônios de Entrada')
fig.colorbar(cax0, ax=axs[0])
axs[0].set_xticks(np.arange(0,
model.rnn.weight_ih_l0.shape[1], 1))
axs[0].set_yticks(np.arange(0,
model.rnn.weight_ih_l0.shape[0], 1))
axs[0].set_xticklabels(np.arange(0,
model.rnn.weight_ih_l0.shape[1], 1))
axs[0].set_yticklabels(np.arange(0,
model.rnn.weight_ih_l0.shape[0], 1))
for i in range(model.rnn.weight_ih_l0.shape[0]):
    for j in range(model.rnn.weight_ih_l0.shape[1]):
        axs[0].text(j,    i,    f'{model.rnn.weight_ih_l0[i,
j].item():.2f}',  ha='center',  va='center',  color='white',
fontsize=10)

cax1 = axs[1].imshow(model.rnn.weight_hh_l0.detach().numpy(),
aspect='auto', cmap='viridis')
axs[1].set_title('Pesos Recorrentes ($W_h$)')
axs[1].set_xlabel('Neurônios de Saída')
axs[1].set_ylabel('Neurônios Ocultos Anteriores')
fig.colorbar(cax1, ax=axs[1])
axs[1].set_xticks(np.arange(0,
model.rnn.weight_hh_l0.shape[1], 1))
axs[1].set_yticks(np.arange(0,
model.rnn.weight_hh_l0.shape[0], 1))
axs[1].set_xticklabels(np.arange(0,
model.rnn.weight_hh_l0.shape[1], 1))
axs[1].set_yticklabels(np.arange(0,
model.rnn.weight_hh_l0.shape[0], 1))
for i in range(model.rnn.weight_hh_l0.shape[0]):
    for j in range(model.rnn.weight_hh_l0.shape[1]):
```

```
        axs[1].text(j,     i,     f'{model.rnn.weight_hh_l0[i,
j].item():.2f}',   ha='center',   va='center',   color='white',
fontsize=10)

cax2     =     axs[2].imshow(model.fc.weight.detach().numpy(),
aspect='auto', cmap='viridis')
axs[2].set_title('Pesos de Saída ($W_y$)')
axs[2].set_xlabel('Neurônios de Saída')
axs[2].set_ylabel('Neurônios Ocultos')
fig.colorbar(cax2, ax=axs[2])
axs[2].set_xticks(np.arange(0, model.fc.weight.shape[1], 1))
axs[2].set_yticks(np.arange(0, model.fc.weight.shape[0], 1))
axs[2].set_xticklabels(np.arange(0,  model.fc.weight.shape[1],
1))
axs[2].set_yticklabels(np.arange(0,  model.fc.weight.shape[0],
1))
for i in range(model.fc.weight.shape[0]):
    for j in range(model.fc.weight.shape[1]):
        axs[2].text(j,        i,        f'{model.fc.weight[i,
j].item():.2f}',   ha='center',   va='center',   color='white',
fontsize=10)

plt.tight_layout()
plt.show()
```

Código-fonte 4 - Visualizando os pesos sinápticos de uma RNN
Fonte: Elaborado pelo autor (2025)

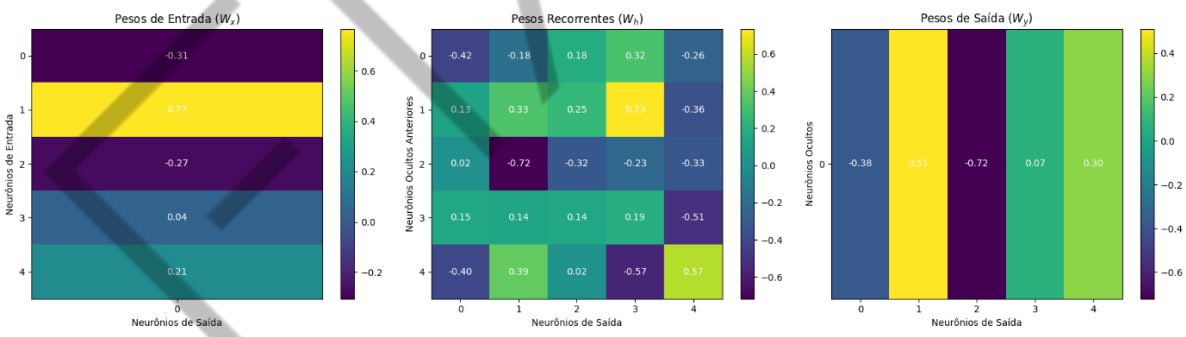

Figura 3 - Saída do Código-fonte 4, pesos sinápticos da RNN do Código-fonte 2
Fonte: Elaborado pelo autor (2025)

2.3 Criando uma RNN para processar texto – Autocompletar NLG

Agora, vamos usar os mesmos conceitos, mas para processar dados de texto. Usaremos como corpus de treinamento o livro *Moby Dick* de Herman Melville, em inglês. Ele pode ser obtido no Projeto Gutenberg, ou utilizando a própria biblioteca do NLTK. Primeiro, é preciso baixar e organizar os dados:

```
import nltk
import torch
import torch.nn as nn
import torch.optim as optim
from nltk.corpus import gutenberg
from collections import Counter
import numpy as np
import random

# Baixar recursos
nltk.download('gutenberg')
nltk.download('punkt')

# Carregar texto de Moby Dick
text = gutenberg.raw('melville-moby_dick.txt')
tokens = nltk.word_tokenize(text.lower())[:10000]  # Reduzir
tamanho para treinamento rápido

# Construir vocabulário
word_counts = Counter(tokens)
vocab = sorted(word_counts, key=word_counts.get, reverse=True)
word_to_ix = {word: i for i, word in enumerate(vocab)}
ix_to_word = {i: word for word, i in word_to_ix.items()}
vocab_size = len(vocab)

# Parâmetros
sequence_length = 5
embedding_dim = 64
hidden_dim = 128
batch_size = 32
epochs = 5
device = torch.device('cuda' if torch.cuda.is_available() else
'cpu')

# Preparar dados (janelas deslizantes)
data = []
for i in range(len(tokens) - sequence_length):
    seq = tokens[i:i+sequence_length]
    target = tokens[i+sequence_length]
    data.append((seq, target))

def vectorize(sequence):
    return torch.tensor([word_to_ix[word] for word in
sequence], dtype=torch.long)
```

Código-fonte 5 - Preparação dos dados textuais para treino de uma RNN
Fonte: Elaborado pelo autor (2025)

Observe que já configuramos alguns parâmetros relacionados a rede neural: o tamanho da sequência de palavras, a dimensão do embedding de texto e o número de neurônios da camada oculta. Também escolhemos o dispositivo de hardware que

será usado: de preferência, redes neurais artificiais, necessitam de placas de vídeo (GPU), mas caso esse recurso não esteja disponível, utilizamos a CPU.

O detalhe mais importante, contudo, é a janela deslizante. Em sinais contínuos — como no exemplo da figura “Saída do Código-fonte 3, processando série temporal com RNN” —, o truque da janela parece trivial, funcionando como uma breve memória anterior do próximo valor no tempo. Para NLP, esse truque demonstra sua força, recebendo até um termo próprio: o **aprendizado autosupervisionado**.

O aprendizado autosupervisionado é uma técnica de aprendizado semi-supervisinado onde não rotulamos os dados, porém utilizamos a própria estrutura dos dados como rótulo. Perceba que estamos utilizando uma sequência de tokens (palavras) para prever o próximo token (lembre-se que token são em si, dados categóricos). Como já temos a frase, basta esconder do modelo o token target e treinar com os tokens precedentes para prever o token target. Como sabemos qual é esse token a princípio, o modelo pode alterar os pesos neurais até que o embedding predito seja correspondente ao token que de fato veio na sequência. Para o treinamento, funciona como se de fato tivéssemos dados previamente rotulados; já na inferência, funciona como se tivéssemos uma **memória do contexto precedente** do aparecimento do token.

No código-fonte a seguir, construímos a RNN que irá operar sobre os dados do código-fonte “Preparação dos dados textuais para treino de uma RNN”:

```
class RNNPredictor(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size,
embedding_dim)
        self.rnn = nn.RNN(embedding_dim, hidden_dim,
batch_first=True)
        self.fc = nn.Linear(hidden_dim, vocab_size)

    def forward(self, x):
        embedded = self.embedding(x)
        output, hidden = self.rnn(embedded)
        out = self.fc(output[:, -1, :])  # Última saída da
sequência
        return out

# Inicializar modelo
model = RNNPredictor(vocab_size, embedding_dim,
hidden_dim).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Treinamento
for epoch in range(epochs):
    random.shuffle(data)
    total_loss = 0
    for i in range(0, len(data) - batch_size, batch_size):
        batch = data[i:i+batch_size]
        seqs = torch.stack([vectorize(x[0]) for x in
batch]).to(device)
        targets = torch.tensor([word_to_ix[x[1]] for x in
batch], dtype=torch.long).to(device)

        optimizer.zero_grad()
        outputs = model(seqs)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()
    print(f"Epoch {epoch+1}, Loss: {total_loss:.4f}")
```

Código-fonte 6 - Criando uma RNN para processar texto usando o PyTorch
Fonte: Elaborado pelo autor (2025)

Essa RNN irá gerar a próxima palavra dado um contexto de 5 palavras. Iremos validar para que a janela de contexto permita apenas tokens conhecidos (previamente no vocabulário do sistema). Assim como um modelo n-gram, esse sistema pode ser entendido como um modelo de geração de linguagem natural (NLG).

Observe como fazer isso no código-fonte a seguir:

```
def predict_next(model, sequence):
    # Verificar se todas as palavras estão no vocabulário
    unknown_words = [word for word in sequence[-
sequence_length:] if word not in word_to_ix]
    if unknown_words:
        print(f"Palavras fora do vocabulário:
{unknown_words}")
        return None

    model.eval()
    with torch.no_grad():
        seq_vec = vectorize(sequence[-
sequence_length:]).unsqueeze(0).to(device)
        output = model(seq_vec)
        predicted_idx = torch.argmax(output, dim=1).item()
        return ix_to_word[predicted_idx]

