SHARDA UNIVERSITY

Beyond Boundaries

AGENTIC AI- LAB

Md Mashroof Kalim (2023479862)

CS-F-G2

CHUNKING METHOD

```
#!/usr/bin/env python3
"""
fine tunning [LAB-1]

General-purpose fine-tuning script for Hugging Face Transformers.

Supports:
- sequence classification (binary/multi-class)
- causal language modeling (GPT-style)

Usage examples:
  # classification from CSV with columns "text" and "label"
  python "fine tunning [LAB-1]" \
    --task classification \
    --model_name_or_path gpt2 \
    --dataset_name_or_path /path/to/data.csv \
    --dataset_format csv \
    --text_column text \
    --label_column label \
    --output_dir ./outputs/classifier \
    --num_train_epochs 3 \
    --per_device_train_batch_size 8

  # causal LM using a HF dataset
  python "fine tunning [LAB-1]" \
    --task causal_lm \
    --model_name_or_path gpt2 \
    --dataset_name_or_path wikitext \
    --dataset_config_name wikitext-2-raw-v1 \
    --output_dir ./outputs/gpt2-lm \
    --do_train
"""

import argparse
import logging
import os
from dataclasses import dataclass
from typing import Optional, Dict, Any

import numpy as np

from datasets import load_dataset, DatasetDict
from transformers import (
    AutoConfig,
```

```
    AutoModelForSequenceClassification,
    AutoModelForCausalLM,
    AutoTokenizer,
    DataCollatorForLanguageModeling,
    DataCollatorWithPadding,
    Trainer,
    TrainingArguments,
)
from sklearn.metrics import accuracy_score, f1_score

logger = logging.getLogger(__name__)
logging.basicConfig(level=logging.INFO)

def parse_args():
    parser = argparse.ArgumentParser(description="Fine-tune a Transformers model (classification or causal LM).")
    parser.add_argument("--task", type=str, required=True, choices=["classification", "causal_lm"],
                        help="Task to run.")
    parser.add_argument("--model_name_or_path", type=str, required=True,
                        help="Pretrained model name or path.")
    parser.add_argument("--dataset_name_or_path", type=str, required=True,
                        help="Hugging Face dataset identifier or local path (CSV/JSON/dir).")
    parser.add_argument("--dataset_config_name", type=str, default=None,
                        help="Dataset config name (if applicable).")
    parser.add_argument("--dataset_format", type=str, default=None, choices=[None, "csv", "json"],
                        help="If loading from local file, specify format.")
    parser.add_argument("--text_column", type=str, default="text", help="Name of the text column.")
    parser.add_argument("--label_column", type=str, default="label", help="Name of the label column (classification).")
    parser.add_argument("--max_length", type=int, default=512)
    parser.add_argument("--output_dir", type=str, default="./outputs")
    parser.add_argument("--per_device_train_batch_size", type=int, default=8)
    parser.add_argument("--per_device_eval_batch_size", type=int, default=8)
    parser.add_argument("--learning_rate", type=float, default=5e-5)
    parser.add_argument("--weight_decay", type=float, default=0.0)
    parser.add_argument("--num_train_epochs", type=int, default=3)
    parser.add_argument("--seed", type=int, default=42)
    parser.add_argument("--do_train", action="store_true")
    parser.add_argument("--do_eval", action="store_true")
    parser.add_argument("--overwrite_output_dir", action="store_true")
    parser.add_argument("--save_total_limit", type=int, default=3)
    parser.add_argument("--logging_steps", type=int, default=100)
    parser.add_argument("--evaluation_strategy", type=str, default="steps", choices=["no", "steps", "epoch"])
    parser.add_argument("--eval_steps", type=int, default=500)
    parser.add_argument("--save_steps", type=int, default=500)
```

```
    parser.add_argument("--block_size", type=int, default=1024, help="For LM: max sequence
length")
    return parser.parse_args()

def load_data(args):
    # tries to be flexible: if dataset_name_or_path is a local file, use dataset_format
    load_kwargs = {}
    if args.dataset_config_name:
        load_kwargs["name"] = args.dataset_config_name

    if args.dataset_format:
        # local file(s)
        if os.path.isdir(args.dataset_name_or_path):
            ds = load_dataset(args.dataset_format, data_files={
                "train": os.path.join(args.dataset_name_or_path, "train." + args.dataset_format),
                "validation": os.path.join(args.dataset_name_or_path, "validation." +
args.dataset_format),
            })
        else:
            ds = load_dataset(args.dataset_format, data_files=args.dataset_name_or_path)
        return ds
    else:
        # try HF hub
        ds = load_dataset(args.dataset_name_or_path, **load_kwargs)
        return ds

def preprocess_for_classification(dataset, tokenizer, args):
    def preprocess_fn(examples):
        texts = examples[args.text_column]
        if isinstance(texts, str):
            texts = [texts]
        return tokenizer(texts, truncation=True, max_length=args.max_length)

    mapped = dataset.map(preprocess_fn, batched=True)
    return mapped

def compute_classification_metrics(predictions, references) -> Dict[str, float]:
    preds = np.argmax(predictions, axis=1)
    acc = accuracy_score(references, preds)
    f1 = f1_score(references, preds, average="weighted")
    return {"accuracy": acc, "f1": f1}

def main():
    args = parse_args()
    # Repro
    np.random.seed(args.seed)

    # Load tokenizer & model
```

```
    logger.info("Loading tokenizer and model from %s", args.model_name_or_path)
    tokenizer = AutoTokenizer.from_pretrained(args.model_name_or_path, use_fast=True)

    if args.task == "classification":
        # Load dataset
        raw_datasets = load_data(args)
        # If labels are not ints, try to map to ints
        # Ensure dataset has train/validation splits (Trainer expects them)
