Finetuning LLMs para Classificação

Tópicos em Ciência de Dados

Prof. Dr. Denis Mayr Lima Martins

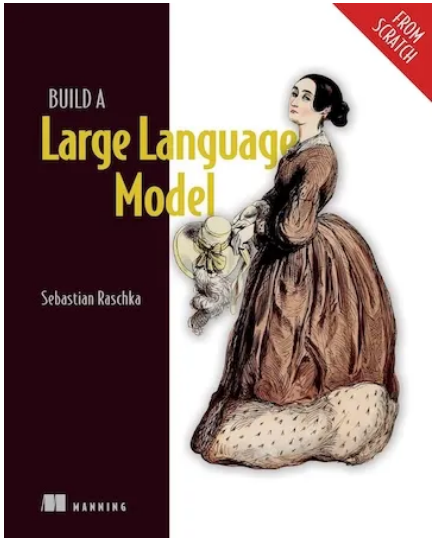
Pontifícia Universidade Católica de Campinas

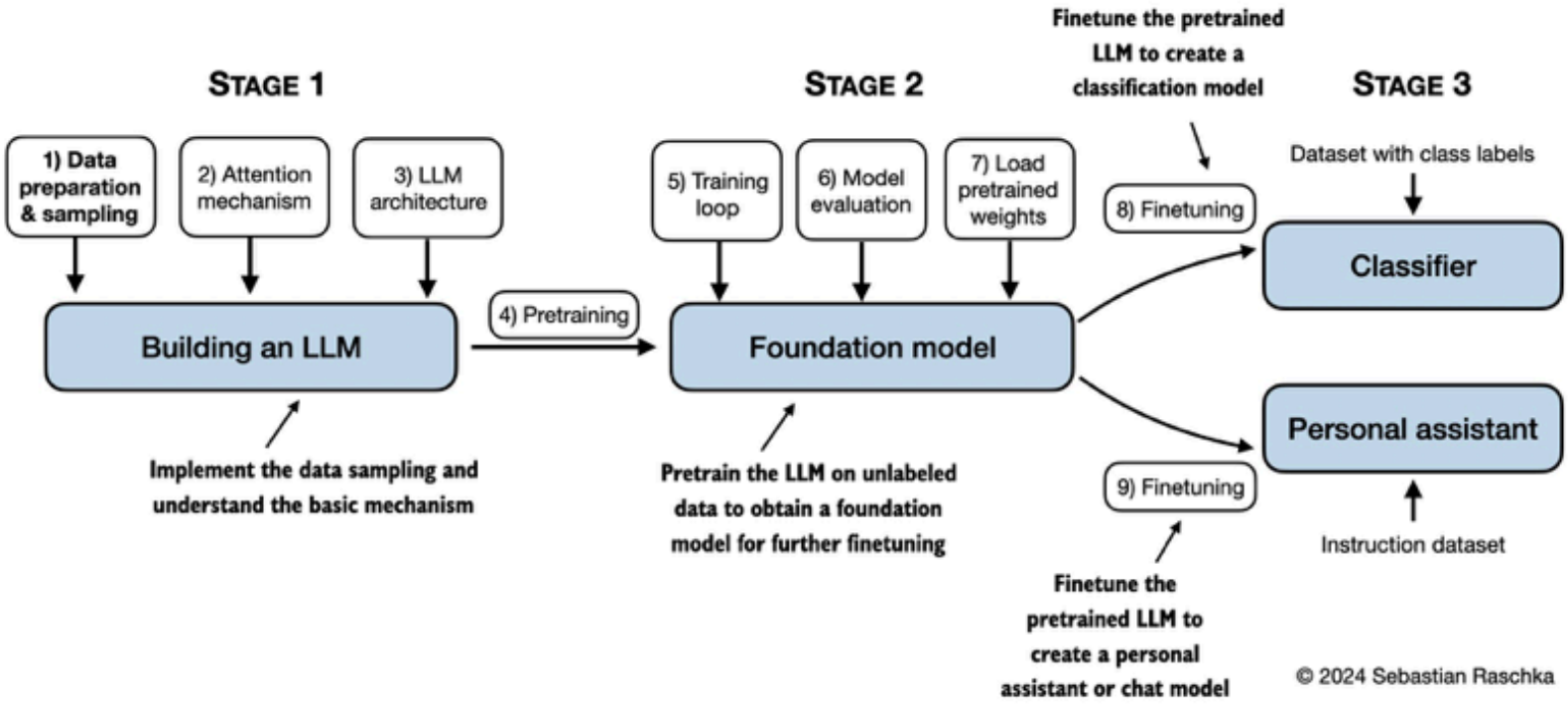
Objetivos de Aprendizagem

- **Definir e Diferenciar:** Conceituar *fine-tuning* e distinguir claramente este processo da fase inicial de *pre-training* (pré-treinamento) de Large Language Models (LLMs).
- **Classificar Metodologias:** Identificar e diferenciar as diversas metodologias de *fine-tuning*, incluindo as abordagens supervisionadas (SFT), não supervisionadas e baseadas em instruções (*Instruction Fine-Tuning*).
- Explicar o papel do *fine-tuning* no *Transfer Learning*, reconhecendo os benefícios de **redução de requisitos de dados** e **melhoria na generalização** para tarefas específicas.
- **Descrever o Pipeline:** Descrever as etapas essenciais do pipeline de *fine-tuning* para LLMs, que abrangem desde a preparação inicial do *dataset* até o monitoramento contínuo.

Baseado no Livro Build a Large Language Model From Scratch de Sebastian Raschka

Code repository: https://github.com/rasbt/LLMs-from-scratch





Visão Geral da construção de um LLM. Fonte: Raschka.