# Exemplo de predição
context = ["the", "white", "whale", "was", "a"]
#context = ["captain", "!", "do", "you", "have"]
next_word = predict_next(model, context)
if next_word:
    print(f"Contexto: {' '.join(context)} -> Próxima palavra:
{next_word}")
```

Código-fonte 7 - Usa da RNN treinada no Código-fonte 6
Fonte: Elaborado pelo autor (2025)

Importante: no jargão de NLP, **contexto** é o conjunto de tokens que serão processados pelo modelo naquela etapa. Por exemplo, uma janela de contexto de 5 significa que são necessários 5 tokens para predizer o seguinte; logo uma janela de 1024 tokens significa que o modelo consegue levar em consideração uma grande quantidade de informação a cada nova predição.

2.4 O que é uma LSTM?

O uso de Long Short-Term Memory em Redes Neurais Recorrentes tem sido essencial para superar as limitações associadas à aprendizagem de dependências de longo prazo em dados sequenciais, especialmente em tarefas de NLP. Embora as RNNs tradicionais sejam eficazes em capturar padrões de curto prazo, elas frequentemente enfrentam dificuldades para manter informações relevantes por muitos passos de tempo, devido ao problema do gradiente que desaparece ou explode.

As LSTMs foram projetadas especificamente para mitigar esses problemas. Elas introduzem uma estrutura de célula de memória mais sofisticada que controla o fluxo de informações por meio de mecanismos chamados portas: porta de entrada, porta de esquecimento e porta de saída. Essas portas permitem que a LSTM selecione quais informações devem ser adicionadas, esquecidas ou passadas adiante, melhorando significativamente a capacidade da rede de aprender dependências de longo prazo.

A figura mostra um esquema básico de uma célula LSTM:

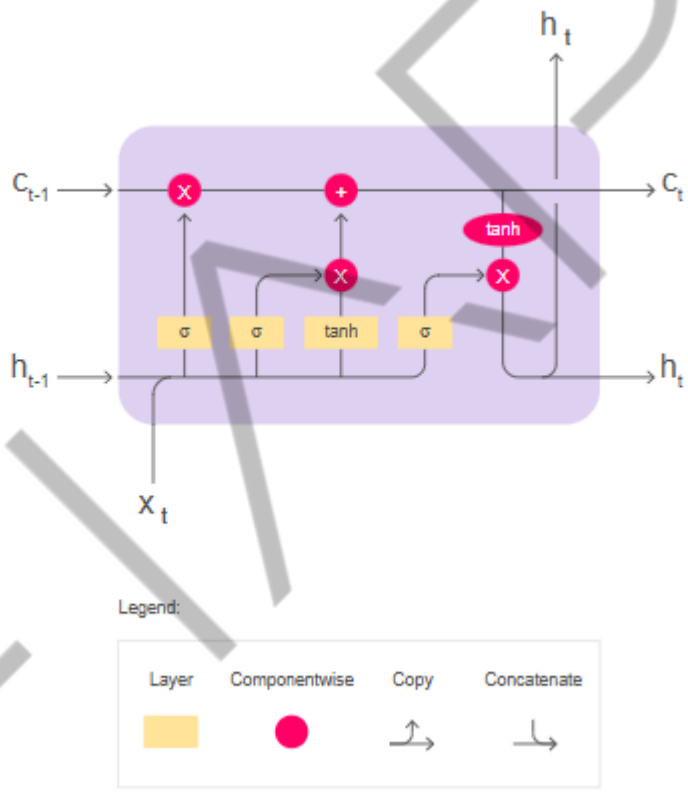

Figura 4 - Esquema de arquitetura de uma LSTM
Fonte: Google imagens (2025), adaptado por FIAP (2025)

Uma célula LSTM é composta por um vetor de estado da célula C_t, que representa a memória de longo prazo, além de um vetor de estado oculto h_t, que representa a saída da célula naquele instante. Matematicamente, o funcionamento de uma célula LSTM pode ser descrito pelas seguintes equações:

1. Entradas e estados:
   - x_t: vetor de entrada no tempo t.
   - h_{t-1}: estado oculto da LSTM no tempo $t-1$.

- C_{t-1}: estado da célula de memória no tempo $t-1$.

2. Portas:
   - Porta de esquecimento f_t: $f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right)$
   - Porta de entrada i_t e vetor de entrada de candidatos $\widetilde{C_t}$:
   - $i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \quad \widetilde{C_t} = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$
   - Porta de saída o_t: $o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$
3. Atualização do estado da célula:
   - O estado da célula C_t é atualizado como: $C_t = f_t \odot \mathrm{C_{t-1}} + i_t \odot \widetilde{C_t}$
   - Aqui, $\odot$ denota a multiplicação element-wise (produto de Hadamard).
4. Atualização do estado oculto:
   - O estado oculto h_t é calculado como: $h_t = o_t \tanh(C_t)$
5. Funções de ativação:
   - σ é a função sigmoide, que mapeia valores para o intervalo (0, 1).
   - $\tanh$ é a função tangente hiperbólica, que mapeia valores para o intervalo (-1, 1).

A principal vantagem das LSTMs está na sua habilidade de regular seletivamente o fluxo de informações por meio de portas de esquecimento e de entrada. Isso permite ao modelo, por exemplo, reter o sujeito de uma frase mesmo quando o predicado só aparece vários tokens adiante. Essa capacidade não é apenas teórica, ela se reflete diretamente na performance de modelos em tarefas como rotulação de sequência (NER, POS tagging), tradução automática neural e sumarização. A retenção de longo prazo viabilizada pelas LSTMs resulta em representações contextuais mais estáveis e sensíveis a estruturas sintáticas e semânticas mais amplas.

2.5 Criando uma LSTM para processar texto – Exemplo de NER por ML

Vejamos um exemplo de como implementar o Reconhecimento de Entidades Nomeadas (NER) utilizando redes LSTM. Primeiramente, precisaremos de um dataset

rotulado (com os valores das entidades). Vamos aproveitar esse dataset para realizar uma avaliação de desempenho no conjunto de treinamento usando uma métrica específica para NER presente na biblioteca `seqeval`.

Para instalar as dependências necessárias, execute:

```
!pip install datasets seqeval
```

Comando de prompt 3 - Instalação do datasets e do seqeval
Fonte: Elaborado pelo autor (2025)

Agora, vamos importar o dataset CoNLL-2003 que contém arquivos de dados em inglês e alemão. Os dados em inglês foram extraídos do Corpus da Reuters, e consiste em notícias da Reuters entre agosto de 1996 e agosto de 1997.

```
from datasets import load_dataset

# Carregando o dataset CoNLL2003
dataset = load_dataset("lhoestq/conll2003", revision="main")

# Visualização de exemplo
for i in range(0,5):
    print(dataset["train"][i],'\n')

# Lista de rótulos NER
ner_label_list = [
    "O", "B-PER", "I-PER", "B-ORG", "I-ORG", "B-LOC", "I-LOC",
"B-MISC", "I-MISC"
]
```

Código-fonte 8 - Carregando dados do CoNLL2003
Fonte: Elaborado pelo autor (2025)

O resultado é apresentado na próxima figura. Observe que na chave `tokens` há uma lista com todas as palavras; na chave `ner_tags` existem números que representam as entidades nomeadas. A lista de rótulos dessas entidades também está no código-fonte "Carregando dados do CoNLL2003", sendo que cada número pode ser diretamente mapeado para a posição correspondente na lista. Assim, o valor 3 na primeira posição do vetor `ner_tags` (primeira frase da figura "Exemplos no dataset CoNLL2003, resultado do Código-fonte 8") representa a entidade `B-ORG` (correspondente a posição 3 do vetor `ner_label_list` no código fonte anterior). A lista `ner_label_list` define rótulos seguindo a convenção BIO (Begin, Inside, Outside): `O` representa palavras que não fazem parte de nenhuma entidade; `B-PER` e `I-PER` indicam, respectivamente, o início e o interior de entidades do tipo pessoa; `B-ORG` e `I-ORG` são usados para marcar o início e o interior de entidades do tipo

organização; `B-LOC` e `I-LOC` identificam o início e o interior de entidades de localização; e `B-MISC` e `I-MISC` são utilizados para marcar o início e o interior de entidades misc.


```
{'id': '0', 'tokens': ['EU', 'rejects', 'German', 'call', 'to', 'boycott', 'British', 'lamb', '.'], 'pos_tags': [22, 42, 16, 21, 35, 37, 16, 21, 7], 'chu
nk_tags': [11, 21, 11, 12, 21, 22, 11, 12, 0], 'ner_tags': [3, 0, 7, 0, 0, 0, 7, 0, 0]}

{'id': '1', 'tokens': ['Peter', 'Blackburn'], 'pos_tags': [22, 22], 'chunk_tags': [11, 12], 'ner_tags': [1, 2]}

{'id': '2', 'tokens': ['BRUSSELS', '1996-08-22'], 'pos_tags': [22, 11], 'chunk_tags': [11, 12], 'ner_tags': [5, 0]}

{'id': '3', 'tokens': ['The', 'European', 'Commission', 'said', 'on', 'Thursday', 'it', 'disagreed', 'with', 'German', 'advice', 'to', 'consumers', 'to',
'shun', 'British', 'lamb', 'until', 'scientists', 'determine', 'whether', 'mad', 'cow', 'disease', 'can', 'be', 'transmitted', 'to', 'sheep', '.'], 'pos_
tags': [12, 22, 22, 38, 15, 22, 28, 38, 15, 16, 21, 35, 24, 35, 37, 16, 21, 15, 24, 41, 15, 16, 21, 21, 20, 37, 40, 35, 21, 7], 'chunk_tags': [11, 12, 1
2, 21, 13, 11, 11, 21, 13, 11, 12, 13, 11, 21, 22, 11, 12, 17, 11, 21, 17, 11, 12, 12, 21, 22, 22, 13, 11, 0], 'ner_tags': [0, 3, 4, 0, 0, 0, 0, 0, 0, 7,
0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]}

{'id': '4', 'tokens': ['Germany', "'s", 'representative', 'to', 'the', 'European', 'Union', "'s", 'veterinary', 'committee', 'Werner', 'Zwingmann', 'sai
d', 'on', 'Wednesday', 'consumers', 'should', 'buy', 'sheepmeat', 'from', 'countries', 'other', 'than', 'Britain', 'until', 'the', 'scientific', 'advic
e', 'was', 'clearer', '.'], 'pos_tags': [22, 27, 21, 35, 12, 22, 22, 27, 16, 21, 22, 22, 38, 15, 22, 24, 20, 37, 21, 15, 24, 16, 15, 22, 15, 12, 16, 21,
38, 17, 7], 'chunk_tags': [11, 11, 12, 13, 11, 12, 12, 11, 12, 12, 12, 12, 21, 13, 11, 12, 21, 22, 11, 13, 11, 1, 13, 11, 17, 11, 12, 12, 21, 1, 0], 'ner
_tags': [5, 0, 0, 0, 0, 0, 3, 4, 0, 0, 0, 1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0]}
```


Figura 5 - Exemplos no dataset CoNLL2003, resultado do Código-fonte 8
Fonte: Elaborado pelo autor (2025)

Agora, vamos ajustar o dataset para o tipo de processamento sequencial de uma RNN. Iremos utilizar uma janela de contexto de 128 tokens (`MAX_LEN=128`). Como nem todas as frases do dataset possuem 128 tokens, então precisamos preencher as janelas de contexto com um token coringa — o token de `PAD` (de preenchimento, do inglês, padding). Aquelas que possuem mais de 128 tokens serão truncadas (cortadas). Além disso, eventualmente teremos que processar frases com tokens que não existiam no dataset de treinamento. Essas palavras desconhecidas serão representadas pelo token coringa `UNK` (de desconhecido, do inglês, unknown). As funções `tag2idx` e `idx2tag` fazem o mapeamento das entidades nomeadas para o valor (índice) correspondente no dataset. Já a função `encode_examples` transforma exemplos de texto com anotações NER em arrays numéricos prontos para serem usados. Ela realiza a conversão dos tokens em índices numéricos (usando um vocabulário); o truncamento e padding das sequências para um comprimento fixo (`MAX_LEN`); e a conversão dos rótulos NER em vetores one-hot.

