In [1]:
```
import tensorflow as tf
import keras
print("TensorFlow version:", tf.__version__)
print("Keras version:", keras.__version__)
```

```
TensorFlow version: 2.18.0
Keras version: 3.6.0
```

In [2]:
```
import tensorflow as tf

# Test optimizéra
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
print("Optimizer created successfully:", optimizer)
```

```
Optimizer created successfully: <keras.src.optimizers.adam.Adam object at 0x00000
21715BA0530>
```

In [3]:
```
import cv2
import pandas as pd
train_df = pd.read_csv('toxic_eng/train.csv')  # Predpokladá stĺpce 'text' a 'la
```

In [ ]:
```
import torch
import torch.nn as nn
from torch.utils.data import DataLoader, Dataset
from transformers import BertTokenizer
import pandas as pd

class TextDataset(Dataset):
    def __init__(self, texts, labels, tokenizer, max_len):
        self.texts = texts
        self.labels = labels
        self.tokenizer = tokenizer
        self.max_len = max_len

    def __len__(self):
        return len(self.texts)

    def __getitem__(self, idx):
        text = self.texts[idx]
        label = self.labels[idx]
        encoding = self.tokenizer.encode_plus(
            text,
            add_special_tokens=True,
            max_length=self.max_len,
            padding='max_length',
            truncation=True,
            return_tensors='pt'
        )
        return {
            'input_ids': encoding['input_ids'].squeeze(0),
            'attention_mask': encoding['attention_mask'].squeeze(0),
            'label': torch.tensor(label, dtype=torch.long)
        }

# nacitanie renovacej a testovacej sady datasetu
train_df = pd.read_csv('toxic_eng/train.csv')
test_df = pd.read_csv('toxic_eng/test.csv')

train_texts = train_df['comment_text'].tolist()
```

```python
train_labels = train_df['toxic'].tolist()
test_texts = test_df['comment_text'].tolist()
test_labels = test_df['toxic'].tolist()

# inicializacia tokenizera
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")

# nastavime si maximalnu dlzku sekvencie
MAX_LEN = 128

# vytvorenie datasetov
train_dataset = TextDataset(train_texts, train_labels, tokenizer, max_len=MAX_LE
test_dataset = TextDataset(test_texts, test_labels, tokenizer, max_len=MAX_LEN)

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

In [ ]:

```python
# v pripade ze su v mnozine nejake chybajuce hodnoty tak musime tieto zaznamy vy
train_texts = train_df.dropna(subset=['comment_text', 'toxic'])
test_texts = test_df.dropna(subset=['comment_text', 'toxic'])
import re
import html
# v nasledujucej casti vycistime text oz zbytocnych html elementov a taktiez nad
def clean_text(text):
    text = html.unescape(text)
    text = re.sub(r'<[^>]+>', '', text)
    text = re.sub(r'\s+', ' ', text)
    text = text.strip()
    return text

train_texts['comment_text'] = train_df['comment_text'].apply(clean_text)
test_texts['comment_text'] = test_texts['comment_text'].apply(clean_text)
from sklearn.utils.class_weight import compute_class_weight
import numpy as np



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

# vypocet vah
class_weights = compute_class_weight(class_weight='balanced', classes=np.unique(
class_weights = torch.tensor(class_weights, dtype=torch.float).to(device)

# stratova funkcia
criterion = nn.CrossEntropyLoss(weight=class_weights)
```

In [4]:

```python
class LSTMClassifier(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim, output_dim, num_layers
        super(LSTMClassifier, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim)
        self.lstm = nn.LSTM(embed_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)
        lstm_out, _ = self.lstm(embedded)
        lstm_out = self.dropout(lstm_out[:, -1, :])
```

```
        output = self.fc(lstm_out)
        return output
```

In [ ]:
```
import torch
from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_sc
import numpy as np
from tqdm import tqdm

def evaluate_model_with_tqdm(model, test_loader, device, silent=False):
    model.eval()
    correct = 0
    total = 0
    all_preds = []
    all_labels = []

    test_loader_tqdm = tqdm(test_loader, desc="Evaluating", leave=False) if not

    with torch.no_grad():
        for batch in test_loader_tqdm:
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            labels = batch['labels'].to(device)

            outputs = model(input_ids=input_ids, attention_mask=attention_mask)
            logits = outputs.logits

            # vytvorenie predikcii
            _, preds = torch.max(logits, dim=1)

            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

            correct += (preds == labels).sum().item()
            total += labels.size(0)

    accuracy = accuracy_score(all_labels, all_preds)
    precision = precision_score(all_labels, all_preds)
    recall = recall_score(all_labels, all_preds)
    f1 = f1_score(all_labels, all_preds)
    cm = confusion_matrix(all_labels, all_preds)

    if not silent:
        print(f"Test Accuracy: {accuracy:.4f}")
        print(f"Precision: {precision:.4f}")
        print(f"Recall: {recall:.4f}")
        print(f"F1 Score: {f1:.4f}")
        print(f"Confusion Matrix:\n{cm}")

    return accuracy, precision, recall, f1, cm
```

In [ ]:
```
def train_model_with_tqdm(model, train_loader, test_loader, criterion, optimizer
    model.train()
    best_accuracy = 0
    epochs_without_improvement = 0
    best_model_state = None

    for epoch in range(num_epochs):
        total_loss = 0
        correct = 0
```

```
        total = 0
        train_loader_tqdm = tqdm(train_loader, desc=f"Training Epoch {epoch+1}/{

        for batch in train_loader_tqdm:
            input_ids = batch['input_ids'].to(device)
            labels = batch['label'].to(device)