        if "train" not in raw_datasets:
            # if single-file dataset, datasets returns dict with key name; try to split
            raw_datasets = DatasetDict({"train": raw_datasets})
        # map labels to ints if needed
        # Try to infer label list from dataset features
        label_list = None
        try:
            features = raw_datasets["train"].features
            if args.label_column in features and hasattr(features[args.label_column], "names"):
                label_list = features[args.label_column].names
        except Exception:
            label_list = None

        # Tokenize
        tokenized = preprocess_for_classification(raw_datasets, tokenizer, args)

        # Model config; if label_list is known, set num_labels
        num_labels = None
        if label_list:
            num_labels = len(label_list)
        else:
            # infer from dataset labels
            try:
                labels = tokenized["train"][args.label_column]
                num_labels = len(set(labels))
            except Exception:
                num_labels = 2

        config = AutoConfig.from_pretrained(args.model_name_or_path, num_labels=num_labels)
        model = AutoModelForSequenceClassification.from_pretrained(args.model_name_or_path,
config=config)

        data_collator = DataCollatorWithPadding(tokenizer=tokenizer)

        def compute_metrics(eval_pred):
            logits, labels = eval_pred
            return compute_classification_metrics(logits, labels)

        training_args = TrainingArguments(
            output_dir=args.output_dir,
```

```
            overwrite_output_dir=args.overwrite_output_dir,
            num_train_epochs=args.num_train_epochs,
            per_device_train_batch_size=args.per_device_train_batch_size,
            per_device_eval_batch_size=args.per_device_eval_batch_size,
            learning_rate=args.learning_rate,
            weight_decay=args.weight_decay,
            evaluation_strategy=args.evaluation_strategy if args.do_eval else "no",
            eval_steps=args.eval_steps,
            save_steps=args.save_steps,
            save_total_limit=args.save_total_limit,
            logging_steps=args.logging_steps,
            seed=args.seed,
        )

        trainer = Trainer(
            model=model,
            args=training_args,
            train_dataset=tokenized.get("train", None),
            eval_dataset=tokenized.get("validation", None),
            tokenizer=tokenizer,
            data_collator=data_collator,
            compute_metrics=compute_metrics,
        )

        if args.do_train:
            trainer.train()
            trainer.save_model()

        if args.do_eval:
            result = trainer.evaluate()
            logger.info("Evaluation results: %s", result)

    elif args.task == "causal_lm":
        # Language modeling path
        raw_datasets = load_data(args)

        # We expect text column; if the dataset items are strings, adjust accordingly
        text_col = args.text_column

        def tokenize_function(examples):
            # examples can be dict or string
            texts = examples[text_col] if isinstance(examples, dict) and text_col in examples else
examples
            return tokenizer(texts, return_special_tokens_mask=True, truncation=True,
max_length=args.block_size)

        # If the dataset provides splits, operate on them
        if "train" not in raw_datasets and not isinstance(raw_datasets, DatasetDict):
```

```
        raw_datasets = DatasetDict({"train": raw_datasets})

    tokenized = raw_datasets.map(
        tokenize_function,
        batched=True,
        remove_columns=[c for c in raw_datasets["train"].column_names if c != text_col] if
text_col in raw_datasets["train"].column_names else None,
    )

    # Group into blocks
    def group_texts(examples):
        concatenated = {k: sum(examples[k], []) for k in examples.keys()}
        total_length = len(concatenated[next(iter(concatenated))])
        block_size = args.block_size
        total_length = (total_length // block_size) * block_size
        result = {}
        for k, v in concatenated.items():
            result[k] = [v[i : i + block_size] for i in range(0, total_length, block_size)]
        result["labels"] = result["input_ids"].copy()
        return result

    lm_datasets = tokenized.map(group_texts, batched=True)

    model = AutoModelForCausalLM.from_pretrained(args.model_name_or_path)
    data_collator = DataCollatorForLanguageModeling(tokenizer=tokenizer, mlm=False)

    training_args = TrainingArguments(
        output_dir=args.output_dir,
        overwrite_output_dir=args.overwrite_output_dir,
        num_train_epochs=args.num_train_epochs,
        per_device_train_batch_size=args.per_device_train_batch_size,
        per_device_eval_batch_size=args.per_device_eval_batch_size,
        learning_rate=args.learning_rate,
        weight_decay=args.weight_decay,
        evaluation_strategy=args.evaluation_strategy if args.do_eval else "no",
        eval_steps=args.eval_steps,
        save_steps=args.save_steps,
        save_total_limit=args.save_total_limit,
        logging_steps=args.logging_steps,
        seed=args.seed,
    )

    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=lm_datasets.get("train", None),
        eval_dataset=lm_datasets.get("validation", None),
        tokenizer=tokenizer,
```

```
            data_collator=data_collator,
        )

        if args.do_train:
            trainer.train()
            trainer.save_model()

        if args.do_eval:
            result = trainer.evaluate()
            logger.info("Evaluation results: %s", result)

    else:
        raise ValueError(f"Unknown task {args.task}")


if __name__ == "__main__":
    main()
```

https://github.com/alishagupta1234raj-rgb/AGENTIC-AI		