```
import numpy as np
from tensorflow.keras.preprocessing.sequence import pad_sequences

# Construção dos vocabulários
all_words = set(w for example in dataset['train'] for w in example['tokens'])
word2idx = {w: i + 2 for i, w in enumerate(sorted(all_words))}
word2idx["PAD"] = 0
word2idx["UNK"] = 1
```

```
tag2idx = {label: i for i, label in enumerate(ner_label_list)}
idx2tag = {i: label for label, i in tag2idx.items()}

MAX_LEN = 128

# Função de codificação
def encode_examples(dataset_split):
    input_ids = []
    label_ids = []

    for example in dataset_split:
        tokens = example['tokens']
        tags = example['ner_tags']

        # Token IDs com fallback para UNK
        token_ids = [word2idx.get(w, word2idx['UNK']) for w in
tokens]
        tag_ids = tags

        # Truncar
        token_ids = token_ids[:MAX_LEN]
        tag_ids = tag_ids[:MAX_LEN]

        input_ids.append(token_ids)
        label_ids.append(tag_ids)

    # Padding
    input_ids   =   pad_sequences(input_ids,   maxlen=MAX_LEN,
padding="post", value=word2idx["PAD"])
    label_ids   =   pad_sequences(label_ids,   maxlen=MAX_LEN,
padding="post", value=tag2idx["O"])

    # One-hot encoding dos rótulos
    label_ids = [to_categorical(seq, num_classes=len(tag2idx))
for seq in label_ids]

    return np.array(input_ids), np.array(label_ids)

# Usando um subconjunto para treino mais rápido
X_train,                        y_train                        =
encode_examples(dataset['train'].select(range(2000)))
```

Código-fonte 9 - Preparação dos dados para o modelo LSTM
Fonte: Elaborado pelo autor (2025)

O modelo LSTM então é treinado conforme o código-fonte a seguir:

```
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Embedding, LSTM,
TimeDistributed, Dense

input = Input(shape=(MAX_LEN,))
model = Embedding(input_dim=len(word2idx), output_dim=128,
input_length=MAX_LEN)(input)
model = LSTM(units=64, return_sequences=True)(model)
output = TimeDistributed(Dense(len(tag2idx),
activation="softmax"))(model)

model = Model(inputs=input, outputs=output)
model.compile(optimizer="adam",
loss="categorical_crossentropy", metrics=["accuracy"])
model.summary()

# Treinamento
model.fit(X_train, y_train, batch_size=2, epochs=10)
```

Código-fonte 10 - Criação da rede LSTM para NER
Fonte: Elaborado pelo autor (2025)

O resultado do treinamento é apresentado na próxima figura, onde podemos ver o número de parâmetros (peso sinpaticos). Perceba que a camada de embedding tem muitos parâmetros porque o vocabulário é grande (23.625 tokens), e cada token é mapeado para um vetor denso de 128 dimensões. O input codifica os tokens como índices inteiros, usados para buscar esses vetores. Basicamente o número de parâmetros dessa camada é o tamanho do vocabulário (número de tokens) vezes a dimensão do embedding.

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 128)	0
embedding (Embedding)	(None, 128, 128)	3,024,000
lstm (LSTM)	(None, 128, 64)	49,408
time_distributed (TimeDistributed)	(None, 128, 9)	585

```
Total params: 3,073,993 (11.73 MB)
Trainable params: 3,073,993 (11.73 MB)
Non-trainable params: 0 (0.00 B)
Epoch 1/10
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 19s 16ms/step - accuracy: 0.9709 - loss: 0.1836
Epoch 2/10
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 16s 16ms/step - accuracy: 0.9875 - loss: 0.0407
Epoch 3/10
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 16s 16ms/step - accuracy: 0.9940 - loss: 0.0203
Epoch 4/10
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 17s 17ms/step - accuracy: 0.9977 - loss: 0.0092
Epoch 5/10
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 16s 16ms/step - accuracy: 0.9990 - loss: 0.0046
Epoch 6/10
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 17s 17ms/step - accuracy: 0.9994 - loss: 0.0022
Epoch 7/10
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 16s 16ms/step - accuracy: 0.9997 - loss: 0.0014
Epoch 8/10
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 16s 16ms/step - accuracy: 0.9997 - loss: 9.6704e-04
Epoch 9/10
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 16s 16ms/step - accuracy: 0.9998 - loss: 8.1264e-04
Epoch 10/10
1000/1000 ━━━━━━━━━━━━━━━━━━━━ 16s 16ms/step - accuracy: 0.9998 - loss: 6.8293e-04
```

Figura 6 - Modelo e treinamento da LSTM para NER criada no Código-fonte 10
Fonte: Elaborado pelo autor (2025)

Para testar o modelo com as métricas de desempenho de NER, podemos fazer conforme o código-fonte a seguir. O relatório de desempenho resultante é mostrado na figura “Relatório de avaliação de desempenho da LSTM para NER”. Observa-se que as melhores métricas ocorrem para a entidade de `LOC`; os piores resultados para `MISC`.

```
from seqeval.metrics import classification_report

X_test, y_test = encode_examples(dataset["validation"].select(range(200)))
y_pred = model.predict(X_test)
y_pred = np.argmax(y_pred, axis=-1)
y_true = np.argmax(y_test, axis=-1)

# Função de conversão dos índices para tags
def decode_predictions(y_indices):
```

```
    decoded = []
    for seq in y_indices:
        # Remove o padding
        decoded_seq = [idx2tag[idx] for idx in seq if idx !=
tag2idx["O"]]
        decoded.append(decoded_seq)
    return decoded

y_pred_tags = decode_predictions(y_pred)
y_true_tags = decode_predictions(y_true)

def adjust_for_padding(y_true, y_pred):
    adjusted_true = []
    adjusted_pred = []

    for true, pred in zip(y_true, y_pred):
        true_filtered = [label for label in true if label !=
"O"]
        pred_filtered = [label for label in pred if label !=
"O"]

        if len(true_filtered) == len(pred_filtered):
            adjusted_true.append(true_filtered)
            adjusted_pred.append(pred_filtered)
        else:
            print(f"Warning: Length mismatch after filtering
(True: {len(true_filtered)}, Pred: {len(pred_filtered)})")

    return adjusted_true, adjusted_pred

y_true_filtered,                y_pred_filtered                =
adjust_for_padding(y_true_tags, y_pred_tags)

# Relatório de classificação
print(classification_report(y_true_filtered,
y_pred_filtered))
```

Código-fonte 11 - Testando a LSTM para NER
Fonte: Elaborado pelo autor (2025)

	precision	recall	f1-score	support
LOC	0.88	0.92	0.90	50
MISC	0.75	0.75	0.75	12
ORG	0.83	0.87	0.85	52
PER	0.80	0.82	0.81	40
micro avg	0.84	0.86	0.85	154
macro avg	0.82	0.84	0.83	154
weighted avg	0.84	0.86	0.85	154

Figura 7 - Relatório de avaliação de desempenho da LSTM para NER
Fonte: Elaborado pelo autor (2025)

Podemos usar o modelo LSTM criado para fazer novas inferências em frases, conforme o código-fonte a seguir:

```
def infer(input_sentence):
    # 1. Tokenização da sentença de entrada
    tokens = input_sentence.split()

    # 2. Codificação dos tokens para IDs
    token_ids = [word2idx.get(w, word2idx['UNK']) for w in
tokens]

    # Truncar (similar ao que foi feito na encode_examples)
    token_ids = token_ids[:MAX_LEN]

    # 3. Padding
    token_ids = pad_sequences([token_ids], maxlen=MAX_LEN,
padding="post", value=word2idx["PAD"])

    # 4. Predição (obtenção das tags preditas pelo modelo)
    y_pred = model.predict(token_ids)
    y_pred = np.argmax(y_pred, axis=-1)

    # 5. Decodificação das predições
    y_pred_tags = decode_predictions(y_pred)

    # 6. Exibição dos resultados
    for token, tag in zip(tokens, y_pred_tags[0]):
        print(f"Token: {token}, Predicted tag: {tag}")

input_sentence = "Bill Clinton was a president of USA"
infer(input_sentence)
```

Código-fonte 12 - Usando LSTM NER em novas frases
Fonte: Elaborado pelo autor (2025)

O resultado é apresentado na figura “Inferência usando a LSTM para NER”. Perceba que Bill Clinton foi corretamente reconhecido como B-PER e I-PER, respectivamente. No entanto, o modelo se confundiu: rotulou was como B-MISC, o que é um erro. Isso ocorreu pois não estamos utilizando o dataset inteiro para treinar o LSTM. Além disso, a LSTM só processa os dados da esquerda para a direita, isto é, apenas os tokens precedentes são usados para prever a classe do token alvo. Para observar toda a estrutura da frase, tanto aquilo que está à esquerda quanto à direita do token alvo, precisamos utilizar LSTM Bidirecionais.

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step
Token: Bill, Predicted tag: B-PER
Token: Clinton, Predicted tag: I-PER
Token: was, Predicted tag: B-MISC
```

Figura 8 - Inferência usando a LSTM para NER
Fonte: Elaborado pelo autor (2025)

2.6 Redes LSTM bidirecionais

Uma LSTM bidirecional, ou Bidirectional Long Short-Term Memory, é uma variação da arquitetura de redes neurais recorrentes conhecida como LSTM, projetada para capturar dependências temporais em sequências de dados. Ao contrário da LSTM tradicional, que processa a sequência apenas em uma direção — normalmente do passado para o futuro —, a LSTM bidirecional processa a sequência em duas direções: uma para frente e outra para trás. Isso significa que, para cada ponto da sequência, o modelo considera não apenas o contexto anterior, mas também o posterior. Essa abordagem é especialmente valiosa em tarefas de NLP, onde o significado de uma palavra ou frase muitas vezes depende do contexto completo, tanto o que vem antes quanto o que vem depois.