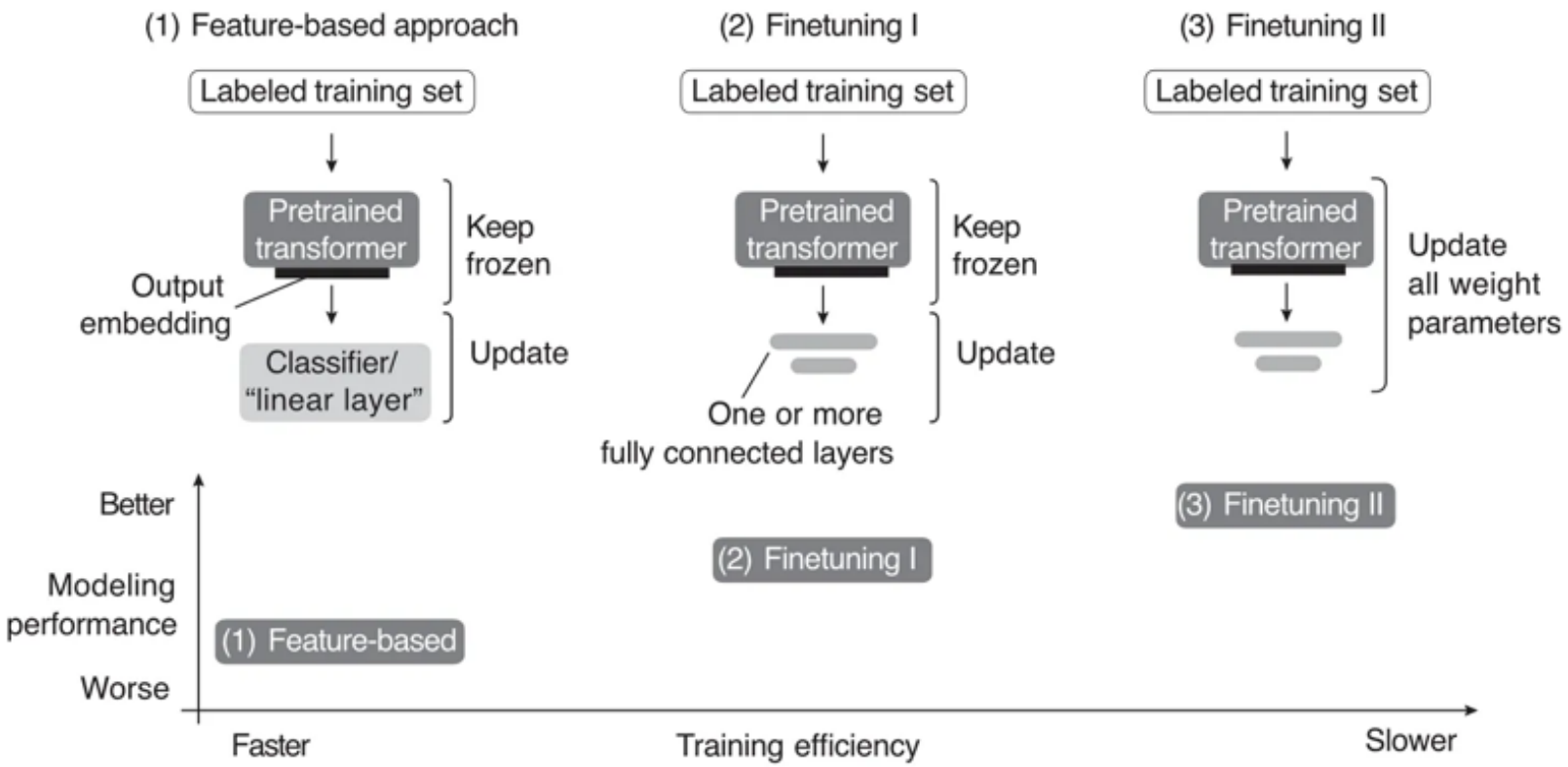
O Conceito de Fine-Tuning

- **Definição:** É o processo de utilizar um modelo pré-treinado como base e treiná-lo adicionalmente em um *dataset* menor e específico de um domínio ou tarefa.
- **Objetivo:** Adaptar o modelo ao novo contexto, aprimorando o desempenho em aplicações especializadas, como tradução de linguagem, análise de sentimento ou sumarização.
- **Vantagem:** O *fine-tuning* se baseia no conhecimento pré-existente do modelo, o que reduz substancialmente os requisitos computacionais e de dados em comparação com o treinamento do modelo do zero (*pre-training*).

In [1]:
```
from IPython.display import YouTubeVideo
YouTubeVideo("2QRlvKSzyVw", width=600, height=350)
```

Out[1]:

Três abordagens de uso de LLMs pré-treinados



Abordagens de uso de LLMs. Veja código aqui. Fonte: Raschka.

In [ ]:
```
emotions = load_dataset("emotion")
```

In [ ]:
```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")
print("Tokenizer input max length:", tokenizer.model_max_length)
print("Tokenizer vocabulary size:", tokenizer.vocab_size)
```

In [ ]:
```
def tokenize(batch):
    return tokenizer(batch["text"], padding='max_length', truncation=Tru
```

In [ ]:
```
emotions_tokenized = emotions.map(tokenize, batched=True, batch_size=10(
```

In [ ]:
```
from transformers import AutoModel

model = AutoModel.from_pretrained("distilbert-base-uncased")
model.to(device);
```

In [ ]:
```
emotions_tokenized.set_format("torch", columns=["input_ids", "attention_
```

In [ ]:

```
def extract_hidden_states(batch):
    inputs = {k:v.to(device) for k,v in batch.items()
              if k in tokenizer.model_input_names}
    with torch.no_grad():
        last_hidden_state = model(**inputs).last_hidden_state
    return {"features": last_hidden_state[:,0].cpu().numpy()}

emotions_features = emotions_tokenized.map(extract_hidden_states, batche
```

