In [ ]:
```
import os
import json
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, GRU, Dense, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

train_data = pd.read_csv("toxic_eng/train.csv")
test_data = pd.read_csv("toxic_eng/test.csv")

train_data['label'] = train_data.get('toxic')
test_data['label'] = test_data.get('toxic')

max_words = 20000
max_length = 128
tokenizer = Tokenizer(num_words=max_words, oov_token="<OOV>")
tokenizer.fit_on_texts(train_data['comment_text'])

train_sequences = tokenizer.texts_to_sequences(train_data['comment_text'])
test_sequences = tokenizer.texts_to_sequences(test_data['comment_text'])

train_padded = pad_sequences(train_sequences, maxlen=max_length, padding="post",
test_padded = pad_sequences(test_sequences, maxlen=max_length, padding="post", t

train_labels = np.array(train_data['label'])
test_labels = np.array(test_data['label'])

model = Sequential([
    Embedding(max_words, 128, input_length=max_length),
    GRU(128, return_sequences=False),
    Dropout(0.3),
    Dense(64, activation="relu"),
    Dropout(0.3),
    Dense(1, activation="sigmoid")
])

model.compile(loss="binary_crossentropy", optimizer=Adam(learning_rate=0.001), m

model.fit(train_padded, train_labels, validation_data=(test_padded, test_labels)

print(f"Test Accuracy: {test_acc:.4f}")

# ulozenie
model.save("saved_gru_model.h5")
print("Model uložený ako saved_gru_model.h5")

# ulozenie tokenizera
tokenizer_json = tokenizer.to_json()
with open("tokenizer.json", "w", encoding="utf-8") as f:
    f.write(tokenizer_json)
print("Tokenizer uložený ako tokenizer.json")
```

In [ ]:
```
import os
import torch
```

```
import torch.nn as nn
import pandas as pd
from torch.utils.data import DataLoader, TensorDataset
import torch.optim as optim
from transformers import BertTokenizer

train_data = pd.read_csv("testy/toxic_eng/train.csv")
test_data = pd.read_csv("testy/toxic_eng/test.csv")

train_data['label'] = train_data.get('label', 0)
test_data['label'] = test_data.get('label', 0)

tokenizer = BertTokenizer.from_pretrained('bert-base-multilingual-cased')

def tokenize_texts(texts, max_length=128):
    encodings = tokenizer(
        texts.tolist(),
        add_special_tokens=True,
        max_length=max_length,
        padding='max_length',
        truncation=True,
        return_tensors='pt'
    )
    return encodings['input_ids']

train_inputs = tokenize_texts(train_data['comment_text'])
test_inputs = tokenize_texts(test_data['comment_text'])

train_labels = torch.tensor(train_data['label'].values)
test_labels = torch.tensor(test_data['label'].values)

class GRUModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, num_layers, dropout):
        super(GRUModel, self).__init__()
        self.embedding = nn.Embedding(input_dim, hidden_dim)
        self.gru = nn.GRU(hidden_dim, hidden_dim, num_layers, batch_first=True,
        self.fc = nn.Linear(hidden_dim, output_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        embedded = self.embedding(x)
        gru_out, _ = self.gru(embedded)
        gru_out = self.dropout(gru_out[:, -1, :])
        output = self.fc(gru_out)
        return output

input_dim = tokenizer.vocab_size
hidden_dim = 128
output_dim = 2
num_layers = 2
dropout = 0.3

model = GRUModel(input_dim, hidden_dim, output_dim, num_layers, dropout)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model.to(device)

optimizer = optim.Adam(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()
```

```
train_dataset = TensorDataset(train_inputs, train_labels)
test_dataset = TensorDataset(test_inputs, test_labels)

train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)

def save_checkpoint(model, optimizer, epoch, checkpoint_dir="checkpoints"):
    os.makedirs(checkpoint_dir, exist_ok=True)
    checkpoint_path = os.path.join(checkpoint_dir, f"checkpoint_epoch_{epoch}.pt
    torch.save({
        'epoch': epoch,
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict()
    }, checkpoint_path)
    print(f"Checkpoint saved: {checkpoint_path}")

def load_checkpoint(model, optimizer, checkpoint_path):
    checkpoint = torch.load(checkpoint_path)
    model.load_state_dict(checkpoint['model_state_dict'])
    optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
    epoch = checkpoint['epoch']
    print(f"Checkpoint loaded: {checkpoint_path}, starting from epoch {epoch + 1
    return epoch

def train_model(model, train_loader, criterion, optimizer, device, num_epochs=5,
    model.train()
    for epoch in range(start_epoch, num_epochs):
        total_loss = 0
        correct = 0
        total = 0
        for inputs, labels in train_loader:
            inputs, labels = inputs.to(device), labels.to(device)

            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            total_loss += loss.item()
            _, predicted = torch.max(outputs, 1)
            correct += (predicted == labels).sum().item()
            total += labels.size(0)

        accuracy = correct / total
        print(f'Epoch {epoch + 1}/{num_epochs}, Loss: {total_loss:.4f}, Accuracy

        save_checkpoint(model, optimizer, epoch, checkpoint_dir)

checkpoint_path = "checkpoints/checkpoint_epoch_2.pth"
if os.path.exists(checkpoint_path):
    start_epoch = load_checkpoint(model, optimizer, checkpoint_path) + 1

train_model(model, train_loader, criterion, optimizer, device, num_epochs=5, sta

def evaluate_model(model, test_loader, device):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
```

```
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            _, predicted = torch.max(outputs, 1)
            correct += (predicted == labels).sum().item()
            total += labels.size(0)

    accuracy = correct / total
    print(f'Test Accuracy: {accuracy:.4f}')

evaluate_model(model, test_loader, device)
```

In [ ]:

```
from sklearn.metrics import precision_score, recall_score, f1_score

# predikcie modelu
y_pred = model.predict(test_padded)
y_pred_binary = (y_pred > 0.5).astype(int)

# vypocet metrik
precision = precision_score(test_labels, y_pred_binary)
recall = recall_score(test_labels, y_pred_binary)
f1 = f1_score(test_labels, y_pred_binary)

print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1-score: {f1:.4f}")
```