Em aplicações como análise de sentimento, tradução automática, reconhecimento de entidades nomeadas e modelagem de linguagem, essa visão mais ampla oferecida pelas LSTMs bidirecionais permite ao modelo realizar inferências mais precisas, capturando nuances do texto que seriam perdidas se apenas a direção passada fosse considerada. Por exemplo, a palavra “banco” pode ter sentidos diferentes dependendo das palavras que vêm depois dela, como “banco de dados” ou “banco de jardim”, e a LSTM bidirecional pode utilizar essas informações contextuais em ambas as direções para desambiguar corretamente o significado.

O conceito de contexto bidirecional pode ser melhor compreendido quando o relacionamos com duas ideias fundamentais no processamento de linguagem natural moderno: **aprendizado autosupervisionado** e **janela de contexto**. No aprendizado autosupervisionado, o modelo aprende representações úteis da linguagem a partir de grandes quantidades de texto não rotulado, utilizando tarefas onde ele próprio gera os rótulos a partir dos dados. Um exemplo clássico é a tarefa de preenchimento de lacunas, como no modelo BERT, onde o objetivo é prever palavras faltando no meio de uma frase com base no restante do contexto. Esse tipo de aprendizado exige que

o modelo entenda tanto o que vem antes quanto o que vem depois da palavra-alvo, o que exige uma compreensão bidirecional da sequência.

A janela de contexto, por sua vez, é o trecho de texto que o modelo considera ao fazer uma previsão ou ao gerar uma representação. Em modelos tradicionais unidirecionais — como as LSTMs comuns ou modelos autorregressivos como o GPT —, essa janela se estende apenas no sentido passado → futuro; ou seja, o modelo só pode ver as palavras anteriores para prever a próxima. Isso limita sua capacidade de capturar relações que dependem de palavras futuras. Já em uma LSTM bidirecional ou em modelos como o BERT, a janela de contexto é expandida para ambos os lados da palavra-alvo, permitindo que o modelo integre informações tanto do passado quanto do futuro ao processar cada palavra.

O próximo código-fonte mostra como criar uma LSTM Bidirecional. Substitua o código-fonte "Criação da rede LSTM para NER" por este no exemplo do NER. Como fica o reconhecimento de entidades para a frase "Bill Clinton was a president of USA" para a mesma quantidade de dados de treinamento?

```
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Embedding, LSTM,
TimeDistributed, Dense, Bidirectional

input = Input(shape=(MAX_LEN,))
model = Embedding(input_dim=len(word2idx), output_dim=128,
input_length=MAX_LEN)(input)
model = Bidirectional(LSTM(units=64,
return_sequences=True))(model)
output = TimeDistributed(Dense(len(tag2idx),
activation="softmax"))(model)

model = Model(inputs=input, outputs=output)
model.compile(optimizer="adam",
loss="categorical_crossentropy", metrics=["accuracy"])
model.summary()

model.fit(X_train, y_train, batch_size=2, epochs=10)
```

Código-fonte 13 - Criação da rede LSTM Bidirecional para NER
Fonte: Elaborado pelo autor (2025)

3 REDES NEURAIS TRANSFORMERS

3.1 O que são Transformers?

Não, os Transformers de que estamos falando não são robôs gigantes! Transformers é o nome de uma arquitetura de rede neural artificial feita exatamente para o processamento de linguagem natural. A arquitetura Transformer marcou uma mudança significativa no panorama NLP e do Aprendizado Profundo quando foi primeiramente apresentada no artigo *Attention is All You Need* de engenheiros do Google. Essa arquitetura foi desenvolvida para superar as limitações das estruturas sequenciais tradicionais, como RNNs e LSTMs, e acabou se tornando a base de modelos sofisticados como o BERT e o GPT. Ao empregar um mecanismo de atenção, os Transformers conseguem analisar sequências inteiras de uma só vez, em vez de processá-las passo a passo. Essa característica aumenta a eficiência computacional, permite maior paralelização e oferece uma melhor capacidade de capturar relações de longo alcance em dados sequenciais.

A figura a seguir apresenta um diagrama de representação da arquitetura Transformer. Perceba que a rede neural incorpora aspectos das redes recorrentes e LSTM, dadas pelas sequências de palavras (tokens) que são processados. Entraremos em detalhes de cada bloco constitutivo nas próximas seções.

Figura 9 - Fluxo de processamento em uma arquitetura Transformer
Fonte: Jurafsky (2025), adaptado por FIAP (2025)

3.2 Embedding semântico

Na arquitetura de um Transformer, a primeira camada responsável pelo processamento dos dados de entrada é a camada de **embedding**, cuja função é mapear cada token discreto (por exemplo, palavras, subpalavras ou caracteres) para um vetor denso em um espaço de dimensão fixa d_{modelo}. Normalmente, esse mapeamento é implementado por meio de uma matriz de pesos aprendida, onde cada linha corresponde ao vetor de embedding associado a um token específico do vocabulário. Na próxima figura, apresentamos um exemplo de embedding com $d_{modelo} = 4$, ou seja, cada token é mapeado para um vetor de 4 dimensões. O bloco demarcado com a letra E representa a matriz de transformação, que pode ser compreendida como um tipo de neurônio artificial da primeira camada da rede:

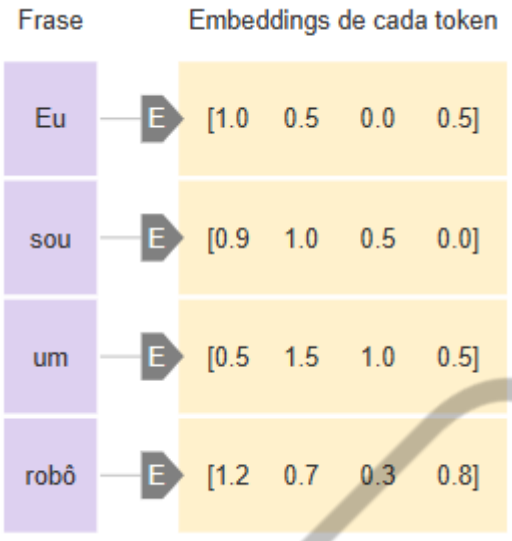

Figura 10 - Exemplo de embedding semântico
Fonte: Elaborado pelo autor (2025)

3.3 Codificação posicional

Além do embedding semântico, o Transformer incorpora uma **codificação posicional** para injetar informação sobre a ordem dos tokens na sequência, uma vez que a arquitetura, por natureza, não é sequencial.

> Lembre-se que gostaríamos de manter aspetos sequências como nas redes recorrentes.

O resultado dessa etapa é uma sequência de vetores contínuos que preserva tanto a identidade do token quanto sua posição relativa, servindo como entrada para as subsequentes camadas de atenção e feed-forward.

Para construir o vetor que codifica a posição original dos tokens, usamos um pouco de trigonometria. Considere que você tenha uma sequência de entrada de comprimento L e precise encontrar a posição do k-ésimo token dentro dessa sequência. A codificação posicional P é dada pelas funções seno e cosseno de frequências variadas, onde d é a dimensão do embedding usado para os tokens, n é um número definido pelo arquiteto da rede (no Transformer original os engenheiros do Google usaram $n = 10000$) e i é utilizado para mapear as funções corretamente:

$$P(k, 2i) = \sin\left(\frac{k}{n^{2i/d}}\right)$$

$$P(k, 2i+1) = \cos\left(\frac{k}{n^{2i/d}}\right)$$

Na figura a seguir, aplicamos as funções anteriores para criar os vetores posicionais de quatro tokens.

Consideramos $n = 100$ e a dimensão do embedding $d = 4$.

Sequência de entrada	Índice do token k	i = 0	i = 0	i = 1	i = 1
Eu	0	$P(0,0) = \sin(0) = 0$	$P(0,1) = \cos(0) = 1$	$P(0,2) = \sin(0) = 1$	$P(0,3) = \cos(0) = 1$
sou	1	$P(1,0) = \sin\left(\frac{1}{1}\right) = 0.84$	$P(1,1) = \cos\left(\frac{1}{10}\right) = 0.54$	$P(1,2) = \sin\left(\frac{1}{10}\right) = 0.10$	$P(1,3) = \cos\left(\frac{1}{10}\right) = 1$
um	2	$P(2,0) = \sin\left(\frac{2}{1}\right) = 0.91$	$P(2,1) = \cos\left(\frac{2}{1}\right) = -0.42$	$P(2,2) = \sin\left(\frac{2}{10}\right) = 0.20$	$P(2,3) = \cos\left(\frac{2}{10}\right) = 0.98$
robô	3	$P(3,0) = \sin\left(\frac{3}{1}\right) = 0.14$	$P(3,1) = \cos\left(\frac{3}{1}\right) = -0.99$	$P(3,2) = \sin\left(\frac{3}{10}\right) = 0.30$	$P(3,3) = \cos\left(\frac{3}{10}\right) = 0.96$

Figura 11 - Codificação posicional na arquitetura Transformer
Fonte: Google imagens (2025), adaptado pelo autor (2025)

Com o embedding dos tokens gerado no passo anterior e os vetores posicionais, podemos criar o vetor de entrada para o bloco de atenção.