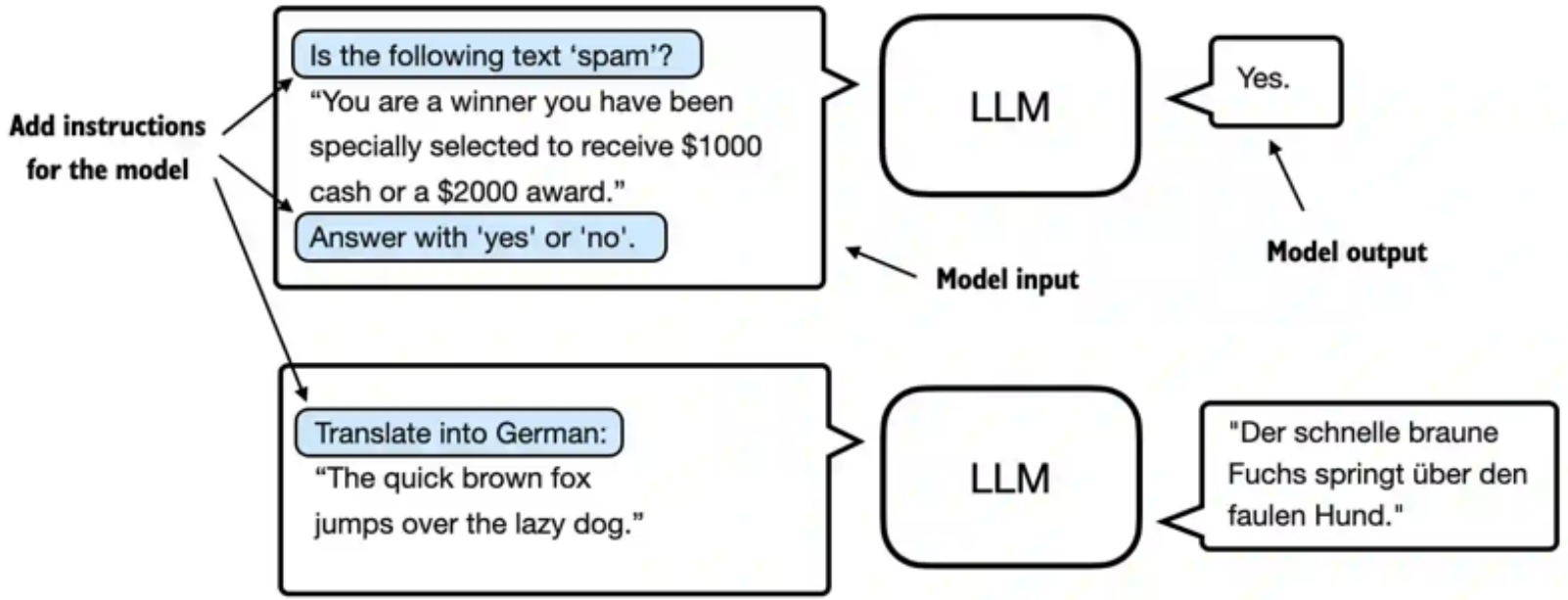
In [ ]:

```
X_train = np.array(emotions_features["train"]["features"])
y_train = np.array(emotions_features["train"]["label"])

X_val = np.array(emotions_features["validation"]["features"])
y_val = np.array(emotions_features["validation"]["label"])

X_test = np.array(emotions_features["test"]["features"])
y_test = np.array(emotions_features["test"]["label"])
```

In [ ]:

```
from sklearn.ensemble import RandomForestClassifier

clf = RandomForestClassifier()
clf.fit(X_train, y_train)

print("Training accuracy", clf.score(X_train, y_train))
print("Validation accuracy", clf.score(X_val, y_val))
print("Test accuracy", clf.score(X_test, y_test))
```

Finetuning para Classificação ou para Instrução



Categorias de Finetuning de LLMs. Fonte: Raschka.

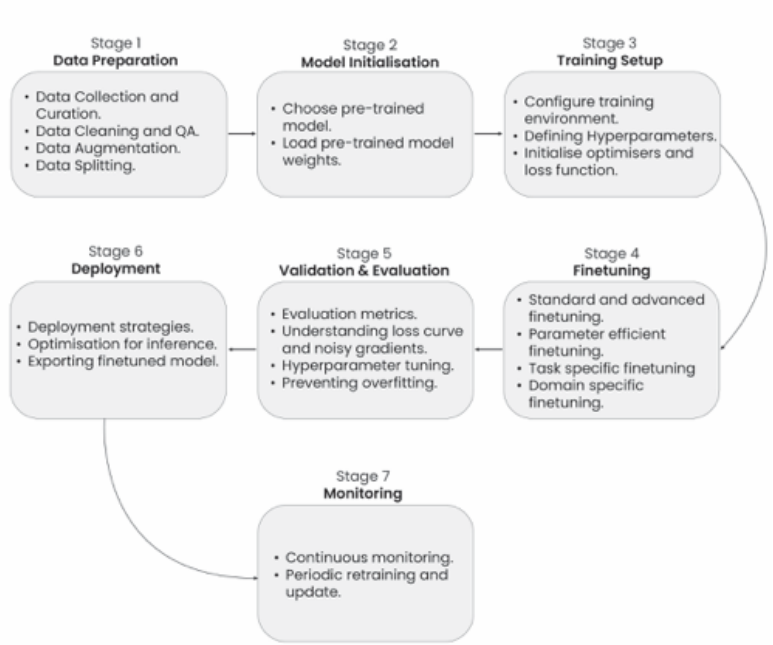
Tipos de *Fine-Tuning* de LLMs

O *fine-tuning* pode ser categorizado pela natureza da supervisão:

Tipo	Característica	Aplicação
Supervisionado	Utiliza dados rotulados adaptados à tarefa alvo.	Classificação de texto, tarefas de Geração de Texto específicas.
Não Supervisionado	Expõe o LLM a um grande *corpus* de texto não rotulado do domínio alvo.	Refino da compreensão linguística em novos domínios (ex: área jurídica ou médica).
Por Instrução	Utiliza instruções em linguagem natural (*Prompt Engineering*) para treinar o modelo a seguir comandos.	Criação de assistentes conversacionais especializados.