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

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

        train_accuracy = correct / total
        print(f"Epoch {epoch+1}, Loss: {total_loss:.4f}, Training Accuracy: {tra

        # spustime model na testovacich datach
        test_accuracy = evaluate_model_with_tqdm(model, test_loader, device, sil

        # model predcastneho ukoncenia
        if test_accuracy > best_accuracy:
            best_accuracy = test_accuracy
            epochs_without_improvement = 0
            best_model_state = model.state_dict()
        else:
            epochs_without_improvement += 1
            if epochs_without_improvement >= patience:
                print(f"Early stopping triggered after {epoch+1} epochs. Best te
                break

    # nacitanie modelu
    if best_model_state:
        model.load_state_dict(best_model_state)
        print("Loaded best model state.")

import torch
from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_sc
import numpy as np
from tqdm import tqdm

def evaluate_model_with_tqdm(model, test_loader, device, silent=False):
    model.eval()
    correct = 0
    total = 0
    all_preds = []
    all_labels = []

    test_loader_tqdm = tqdm(test_loader, desc="Evaluating", leave=False) if not

    with torch.no_grad():
        for batch in test_loader_tqdm:
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            labels = batch['labels'].to(device)
```

```
            outputs = model(input_ids=input_ids, attention_mask=attention_mask)
            logits = outputs.logits

            _, preds = torch.max(logits, dim=1)

            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

            correct += (preds == labels).sum().item()
            total += labels.size(0)

    accuracy = accuracy_score(all_labels, all_preds)
    precision = precision_score(all_labels, all_preds)
    recall = recall_score(all_labels, all_preds)
    f1 = f1_score(all_labels, all_preds)
    cm = confusion_matrix(all_labels, all_preds)

    if not silent:
        print(f"Test Accuracy: {accuracy:.4f}")
        print(f"Precision: {precision:.4f}")
        print(f"Recall: {recall:.4f}")
        print(f"F1 Score: {f1:.4f}")
        print(f"Confusion Matrix:\n{cm}")

    return accuracy, precision, recall, f1, cm
```

In [ ]:
```
import os

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

# nacitanie
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

# Trening
def train_model_with_tqdm_and_checkpoints(model, train_loader, criterion, optimi
    model.train()
    start_epoch = 0
    checkpoint_path = os.path.join(checkpoint_dir, f"{model_name}_epoch_{start_e
    if os.path.exists(checkpoint_path):
        start_epoch = load_checkpoint(model, optimizer, checkpoint_path) + 1

    for epoch in range(start_epoch, num_epochs):
        total_loss = 0
        correct = 0
```

```
        total = 0
        train_loader_tqdm = tqdm(train_loader, desc=f"Training Epoch {epoch+1}/{
        for batch in train_loader_tqdm:
            input_ids = batch['input_ids'].to(device)
            labels = batch['label'].to(device)

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

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

            train_loader_tqdm.set_postfix(loss=loss.item())

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

        # Ulož checkpoint po každej epoch
        save_checkpoint(model, optimizer, epoch, checkpoint_dir, model_name)
```

In [ ]:

```
# Definujeme si parametre modelu
vocab_size = tokenizer.vocab_size
embed_dim = 128
hidden_dim = 256
output_dim = 2  # kedze ide o binarnu kalsifikaciu tak musime nastavit hodnotu 2
num_layers = 2
dropout = 0.3

# zvolime zariadenie a inicializujeme modelov
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

lstm_model = LSTMClassifier(vocab_size, embed_dim, hidden_dim, output_dim, num_l
lstm_optimizer = torch.optim.Adam(lstm_model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()

print("Training LSTM s checkpointy")
train_model_with_tqdm_and_checkpoints(lstm_model, train_loader, criterion, lstm_
evaluate_model_with_tqdm(lstm_model, test_loader, device)
```

In [ ]:

```
model_save_path = "lstm_model_final.pth"
torch.save(lstm_model.state_dict(), model_save_path)
print(f"ulozenie do: {model_save_path}")
```

In [ ]:

```
vocab_size = tokenizer.vocab_size
embed_dim = 128
hidden_dim = 256
output_dim = 2  # binarna klasifikacia
num_layers = 2
dropout = 0.3

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
lstm_model = LSTMClassifier(vocab_size, embed_dim, hidden_dim, output_dim, num_l
lstm_optimizer = torch.optim.Adam(lstm_model.parameters(), lr=0.001)
```

```
criterion = nn.CrossEntropyLoss()
accuracy, precision, recall, f1, cm = evaluate_model_with_tqdm(lstm_model, test_
```

In [ ]:

```

```