A próxima figura demonstra essa continuação do exemplo:

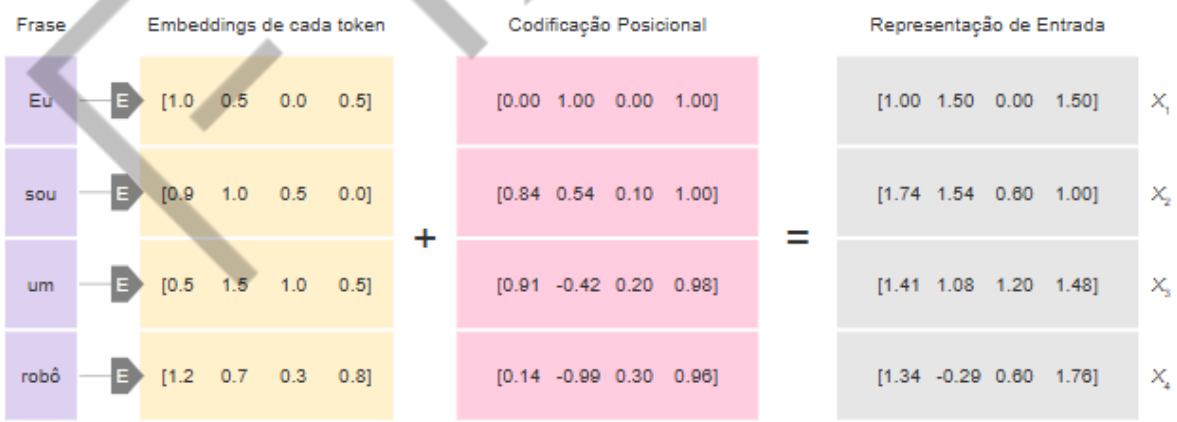

Figura 12 - Vetor de entrada da rede Transformer
Fonte: Elaborado pelo autor (2025)

3.4 Mecanismo de atenção

O mecanismo de atenção é uma técnica fundamental em modelos de Deep Learning — especialmente nos Transformers —, pois permite que o modelo foque nas

partes mais relevantes de uma sequência de entrada ao processar uma tarefa. Em vez de tratar cada elemento da sequência igualmente, a atenção atribui diferentes “pesos” a cada parte, dependendo de sua importância para a tarefa atual. Isso significa que, ao gerar uma saída (como uma palavra em uma tradução), o modelo pode considerar diretamente todas as palavras da entrada, dando mais destaque às que mais influenciam a resposta correta. Esse processo torna possível capturar dependências de longo alcance e melhorar o desempenho em tarefas complexas de linguagem natural.

O mecanismo de atenção funciona com base no cálculo de três vetores distintos para cada token da sequência de entrada: **Query** (Q), **Key** (K) e **Value** (V). Esses vetores são derivados dos embeddings dos tokens por meio de matrizes de projeção aprendidas durante o treinamento. A atenção é determinada ao calcular a similaridade entre a Query de um token e as Keys de todos os outros tokens na sequência, geralmente por meio do produto escalar. O resultado é um conjunto de pesos que indicam o quanto cada palavra da entrada deve influenciar a representação do token atual. Esses pesos são então aplicados aos vetores Value correspondentes, e a combinação ponderada desses valores resulta em uma nova representação contextualizada para cada token. Esse processo permite que o modelo considere, de forma paralela e eficiente, todas as palavras da sequência ao construir a representação de cada uma, capturando assim relações complexas de curto e longo alcance entre palavras.

Continuemos com o exemplo da frase “Eu sou um robô” sendo processada pela camada de entrada da rede Transformer. Após a geração do embedding final (embedding semântico + codificação posicional), devemos gerar os vetores de Query, Key e Value. Para isso, multiplicamos o embedding final E pelas matrizes de projeção W_Q, W_K e W_V. Essas matrizes são aprendidas pelo treino da rede, mas vamos supor que elas sejam conforme apresentadas na próxima figura. Assim, podemos calcular o vetor Query $Q = E \times W_Q$, o vetor Key $K = E \times W_K$ e o vetor Value $V = E \times W_V$.

$$W_Q = \begin{bmatrix} 1 & 0 & 0 & 0.5 \\ 0 & 1 & 0.5 & 0 \\ 0.5 & 0 & 1 & 0 \\ 0 & 0.5 & 0 & 1 \end{bmatrix} \quad W_K = \begin{bmatrix} 1 & 0.5 & 0 & 0 \\ 0 & 1 & 0 & 0.5 \\ 0.5 & 0 & 1 & 0 \\ 0 & 0 & 0.5 & 1 \end{bmatrix} \quad W_E = \begin{bmatrix} 1 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0.5 \\ 0.5 & 0 & 1 & 0 \\ 0 & 0.5 & 0 & 1 \end{bmatrix}$$

E (entrada)	Token	Q	K	V
[1.00 1.50 0.00 1.50]	eu	[1.00, 2.25, 0.50, 2.25]	[1.75, 1.75, 0.50, 2.25]	[1.25, 2.25, 0.50, 3.00]
[1.74 1.54 0.60 1.00]	sou	[2.24, 2.04, 1.47, 1.54]	[2.04, 2.24, 1.47, 1.84]	[2.04, 2.04, 1.47, 1.54]
[1.41 1.08 1.20 1.48]	um	[2.15, 1.82, 1.81, 2.02]	[2.15, 2.15, 1.81, 2.23]	[2.15, 1.82, 1.81, 2.02]
[1.34 -0.29 0.60 1.76]	robô	[2.22, 0.59, 1.97, 1.61]	[2.04, 0.55, 1.97, 1.46]	[2.04, 0.59, 1.97, 1.61]

Figura 13 - Vetores QKV do mecanismo de atenção
Fonte: Elaborado pelo autor (2025)

Em seguida, calculamos a similaridade entre a Query e a Key entre cada token, por exemplo “eu” com “sou”, “eu” com “um” e “eu” com “robô”. A similaridade $S(i,j)$ entre o token i e o token j é dada por:

$$S(i,j) = \sum_{t}^{d} Q_{i,t} K_{j,t}$$

A seguir, a matriz de similaridade para o exemplo da frase “eu sou um robô” é dada na figura:

	eu	sou	um	robô
eu	11.687	11.955	13.360	9.902
sou	13.018	12.629	14.033	9.889
um	13.536	13.176	14.620	10.485
robô	9.270	9.152	10.230	8.491

Figura 14 - Matriz de similaridade QK
Fonte: Elaborado pelo autor (2025)

Para construir a matriz de atenção final, os valores são normalizados usando a função softmax. O resultado pode ser visto na figura “Matriz de atenção”. A representação contextualizada será a multiplicação da matriz de atenção pelo vetor Value V calculado anteriormente.

	eu	sou	um	robô
eu	0.128	0.169	0.682	0.021
sou	0.263	0.178	0.550	0.009
um	0.274	0.192	0.523	0.011
robô	0.282	0.249	0.459	0.010

Figura 15 - Matriz de atenção
Fonte: Elaborado pelo autor (2025)

Na figura a seguir, apresentamos o resultado final da representação das palavras de entrada após cada etapa de processamento.

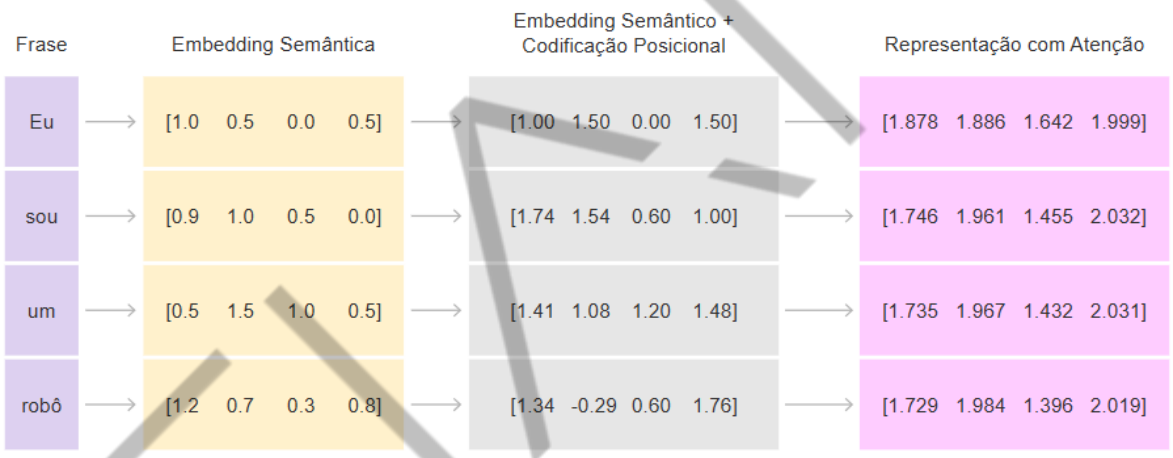

Figura 16 - Embeddings dos tokens ao longo do processamento
Fonte: Elaborado pelo autor (2025)

Um avanço importante sobre o mecanismo de atenção simples é a **atenção multi-cabeças** (do inglês, *multi-head attention*), que é central no funcionamento dos Transformers. Em vez de realizar apenas um único cálculo de atenção, o modelo executa várias “cabeças” de atenção em paralelo, cada uma com suas próprias projeções para Q, K e V. Isso permite que cada cabeça aprenda a capturar diferentes tipos de relações entre tokens: por exemplo, uma cabeça pode focar em relações sintáticas (como sujeito-verbo), enquanto outra pode capturar aspectos semânticos (como associação entre conceitos). Ao final, as saídas de todas as cabeças são concatenadas e projetadas de volta para o espaço de dimensão original, produzindo representações ricas e variadas.

3.5 Cabeça de modelagem de linguagem

A cabeça de modelagem de linguagem (do inglês, *language modeling head*) é a camada final de um Transformer responsável por transformar as representações internas de cada token em probabilidades sobre o vocabulário. Após as camadas de atenção e feed-forward, cada posição na sequência possui um vetor de embedding contextualizado, geralmente de dimensão d_{modelo}. O *language modeling head* é implementado como uma camada linear que multiplica esse vetor por uma matriz de pesos de tamanho $|V| \times d_{modelo}$ (onde $|V|$ é o tamanho do vocabulário) e adiciona um viés, gerando assim um vetor de logits. Em seguida, aplica-se a função softmax para converter esses logits em probabilidades, indicando a chance de cada palavra do vocabulário ocorrer naquela posição. No treinamento, essas probabilidades são comparadas com as palavras reais por meio de cross-entropy loss. Esse mecanismo é fundamental tanto para a predição da próxima palavra ou de palavras mascaradas quanto para a geração de texto e, em muitos modelos, os pesos dessa camada são compartilhados com a matriz de embeddings de entrada, reduzindo o número de parâmetros e melhorando a generalização.

No caso do exemplo com a frase "eu sou um robô", após passar pelos blocos de atenção e feed-forward, cada token ("eu", "sou", "um", "robô") teria um vetor final de dimensão dmodel=4d_{\text{model}} = 4dmodel=4 — como definimos nos cálculos anteriores. A cabeça de modelagem de linguagem pegaria cada um desses vetores e, por meio de sua matriz de pesos, projetaria cada embedding em um vetor de logits de tamanho igual ao vocabulário. Por exemplo, o vetor final de "sou" poderia gerar logits que, após o softmax, resultariam em altas probabilidades para "um" e baixas para palavras menos prováveis no contexto. Assim, o *language modeling head* atua como a última tradução entre a compreensão contextual do Transformer e a escolha efetiva das próximas palavras, permitindo que o modelo, a partir do estado interno, decida que "robô" é a continuação mais plausível para "eu sou um".