Pipeline Geral de Finetuning de LLMs

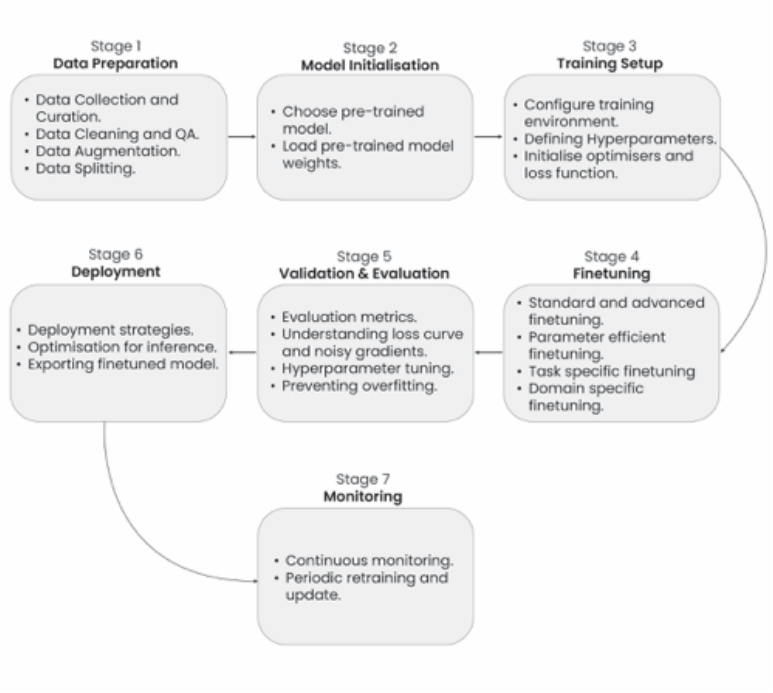
- **Estágio 1: Preparação do Conjunto de Dados**
  - Limpeza, formatação e divisão do conjunto de dados para corresponder à tarefa alvo.
- **Estágio 2: Inicialização do Modelo**
  - Consiste em configurar os parâmetros iniciais e as configurações do LLM.
- **Estágio 3: Configuração do Ambiente de Treinamento**
  - Configuração de *hardware* e *software*, hiperparâmetros e arquitetura do modelo.
- **Estágio 4: Ajuste Fino Parcial ou Completo**
  - É o processo de atualização dos parâmetros do LLM utilizando o *dataset* específico da tarefa.



Seven Stage Fine-Tuning Pipeline for LLM.
Fonte: Parthasarathy et al.

Pipeline Geral de Finetuning de LLMs (cont.)

- **Estágio 5: Avaliação e Validação**
  - Métricas como **entropia cruzada** e o monitoramento de curvas de perda (*loss curves*).
- **Estágio 6: Implantação**
  - Execução eficiente em plataformas de *hardware* ou *software* designadas e a implementação de medidas de segurança e sistemas de monitoramento.
- **Estágio 7: Monitoramento e Manutenção**
  - Rastreamento contínuo do desempenho, atualização do modelo conforme necessário para se adaptar a novos dados ou requisitos em constante mudança.



Seven Stage Fine-Tuning Pipeline for LLM.
Fonte: Parthasarathy et al.

Desafios de Escalabilidade e Recursos

O *fine-tuning* de LLMs apresenta barreiras significativas:

- **Requisitos de Memória:** O *fine-tuning* exige memória substancialmente maior do que a inferência. Por exemplo, o LLaMA 2 (7B parâmetros) em FP32 exige aproximadamente **112 GB de VRAM** para o *fine-tuning*, limitando a acessibilidade para a maioria dos hardwares de consumo.
- **Otimização de Recursos:** A implantação eficiente requer técnicas como *Quantized LLMs* (quantização de modelos), que representam parâmetros com menos bits (ex: 8-bit ou 4-bit) para reduzir o tamanho do modelo e otimizar a inferência.

Considerações Éticas

O *fine-tuning* pode exacerbar questões éticas:

- **Vieses (*Bias*) e Justiça (*Fairness*):** *Datasets* de *fine-tuning* podem conter vieses que são transferidos e amplificados no modelo. É crucial usar dados diversos e representativos.
- **Privacidade:** O uso de dados sensíveis exige técnicas de preservação, como a **Privacidade Diferencial** e o **Aprendizado Federado** (*Federated Learning*).
- **Responsabilidade e Transparência:** Dada a capacidade do *fine-tuning* de alterar o comportamento do LLM, é fundamental documentar detalhadamente o processo, o *dataset* e o impacto, utilizando *frameworks* como *Model Cards* ou *AI FactSheets*.