3.6 Juntando todas as peças

Agora que entendemos cada um dos componentes da arquitetura Transformer, podemos resumir o fluxo de processamento. Compare a próxima sequência com as

figuras “Fluxo de processamento em uma arquitetura Transformer” e “Arquitetura da RNA Transformer”:

- **Tokenização:** a frase "O gato dorme no sofá" é quebrada em unidades menores (tokens), que podem ser palavras inteiras ou pedaços de palavras (subwords), dependendo do tokenizador que for utilizado. Exemplo: ["O", "gato", "dorme", "no", "sofã"] (se fosse *word-level*) ou ["O", "gat", "o", "dorme", "no", "sof", "á"] (se fosse *subword-level*).
- **Mapeamento para índices:** cada token é mapeado para um número inteiro (ID) que representa sua posição no vocabulário do modelo. Exemplo: "gato" → 503, "dorme" → 1204, entre outros. Os vocabulários de modelos modernos podem ser enormes e incluir palavras de múltiplos idiomas, símbolos, emojis e até trechos de código-fonte de diferentes linguagens.
- **Camada de embedding:** esses índices (IDs) passam por uma matriz de embedding aprendida durante o treinamento, onde cada linha é o vetor associado a um token do vocabulário. O que acontece é que o embedding é apenas uma camada inicial de uma rede neural, e seus pesos são ajustados pelo mesmo processo de retropropagação que ajusta qualquer camada. Por exemplo: se o vetor tiver dimensão $d = 512$, então "gato" será representado por um vetor de 512 números reais, ajustados de forma que tokens com significados parecidos fiquem próximos nesse espaço vetorial.
- **Adição de codificação posicional:** como o embedding puro não traz noção de ordem, adiciona-se o **positional encoding** para informar ao modelo onde o token aparece na sequência.
- **Cálculo da atenção:** o vetor de cada token é projetado em três vetores diferentes: Query (Q), Key (K) e Value (V), usando matrizes de pesos aprendidas. As similaridades entre Queries e Keys são calculadas para gerar os pesos de atenção.
- **Aplicação dos pesos de atenção aos valores (Values):** os pesos calculados a partir da similaridade entre Queries e Keys são usados para ponderar os vetores Value correspondentes, gerando uma representação combinada que captura a importância relativa de cada token no contexto da frase.

- **Multi-head attention:** para permitir que o modelo capture diferentes tipos de relações entre tokens, a atenção é calculada em várias “cabeças” paralelas, cada uma com seus próprios parâmetros Q, K e V. As saídas dessas cabeças são concatenadas e projetadas novamente para formar a representação final da atenção.
- **Feed-forward e normalização:** após a atenção, cada vetor passa por uma camada feed-forward (uma rede neural simples, geralmente com duas camadas lineares e uma função de ativação) seguida por camadas de normalização e mecanismos de residual connection, que ajudam na estabilidade e aprendizado do modelo.
- **Empilhamento de camadas:** esses blocos de atenção multi-head seguidos pelo feed-forward são empilhados várias vezes (por exemplo, 12 vezes em BERT-base), permitindo que o modelo construa representações cada vez mais abstratas e ricas da entrada.
- **Language modeling head:** por fim, a saída final de cada token é passada por uma camada linear chamada *language modeling head*, que transforma as representações em distribuições de probabilidade sobre o vocabulário, permitindo prever a próxima palavra, preencher lacunas ou realizar outras tarefas de linguagem.

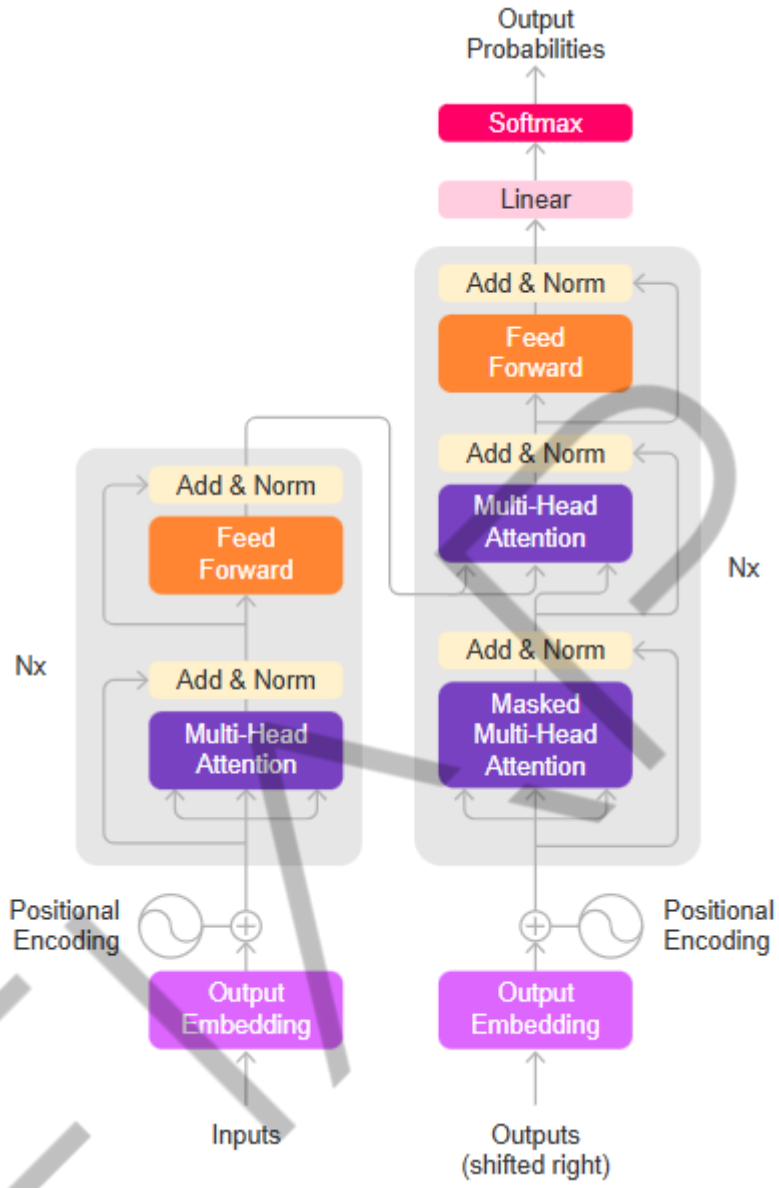

Figura 17 - Arquitetura da RNA Transformer
Fonte: Google imagens (2025), adaptado por FIAP (2025)

3.7 Exemplo de Transformer: O BERT

Para construirmos um Transformer do zero, precisaríamos não só implementar toda a arquitetura discutida anteriormente, como também usar dados de treinamento para que os pesos sinpaticos de cada camada fossem aprendidos. Isso pode ser muito custoso computacionalmente! Por isso, vamos utilizar um modelo pré-treinado.

O **BERT** (Bidirectional Encoder Representations from Transformers) é um modelo de linguagem baseado na arquitetura **Transformer**, projetado pelo Google em 2018 para aprender representações contextuais profundas de palavras. Sua principal característica é a capacidade de processar texto de forma **bidirecional**, ou seja,

levando em conta simultaneamente o contexto à esquerda e à direita de cada palavra, o que melhora a compreensão semântica em comparação a modelos unidirecionais. Ele é treinado de forma prévia (*pretraining*) em grandes corpora usando duas tarefas principais: **Masked Language Modeling** (prever palavras mascaradas) e **Next Sentence Prediction** (prever se uma frase segue outra no texto). Após esse treinamento genérico, o BERT pode ser ajustado (fine-tuning) para tarefas específicas de PLN, como classificação de textos, análise de sentimentos, resposta a perguntas e extração de informações.

O **BERTModel** utiliza alguns **tokens especiais** para estruturar a entrada e permitir que o modelo entenda o contexto e a segmentação do texto: `[CLS]`, no início da sequência, serve como representação global para tarefas como classificação; `[SEP]` separa sentenças ou marca o fim da entrada; `[PAD]` preenche sequências para terem o mesmo tamanho no *batch* e é ignorado pela atenção; e `[MASK]`, usado no pré-treinamento, permite ao modelo prever palavras ocultas no *Masked Language Modeling*. Esses tokens são inseridos automaticamente pelo tokenizador, garantindo que o texto siga o formato usado no treinamento.

No código-fonte a seguir, importamos o modelo BERT pré-treinado do Huggingface. Primeiramente, carregamos o módulo tokenizador (analisador léxico), responsável por quebrar o texto nos tokens no mesmo padrão daquele usado no treinamento do modelo. Em seguida, carregamos o modelo BERT em si. Configuramos a sentença “I am a robô” (“Eu sou um robô”) e tokenizamos ela com o tokenizador carregado. Também convertemos os tokens para os IDs do vocabulário para mostrar na tela. No código, não foram adicionados os tokens especiais [CLS] e [SEP] que o BERT normalmente usa. Através do tokens id, pegamos no modelo os embeddings contextuais que representam cada um dos tokens de entrada.

```
from transformers import BertTokenizer, BertModel
import torch

# 1. Carregar o tokenizador e o modelo BERT-base pré-treinado
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertModel.from_pretrained('bert-base-uncased')

# 2. Frase de exemplo
sentence = "I am a robot"

# 3. Tokenizar a frase (retorna IDs e tokens)
tokens = tokenizer.tokenize(sentence)
token_ids = tokenizer.convert_tokens_to_ids(tokens)

# 4. Obtendo os embedding contextualizados
input_ids = torch.tensor([token_ids])
with torch.no_grad():
    outputs = model(input_ids)
embeddings_contextualizados = outputs.last_hidden_state

print("Tokens:", tokens)
print("Token IDs:", token_ids)
print("Vocab size:", tokenizer.vocab_size)
print("Dimensão embeddings contextuais:", embeddings_contextualizados.shape)
print("Embeddings contextuais:\n",embeddings_contextualizados)
```

Código-fonte 14 - Entendendo os vetores Transformer com o BERT
Fonte: Elaborado pelo autor (2025)

O modelo `bert-base-uncased` é chamado de “BERT cru”, por ter apenas o corpo do transformador que gera embeddings contextuais para cada token, sem incluir uma camada final (“cabeça”) específica para uma tarefa. Isso significa que ele não produz previsões prontas, mas fornece representações ricas do texto que podem ser usadas como entrada para diferentes aplicações. Para transformá-lo em um classificador de sentimentos, um sistema de perguntas e respostas ou um modelo de reconhecimento de entidades, por exemplo, é necessário acrescentar manualmente uma camada de saída adequada — como uma rede totalmente conectada (linear) seguida de uma função de ativação — e treinar ou ajustar essa parte para a tarefa desejada. Entenderemos mais sobre isso na próxima seção.

Nota: o modelo Transformer pode ser entendido como uma RNA especial que permite construir embedding contextuais poderosos para representar texto. Posteriormente, esses embeddings são usados para diferentes tarefas de NLP.

4 TRANSFERÊNCIA DE APRENDIZADO E DOWNSTREAM TASKS

4.1 As famílias de modelos Transformer

Os modelos baseados em Transformer podem ser agrupados em três famílias principais, de acordo com a forma como utilizam os blocos de **encoder** e **decoder** (repetimos a arquitetura Transformer na figura assinalando os blocos de codificação e decodificação). Os modelos **encoder-only**, como o **BERT**, são voltados para compreender e representar o significado do texto, processando toda a sequência de forma bidirecional. Eles são ideais para tarefas de compreensão, classificação e extração de informação. Já os modelos **decoder-only**, como o **GPT**, usam apenas a parte decodificadora, gerando texto de forma autorregressiva (prevendo o próximo token a partir do histórico anterior), o que os torna particularmente eficazes em tarefas de geração de linguagem natural. Já os modelos **encoder-decoder**, como o **T5** ou o **BART**, combinam os dois blocos: o encoder lê e codifica a entrada em uma representação contextual, e o decoder gera uma saída condicionada a essa codificação, o que é especialmente útil em tarefas de transformação de texto, como tradução automática, sumarização e reformulação.

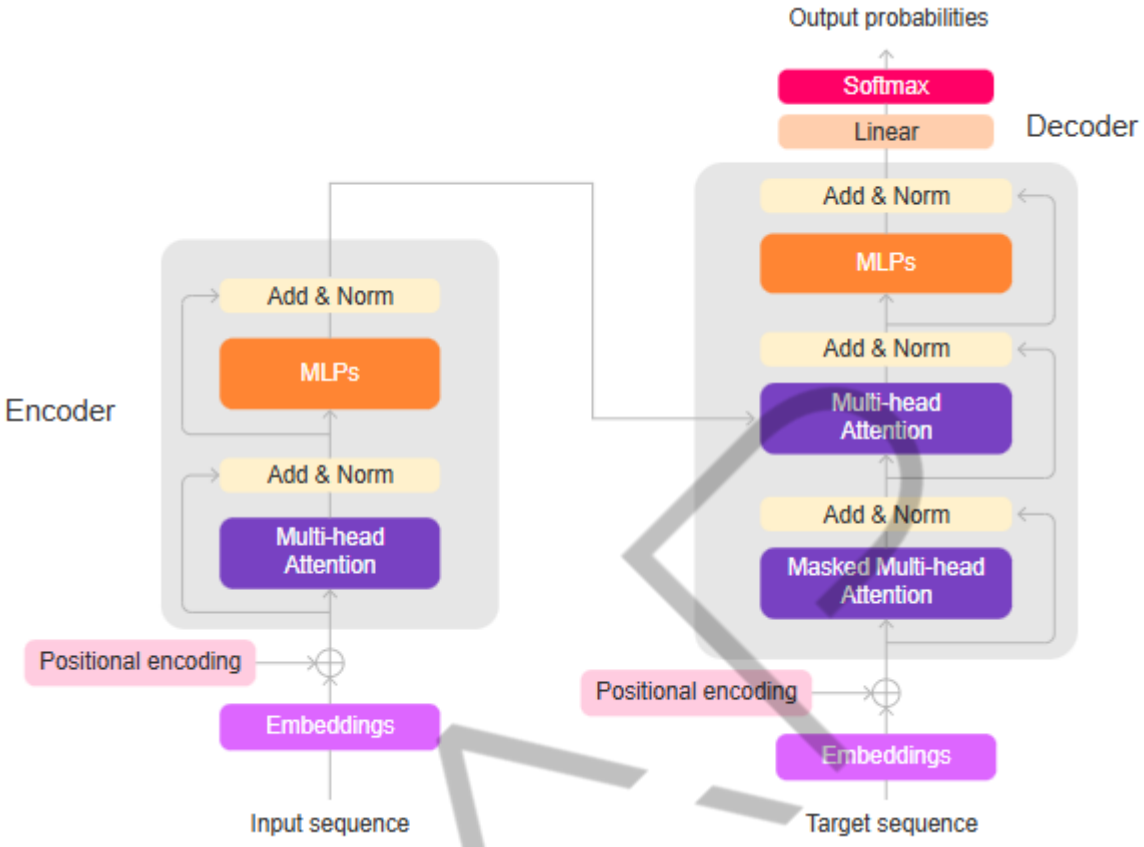

Figura 18 - Os blocos encoder e decoder da arquitetura Transformer
Fonte: Google imagens (2025), adaptado por FIAP (2025)

A evolução dos modelos baseados em Transformer tem sido marcada por um rápido aumento de escala, diversidade arquitetural e especialização em tarefas. Inicialmente, modelos como o **BERT** e o **GPT** exploraram, de forma separada, os paradigmas de compreensão e geração de linguagem, estabelecendo as bases para o uso de **pré-treinamento em grandes corpora** seguido de ajuste fino. Com o tempo, surgiram variantes mais robustas e flexíveis, como **T5** e **BART**, que unificam tarefas de entrada e saída no formato “texto para texto”. A tendência recente tem sido ampliar o tamanho dos modelos (bilhões ou até trilhões de parâmetros), incorporar **aprendizado multimodal** (texto, imagem, áudio e vídeo), reduzir custos de inferência com técnicas de compressão e afinar o treinamento com feedback humano (**RLHF**) para alinhar melhor o comportamento às expectativas dos usuários. Esse processo levou dos primeiros Transformers voltados a tarefas específicas para **modelos fundacionais** capazes de atuar como sistemas gerais de linguagem e raciocínio.

Na próxima figura, apresentamos uma árvore com a evolução dos principais modelos que se basearam no Transformer nos últimos anos:

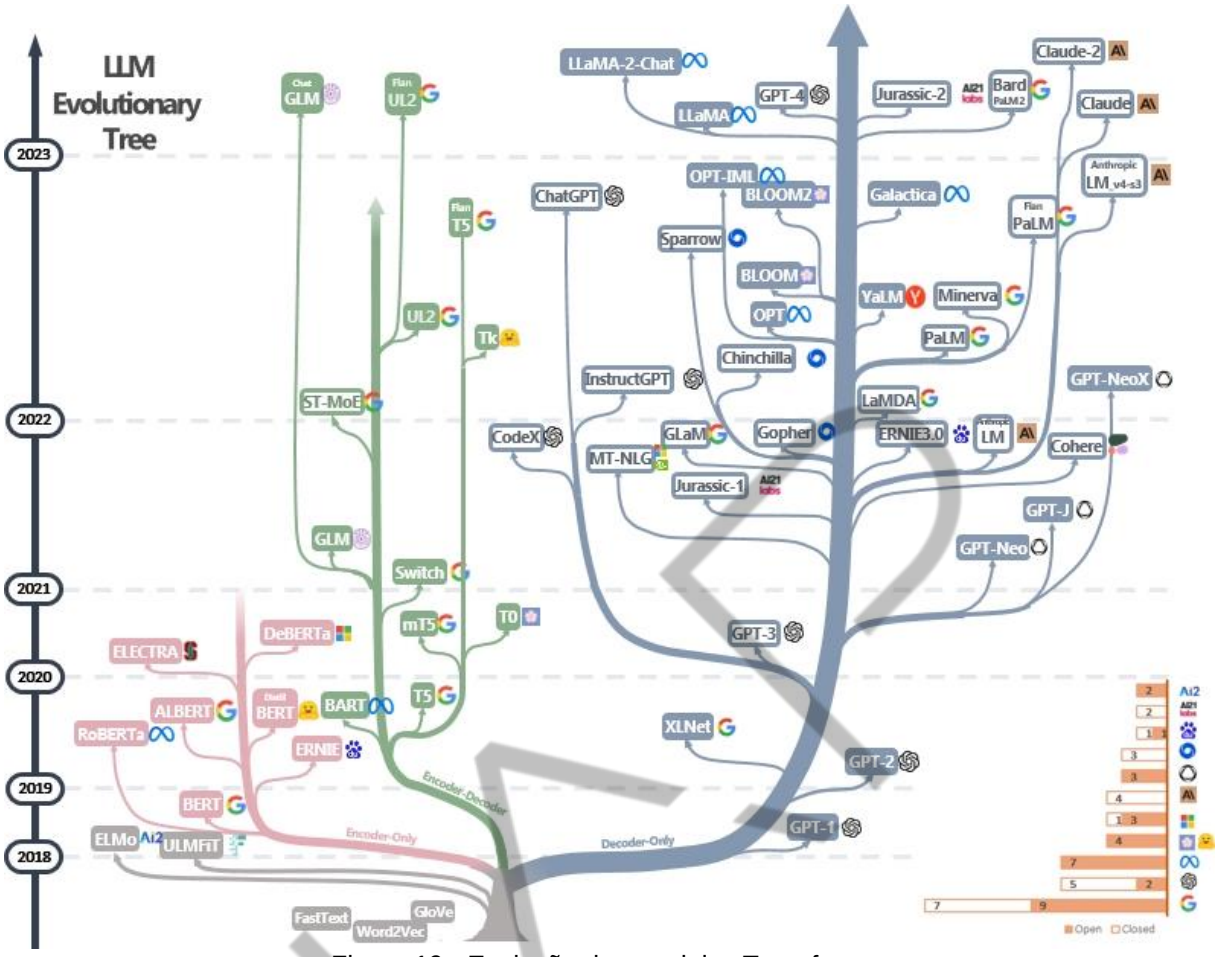

Figura 19 - Evolução dos modelos Transformer
Fonte: Google imagens (2025)

4.2 Transferência de aprendizado e fine-tuning

A transferência de aprendizado (*transfer learning*) é o padrão que tornou os Transformers tão poderosos: em vez de treinar um modelo do zero para cada tarefa, primeiro treina-se um modelo pré-treinado em grandes corpora não rotulados para aprender representações linguísticas gerais (sintaxe, semântica, fatos, padrões de coocorrência). Essas representações são então transferidas para tarefas específicas, servindo como ponto de partida que reduz drasticamente a necessidade de dados rotulados, acelera convergência e geralmente melhora a generalização. Nos Transformers iniciais, isso acontece da seguinte forma: o BERT (encoder-only) aprende com Masked Language Modeling (e originalmente Next Sentence Prediction), GPT (decoder-only) aprende autoregressivamente a prever o próximo token, e BART/T5 (encoder-decoder) aprendem a restaurar texto corrompido — cada objetivo cria tipos diferentes de sinais que são úteis para tarefas distintas.