Demo: Finetuning de um modelo GPT-2

- Tarefa: Classificação de Spam.
- Dataset: SMS Spam Collection (público)

Preparando o dataset

In [ ]:
```
import urllib.request
import zipfile
import os
from pathlib import Path

url = "https://archive.ics.uci.edu/static/public/228/sms+spam+collectio
zip_path = "sms_spam_collection.zip"
extracted_path = "sms_spam_collection"
data_file_path = Path(extracted_path) / "SMSSpamCollection.tsv"
```

In [ ]:

```
def download_and_unzip_spam_data(url, zip_path, extracted_path, data_fi
    if data_file_path.exists():
        print(f"{data_file_path} already exists. Skipping download and e
        return

    # Download
    with urllib.request.urlopen(url) as response:
        with open(zip_path, "wb") as out_file:
            out_file.write(response.read())

    # Unzip
    with zipfile.ZipFile(zip_path, "r") as zip_ref:
        zip_ref.extractall(extracted_path)

    # Extensão .tsv
    original_file_path = Path(extracted_path) / "SMSSpamCollection"
    os.rename(original_file_path, data_file_path)
    print(f"File downloaded and saved as {data_file_path}")

download_and_unzip_spam_data(url, zip_path, extracted_path, data_file_pa
```

```
File downloaded and saved as sms_spam_collection/SMSSpamCollecti
on.tsv
```

In [ ]:

```
import pandas as pd

df = pd.read_csv(data_file_path, sep="\t", header=None, names=["Label",
df.head()
```

Out[ ]:

	Label	Text
0	ham	Go until jurong point, crazy.. Available only ...
1	ham	Ok lar... Joking wif u oni...
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...
3	ham	U dun say so early hor... U c already then say...
4	ham	Nah I don't think he goes to usf, he lives aro...

In [ ]:
```
print(df["Label"].value_counts())
```

```
Label
ham     4825
spam     747
Name: count, dtype: int64
```

In [ ]:

```
def create_balanced_dataset(df):
    num_spam = df[df["Label"] == "spam"].shape[0]

    # Seleciona amostras não-spam
    ham_subset = df[df["Label"] == "ham"].sample(num_spam, random_state=

    # Combina dados das duas classes
    balanced_df = pd.concat([ham_subset, df[df["Label"] == "spam"]])

    return balanced_df

balanced_df = create_balanced_dataset(df)
print(balanced_df["Label"].value_counts())
```

```
Label
ham     747
spam    747
Name: count, dtype: int64
```

In [ ]:
```
balanced_df["Label"] = balanced_df["Label"].map({"ham": 0, "spam": 1})
```

In [ ]:

```
def random_split(df, train_frac, validation_frac):
    # Embaralha o DataFrame
    df = df.sample(frac=1, random_state=123).reset_index(drop=True) # fr

    # Calcula índices de treino e validação
    train_end = int(len(df) * train_frac)
    validation_end = train_end + int(len(df) * validation_frac)

    # Divide os dados
    train_df = df[:train_end]
    validation_df = df[train_end:validation_end] # to get an estimate du
    test_df = df[validation_end:]

    return train_df, validation_df, test_df

train_df, validation_df, test_df = random_split(balanced_df, 0.7, 0.1)

train_df.to_csv("train.csv", index=None)
validation_df.to_csv("validation.csv", index=None)
test_df.to_csv("test.csv", index=None)
```

Criando Data Loaders

In [ ]:
```
import tiktoken

tokenizer = tiktoken.get_encoding("gpt2")
print(tokenizer.encode("<|endoftext|>", allowed_special={"<|endoftext|>"
```

```
[50256]
```

In [ ]:

```
print(f"{len(train_loader)} training batches")
print(f"{len(val_loader)} validation batches")
print(f"{len(test_loader)} test batches")
```

```
130 training batches
19 validation batches
38 test batches
```

Inicializando o Modelo com Parâmetros Pré-treinados

In [ ]:

```
CHOOSE_MODEL = "gpt2-small (124M)"
INPUT_PROMPT = "Every effort moves"

BASE_CONFIG = {
    "vocab_size": 50257,     # Vocabulary size
    "context_length": 1024,  # Context length
    "drop_rate": 0.0,        # Dropout rate
    "qkv_bias": True         # Query-key-value bias
}

model_configs = {
    "gpt2-small (124M)": {"emb_dim": 768, "n_layers": 12, "n_heads": 12
    "gpt2-medium (355M)": {"emb_dim": 1024, "n_layers": 24, "n_heads": 1
    "gpt2-large (774M)": {"emb_dim": 1280, "n_layers": 36, "n_heads": 20
    "gpt2-xl (1558M)": {"emb_dim": 1600, "n_layers": 48, "n_heads": 25},
}

BASE_CONFIG.update(model_configs[CHOOSE_MODEL])
```

In [ ]:

```
from llmdefinitions import GPTModel, load_weights_into_gpt

model_size = CHOOSE_MODEL.split(" ")[-1].lstrip("(").rstrip(")")
settings, params = download_and_load_gpt2(model_size=model_size, models_

model = GPTModel(BASE_CONFIG)
load_weights_into_gpt(model, params)
model.eval();
```

In [ ]:

```
from llmdefinitions import (
    generate_text_simple,
    text_to_token_ids,
    token_ids_to_text
)


text_1 = "Every effort moves you"

token_ids = generate_text_simple(
    model=model,
    idx=text_to_token_ids(text_1, tokenizer),
    max_new_tokens=15,
    context_size=BASE_CONFIG["context_length"]
)

print(token_ids_to_text(token_ids, tokenizer))
```

```
Every effort moves you forward.

The first step is to understand the importance of your work
```

In [ ]:

```
text_2 = (
    "Is the following text 'spam'? Answer with 'yes' or 'no':"
    " 'You are a winner you have been specially"
    " selected to receive $1000 cash or a $2000 award.'"
    " Answer with 'yes' or 'no'."
)

token_ids = generate_text_simple(
    model=model,
    idx=text_to_token_ids(text_2, tokenizer),
    max_new_tokens=23,
    context_size=BASE_CONFIG["context_length"]
)

print(token_ids_to_text(token_ids, tokenizer))
```

```
Is the following text 'spam'? Answer with 'yes' or 'no': 'You ar
e a winner you have been specially selected to receive $1000 cas
h or a $2000 award.' Answer with 'yes' or 'no'. Answer with 'ye
s' or 'no'. Answer with 'yes' or 'no'. Answer with 'yes'
```

Adicionando Camada de Classificação (Classification Head)

Substituiremos a camada de saída original – que mapeava a representação oculta para um vocabulário de 50 257 tokens – por uma camada de saída mais compacta que projeta esses mesmos 768 unidades ocultas em apenas duas classes: 0 (“não spam”) e 1 (“spam”).