Fine-tuning (ajuste fino) é a etapa prática da transferência: pega-se o modelo pré-treinado e o "ajusta" em um conjunto rotulado da tarefa-alvo, atualizando (parte ou todo) o conjunto de pesos para otimizar a métrica da tarefa. Existem várias formas de fazer esse "ajuste fino", mas todas elas focam na "downstream task". Por exemplo, se queremos fazer análise de sentimentos, podemos usar o modelo `bert-base-uncased` e adicionar uma camada final na cabeça do modelo, que será responsável pela classificação.

É exatamente isso que estamos realizando no código-fonte:

```
from transformers import BertTokenizer, BertModel
import torch
import torch.nn as nn

# 1. Carregar tokenizador e modelo BERT-base
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
bert = BertModel.from_pretrained('bert-base-uncased')

# 2. Criar uma "cabeça" de classificação (ex: 2 classes: positivo/negativo)
class SentimentClassifier(nn.Module):
    def __init__(self, bert, num_classes=2):
        super(SentimentClassifier, self).__init__()
        self.bert = bert
        self.classifier = nn.Linear(bert.config.hidden_size, num_classes)

    def forward(self, input_ids, attention_mask=None, token_type_ids=None):
        outputs = self.bert(input_ids=input_ids, attention_mask=attention_mask, token_type_ids=token_type_ids)
        cls_embedding = outputs.last_hidden_state[:, 0, :]
        logits = self.classifier(cls_embedding)
        return logits


model = SentimentClassifier(bert)

# 3. Tokenizar frase
sentence = "You are a bad person"
#sentence = "You are a good person"
#sentence = "A nice cat"
#sentence = "They are so evil"
inputs = tokenizer(sentence, return_tensors="pt")

# 4. Passar pelo modelo
with torch.no_grad():
```

```
    logits = model(**inputs)

# 5. Calcular probabilidades
probs = torch.softmax(logits, dim=1)
print("Frase:", sentence)
print("Probabilidades:", probs)
print("Classe prevista:", torch.argmax(probs).item())
```

Código-fonte 15 - Criando uma cabeça de downstream task com o BERT
Fonte: Elaborado pelo autor (2025)

Nesse exemplo, embora a cabeça de classificação tenha sido adicionada, ainda não realizamos o treinamento dela com dados rotulados — o modelo está apenas fazendo uma inferência direta com pesos pré-treinados. Para um fine-tuning completo, seria necessário usar um conjunto de dados rotulados específicos da tarefa (por exemplo, frases com suas classes de sentimento) para ajustar os pesos dessa cabeça e possivelmente do próprio modelo base, minimizando a função de perda da tarefa. Assim, o modelo aprende a associar as representações gerais já adquiridas no pré-treinamento às decisões específicas da tarefa, aprimorando a precisão e a robustez.

Existem várias abordagens de fine-tuning possíveis:

- **Freezing:** consiste em manter fixos os pesos do modelo base e treinar apenas a cabeça adicionada. Isso é útil quando há poucos dados rotulados ou recursos limitados, mas pode limitar o desempenho.
- **Adapters:** pequenas redes adicionais inseridas entre camadas do modelo, que são treinadas enquanto o modelo base permanece congelado. Permitem adaptar modelos grandes a várias tarefas sem precisar armazenar cópias completas para cada uma.
- **Prompt tuning e prefix tuning:** técnicas que ajustam ou adicionam tokens especiais no início da entrada para guiar o modelo a responder conforme a tarefa, sem modificar diretamente os pesos principais.
- **Continued pretraining:** consiste em retreinar parcialmente o modelo pré-treinado com texto não rotulado do domínio-alvo antes do fine-tuning, para alinhar melhor as representações às características específicas do novo domínio.

Cada uma dessas estratégias tem vantagens e compromissos (*trade-offs*) em termos de custo computacional, quantidade de dados necessários e desempenho final, e são escolhidas conforme o contexto e a aplicação desejada.

4.3 Testando modelos de outras famílias

No exemplo do próximo código-fonte, usamos o GPT-2, um modelo decoder-only. Esse tipo de modelo recebe um prompt e gera texto continuando a sequência de forma autorregressiva, prevendo um token após o outro com base no contexto gerado até o momento. Diferentemente de modelos focados em classificação como o BERT, o GPT é projetado para tarefas de geração de linguagem natural, como escrita criativa, diálogo e completamento de texto.

```
from transformers import GPT2Tokenizer, GPT2LMHeadModel
import torch

tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
model = GPT2LMHeadModel.from_pretrained('gpt2')

# Texto inicial (prompt) para o modelo continuar
prompt = "Once upon a time, in a distant land, there was a"

# Tokenizar o prompt
inputs = tokenizer(prompt, return_tensors="pt")

# Gerar continuação do texto (autorregeressivamente)
outputs = model.generate(
    inputs['input_ids'],
    max_length=50,
    num_return_sequences=1,
    no_repeat_ngram_size=2,
    do_sample=True,
    top_k=50,
    top_p=0.95,
    temperature=0.7
)

# Decodificar e mostrar o texto gerado
generated_text = tokenizer.decode(outputs[0],
skip_special_tokens=True)
print("Texto gerado:", generated_text)
```

Código-fonte 16 - Usando o GPT2 para gerar texto
Fonte: Elaborado pelo autor (2025)

A saída do código-fonte é:

```
Texto gerado: Once upon a time, in a distant land, there
was a man called Sir Isaac Newton who was born in the year 1470,
and whose name is known to have been Newton, a mathematician,
scientist, physician, philosopher, surgeon, mathematician.
```

Perceba que, mesmo sendo **decoder-only**, o GPT processa o texto de entrada de forma **autorregressiva**, ou seja, ele lê o texto token por token da esquerda para a direita, usando cada token anterior como contexto para prever o próximo. Na prática, isso significa que o modelo não “vê” a frase inteira de uma vez, como o BERT (que é bidirecional), mas constrói uma representação incremental à medida que os tokens são processados sequencialmente.

Durante a geração, o GPT recebe um prompt inicial (texto de entrada) e prevê o próximo token com base nele, depois adiciona esse token à sequência e prevê o seguinte, e assim por diante — tudo isso usando o mesmo bloco decodificador. Internamente, o modelo utiliza máscaras de atenção para garantir que, ao prever o token na posição *t*, só possa “ver” os tokens anteriores (de 1 até *t-1*), evitando informações futuras.

Por isso, apesar de ser só decoder, o GPT consegue incorporar o contexto passado para gerar texto coerente e fluido, porém sua capacidade é limitada a contextos sequenciais da esquerda para a direita, diferente dos modelos bidirecionais como o BERT, que consideram contexto anterior e posterior simultaneamente.

O BART (Bidirectional and Auto-Regressive Transformers) é um modelo da família **encoder-decoder** que combina os benefícios dos modelos bidirecionais, como o BERT, e dos modelos autorregressivos, como o GPT. Ele é treinado para reconstruir texto corrompido por meio de um processo de denoising, no qual o encoder lê a entrada danificada e o decoder gera o texto original, token por token. Essa arquitetura torna o BART especialmente eficaz para tarefas de geração de texto condicionada, como sumarização, tradução e correção automática, em que é necessário entender o contexto completo e gerar uma sequência coerente e fluida. Diferentemente do BERT, que produz embeddings para tarefas de classificação, o BART gera texto sequencialmente, podendo ser ajustado para diferentes formatos de saída.

```
from transformers import BartTokenizer, BartForConditionalGeneration

# Carregar tokenizer e modelo pré-treinado BART
tokenizer = BartTokenizer.from_pretrained('facebook/bart-large-cnn')
model = BartForConditionalGeneration.from_pretrained('facebook/bart-large-cnn')

# Texto de entrada para sumarização
text = '''
Just had the best day ever! Went hiking with some amazing friends up in the mountains. The weather was perfect — sunny but not too hot. We saw breathtaking views and even spotted some wildlife along the trail. Finished the day with a delicious picnic by the lake. Feeling so grateful for moments like these and for friends who make life so special! Can’t wait for the next adventure. 🏔️🌞👟
'''

# Tokenizar entrada
inputs = tokenizer(text, return_tensors="pt", max_length=1024, truncation=True)

# Gerar sumarização (texto gerado token a token pelo decoder)
summary_ids = model.generate(inputs['input_ids'], max_length=50, num_beams=4, early_stopping=True)

# Decodificar e imprimir o resumo
summary = tokenizer.decode(summary_ids[0], skip_special_tokens=True)
print("Resumo:", summary)
print(len(text), len(summary))
```

Código-fonte 17 - Usando o BART para sumarizar texto
Fonte: Elaborado pelo autor (2025)

Nesse exemplo, o modelo BART utiliza sua arquitetura encoder-decoder para compreender o texto completo na entrada (encoder) e gerar um resumo coerente e fluido na saída (decoder), mostrando a diferença fundamental em relação ao BERT, que foca na geração de representações para tarefas de classificação, e ao GPT, que é puramente decoder e autorregressivo para geração de texto a partir de prompts.

REFERÊNCIAS

DEVLIN, Jacob *et al*., BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. **ArXiv**, 24 de maio de 2019. Disponível em: <https://arxiv.org/abs/1810.04805>. Acesso em: 03 set. 2025.

JURAFSKY, Dan; MARTIN, James. Speech and Language Processing. Edição 3ª. **Stanford**, 24 de agosto de 2025. Disponível em: <https://web.stanford.edu/~jurafsky/slp3/>. Acesso em: 03 set. 2025.

SAEED, Mehreen, A Gentle Introduction to Positional Encoding in Transformer Models. **Cs.bu**, 06 de janeiro de 2023. Disponível em: <https://www.cs.bu.edu/fac/snyder/cs505/PositionalEncodings.pdf>. Acesso em: 03 set. 2025.

VASWANI, Ashish *et al*. Attention Is All You Need. **ArXiv**, 02 de agosto de 2023. Disponível em: <https://arxiv.org/abs/1706.03762>. Acesso em: 03 set. 2025.

YANG, Jinggeng. The Practical Guides for Large Language Models. **GitHub**, 2023. Disponível em: <https://github.com/Mooler0410/LLMsPracticalGuide>. Acesso em: 03 set. 2025.