In [ ]:
```
# Congelamos o modelo, isto é, tornamos todas as camadas não treináveis.
for param in model.parameters():
    param.requires_grad = False
```

Em seguida, substituímos a camada de saída (`model.out_head`), que originalmente mapeia as entradas da camada para 50 257 dimensões (o tamanho do vocabulário). Como o modelo será fine-tuned para classificação binária, podemos trocar a camada de saída conforme mostrado abaixo; tal nova camada será treinável por padrão. Observe que utilizamos `BASE_CONFIG["emb_dim"]` (que equivale a 768 no modelo "gpt2-small (124M)") para manter o trecho de código mais genérico.

In [ ]:
```
torch.manual_seed(123)

num_classes = 2
model.out_head = torch.nn.Linear(in_features=BASE_CONFIG["emb_dim"], ou
```

Tecnicamente, basta treinar apenas a camada de saída. Contudo, o fine-tuning de camadas adicionais pode melhorar significativamente o desempenho preditivo do modelo. Portanto, também tornamos treináveis o último bloco transformador e o módulo final `LayerNorm` que conecta esse bloco ao layer de saída.

In [ ]:
```
for param in model.trf_blocks[-1].parameters():
    param.requires_grad = True

for param in model.final_norm.parameters():
    param.requires_grad = True
```

In [ ]:

```
inputs = tokenizer.encode("Do you have time")
inputs = torch.tensor(inputs).unsqueeze(0)
print("Inputs:", inputs)
print("Inputs dimensions:", inputs.shape) # shape: (batch_size, num_toke
```

```
Inputs: tensor([[5211,  345,  423,  640]])
Inputs dimensions: torch.Size([1, 4])
```

In [ ]:

```
with torch.no_grad():
    outputs = model(inputs)

print("Outputs:\n", outputs)
print("Outputs dimensions:", outputs.shape) # shape: (batch_size, num_t
```

```
Outputs:
 tensor([[[-1.5854,  0.9904],
          [-3.7235,  7.4548],
          [-2.2661,  6.6049],
          [-3.5983,  3.9902]]])
Outputs dimensions: torch.Size([1, 4, 2])
```

Ao discutir o mecanismo de atenção – que conecta cada token de entrada a todos os demais tokens de entrada – introduzimos a máscara causal de atenção utilizada em modelos do tipo GPT; tal máscara causal permite que um token atual atenda apenas às posições atuais e anteriores. Com base nesse mecanismo causal, o quarto (último) token contém a maior quantidade de informação dentre todos os tokens, pois é o único que incorpora dados sobre todos os demais tokens.

Assim, nosso foco recai especificamente sobre esse último token, que será fine-tuned para a tarefa de classificação de spam:

In [ ]:
```
print("Last output token:", outputs[:, -1, :])
```

```
Last output token: tensor([[-3.5983,  3.9902]])
```

Calculando Loss e Accuracy

Convertemos os valores de saída (logits) em escores de probabilidade por meio da função `softmax` e, em seguida, obtivemos a posição do índice correspondente ao maior valor de probabilidade utilizando a função `argmax`.

In [ ]:
```
probas = torch.softmax(outputs[:, -1, :], dim=-1)
label = torch.argmax(probas)
print("Class label:", label.item())
```

```
Class label: 1
```

O código retorna 1, o que indica que o modelo prevê que o texto de entrada seja “spam”. Observe que a aplicação da função `softmax` é opcional neste ponto, pois os valores de saída maiores correspondem aos maiores escores de probabilidade. Consequentemente, podemos simplificar o trecho de código conforme abaixo, omitindo a chamada ao `softmax`.

In [ ]:
```
logits = outputs[:, -1, :]
label = torch.argmax(logits) # Pula Softmax
print("Class label:", label.item())
```

```
Class label: 1
```

In [ ]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

torch.manual_seed(123)

train_accuracy = calc_accuracy_loader(train_loader, model, device, num_b
val_accuracy = calc_accuracy_loader(val_loader, model, device, num_batch
test_accuracy = calc_accuracy_loader(test_loader, model, device, num_ba

print(f"Training accuracy: {train_accuracy*100:.2f}%")
print(f"Validation accuracy: {val_accuracy*100:.2f}%")
print(f"Test accuracy: {test_accuracy*100:.2f}%")
```

```
Training accuracy: 46.25%
Validation accuracy: 45.00%
Test accuracy: 48.75%
```

A acurácia de classificação não é uma função diferenciável. Portanto, em vez disso, minimizamos a perda de entropia cruzada como um proxy para maximizar essa acurácia. Nos concentramos apenas na otimização do último token (`model(input_batch)[:, -1, :]`) ao invés de todos os tokens (`model(input_batch)`).

In [ ]:

```
with torch.no_grad():
    train_loss = calc_loss_loader(train_loader, model, device, num_batch
    val_loss = calc_loss_loader(val_loader, model, device, num_batches=5
    test_loss = calc_loss_loader(test_loader, model, device, num_batches

print(f"Training loss: {train_loss:.3f}")
print(f"Validation loss: {val_loss:.3f}")
print(f"Test loss: {test_loss:.3f}")
```

```
Training loss: 2.453
Validation loss: 2.583
Test loss: 2.322
```

Finetuning do Modelo usando Dados Rotulados

In [ ]:

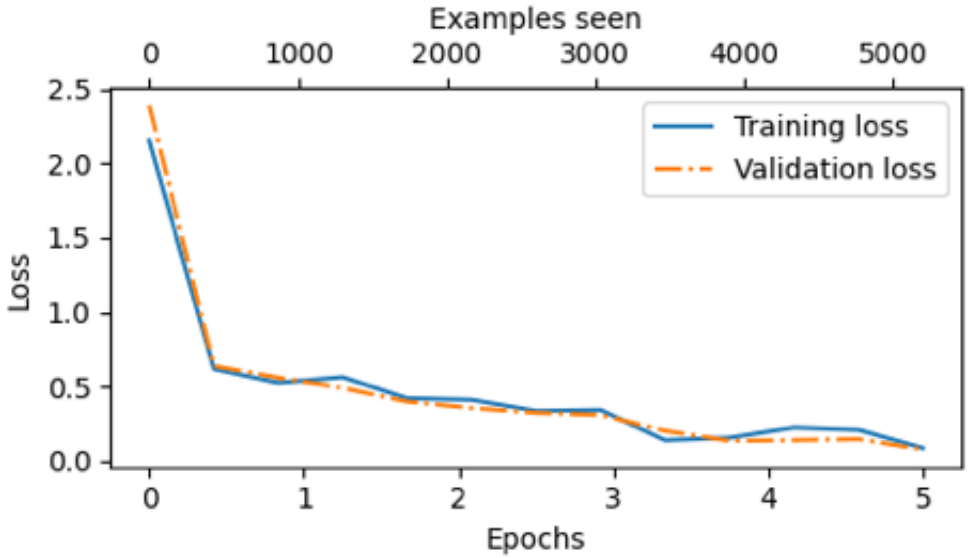
```
def evaluate_model(model, train_loader, val_loader, device, eval_iter):
    model.eval()
    with torch.no_grad():
        train_loss = calc_loss_loader(train_loader, model, device, num_
        val_loss = calc_loss_loader(val_loader, model, device, num_batc
    model.train()
    return train_loss, val_loss
```

In [ ]:

```
import time

start_time = time.time()
torch.manual_seed(123)
optimizer = torch.optim.AdamW(model.parameters(), lr=5e-5, weight_decay=

num_epochs = 5
train_losses, val_losses, train_accs, val_accs, examples_seen = train_c
    model, train_loader, val_loader, optimizer, device,
    num_epochs=num_epochs, eval_freq=50, eval_iter=5,
    tokenizer=tokenizer
)

end_time = time.time()
execution_time_minutes = (end_time - start_time) / 60
print(f"Training completed in {execution_time_minutes:.2f} minutes.")
```

```
Ep 1 (Step 000000): Train loss 2.153, Val loss 2.392
Ep 1 (Step 000050): Train loss 0.617, Val loss 0.637
Ep 1 (Step 000100): Train loss 0.523, Val loss 0.557
Training accuracy: 70.00% | Validation accuracy: 72.50%
Ep 2 (Step 000150): Train loss 0.561, Val loss 0.489
Ep 2 (Step 000200): Train loss 0.419, Val loss 0.397
Ep 2 (Step 000250): Train loss 0.409, Val loss 0.353
Training accuracy: 82.50% | Validation accuracy: 85.00%
Ep 3 (Step 000300): Train loss 0.333, Val loss 0.320
Ep 3 (Step 000350): Train loss 0.340, Val loss 0.306
Training accuracy: 90.00% | Validation accuracy: 90.00%
Ep 4 (Step 000400): Train loss 0.136, Val loss 0.200
Ep 4 (Step 000450): Train loss 0.153, Val loss 0.132
Ep 4 (Step 000500): Train loss 0.222, Val loss 0.137
Training accuracy: 100.00% | Validation accuracy: 97.50%
Ep 5 (Step 000550): Train loss 0.207, Val loss 0.143
Ep 5 (Step 000600): Train loss 0.083, Val loss 0.074
Training accuracy: 100.00% | Validation accuracy: 97.50%
Training completed in 60.93 minutes.
```

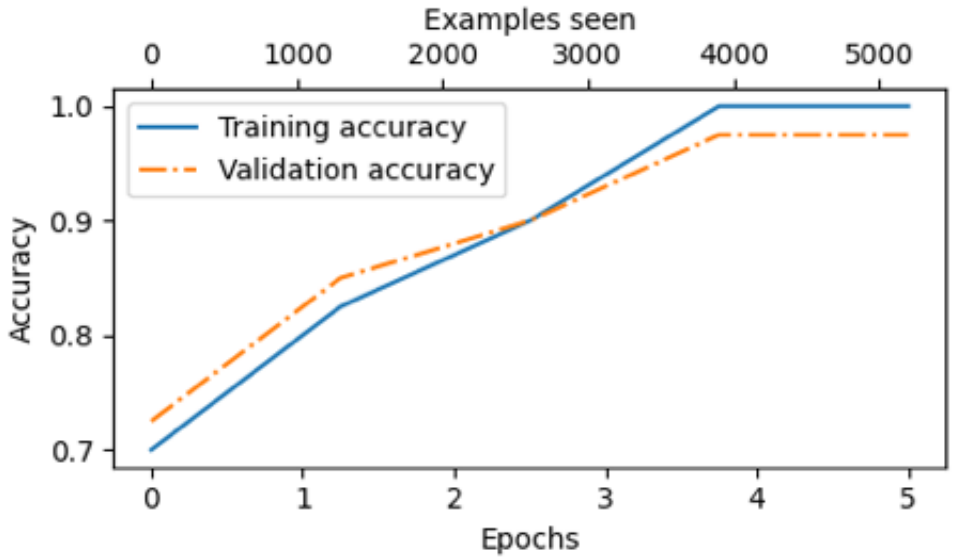
In [ ]:
```
epochs_tensor = torch.linspace(0, num_epochs, len(train_losses))
examples_seen_tensor = torch.linspace(0, examples_seen, len(train_losse

plot_values(epochs_tensor, examples_seen_tensor, train_losses, val_losse
```



Como se observa pelo declive acentuado, o modelo está aprendendo eficazmente a partir dos dados de treinamento, sem qualquer indicação perceptível de overfitting; isto é, não há lacuna notável entre as perdas do conjunto de treinamento e da validação.

In [ ]:

```
epochs_tensor = torch.linspace(0, num_epochs, len(train_accs))
examples_seen_tensor = torch.linspace(0, examples_seen, len(train_accs))

plot_values(epochs_tensor, examples_seen_tensor, train_accs, val_accs,
```



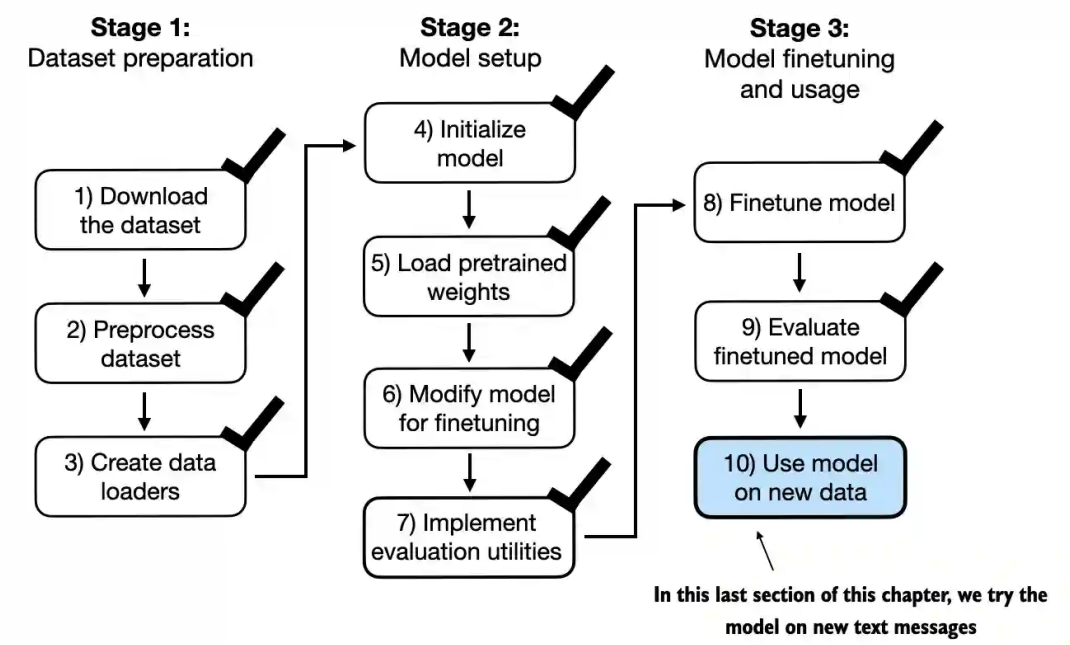
Tanto a acurácia de treinamento (linha sólida) quanto a acurácia de validação (linha tracejada) aumentam substancialmente nas primeiras épocas e, em seguida, atingem um platô, obtendo pontuações quase perfeitas de 1,0. A proximidade estreita das duas curvas ao longo das épocas indica que o modelo não sofre um overfitting significativo dos dados de treinamento.

Com base no gráfico de acurácia acima, observamos que o modelo alcança uma acurácia relativamente alta tanto em treinamento quanto em validação após as 4ª e 5ª épocas. Contudo, é importante lembrar que especificamos `eval_iter=5` na função de treino anterior, o que implica que apenas estimamos os desempenhos nos conjuntos de treinamento e validação.

In [ ]:

```
train_accuracy = calc_accuracy_loader(train_loader, model, device)
val_accuracy = calc_accuracy_loader(val_loader, model, device)
test_accuracy = calc_accuracy_loader(test_loader, model, device)

print(f"Training accuracy: {train_accuracy*100:.2f}%")
print(f"Validation accuracy: {val_accuracy*100:.2f}%")
print(f"Test accuracy: {test_accuracy*100:.2f}%")
# Training accuracy: 97.21%
# Validation accuracy: 97.32%
# Test accuracy: 95.67%
```

Usando o LLM como um classificador de Spam



In [ ]:

```
def classify_review(text, model, tokenizer, device, max_length=None, pad
    model.eval()
    input_ids = tokenizer.encode(text)
    supported_context_length = model.pos_emb.weight.shape[1]

    # Truncamento
    input_ids = input_ids[:min(max_length, supported_context_length)]

    # Pad
    input_ids += [pad_token_id] * (max_length - len(input_ids))
    input_tensor = torch.tensor(input_ids, device=device).unsqueeze(0) #

    # Inferência
    with torch.no_grad():
        logits = model(input_tensor)[:, -1, :]  # Logits of the last out
    predicted_label = torch.argmax(logits, dim=-1).item()

    return "spam" if predicted_label == 1 else "not spam"
```

In [ ]:

```
text_1 = (
    "You are a winner you have been specially"
    " selected to receive $1000 cash or a $2000 award."
)

print(classify_review(
    text_1, model, tokenizer, device, max_length=train_dataset.max_leng
))
# Resposta: Spam
```

In [ ]:

```
text_2 = (
    "Hey, just wanted to check if we're still on"
    " for dinner tonight? Let me know!"
)

print(classify_review(
    text_2, model, tokenizer, device, max_length=train_dataset.max_leng
))
# Resposta: Not Spam
```

Salva o Modelo

In [ ]:
```
torch.save(model.state_dict(), "review_classifier.pth")
```

Carrega o Modelo

In [ ]:
```
model_state_dict = torch.load("review_classifier.pth")
model.load_state_dict(model_state_dict)
```

Leitura Recomendada

- Using and Finetuning Pretrained Transformers
- The Ultimate Guide to Fine-Tuning LLMs from Basics to Breakthroughs