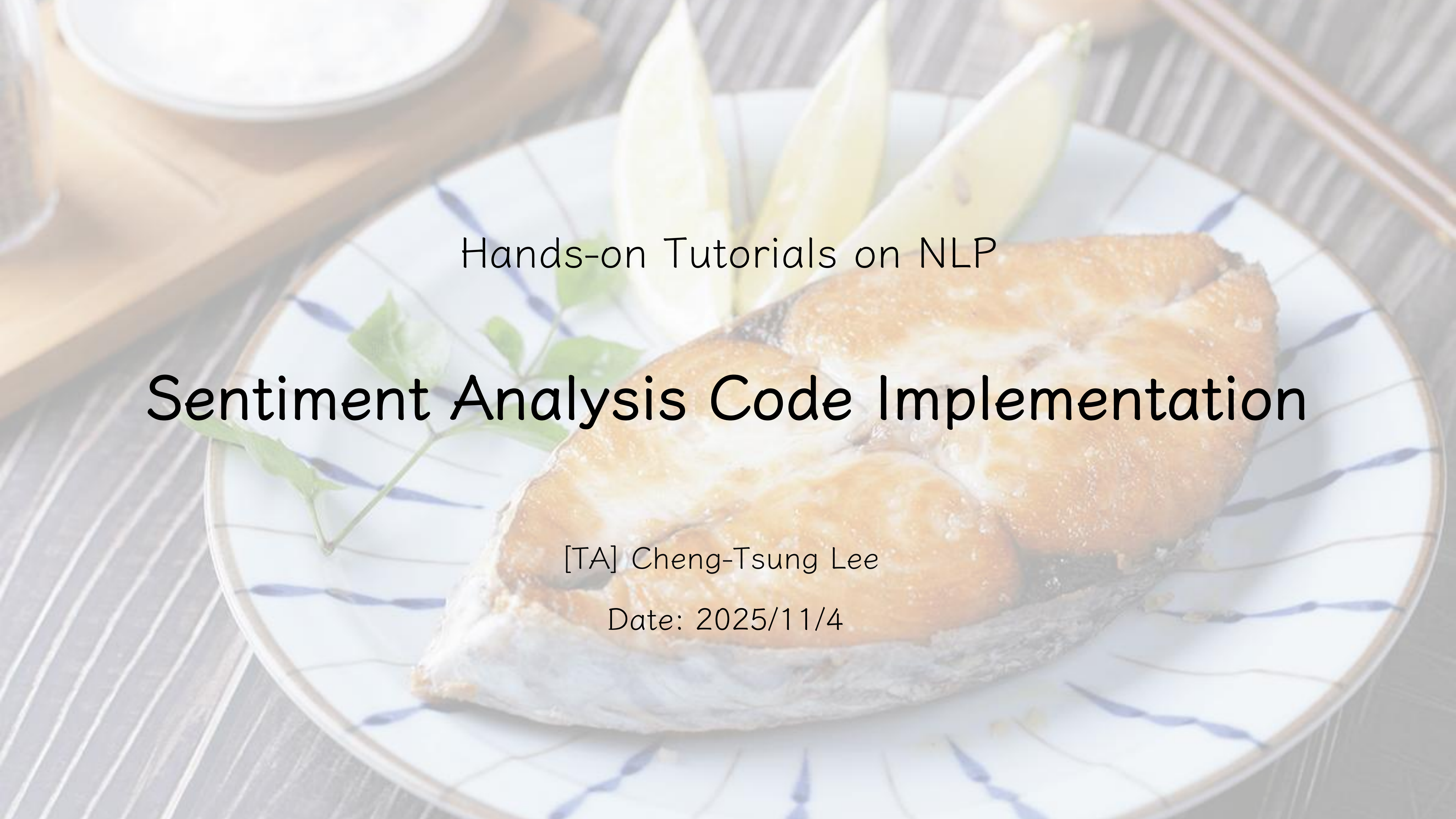


Hands-on Tutorials on NLP

Sentiment Analysis Code Implementation

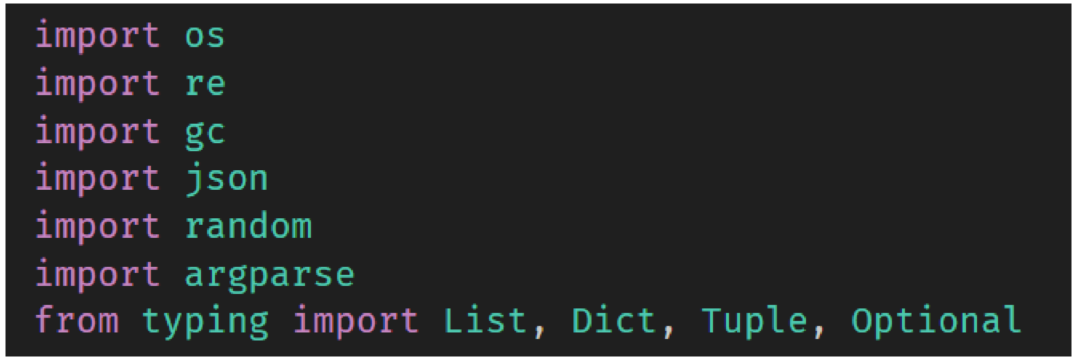
[TA] Cheng-Tsung Lee

Date: 2025/11/4

Outline:

Useful Libraries:

Python Standard Libraries



```python
import os
import re
import gc
import json
import random
import argparse
from typing import List, Dict, Tuple, Optional
```

"os" (operating system): handle file paths and directory creation

"re" (regular expression): regular expressions for text pattern matching

"gc" (garbage collection): help free unused memory

"json" (javascript object notation): read / write json files

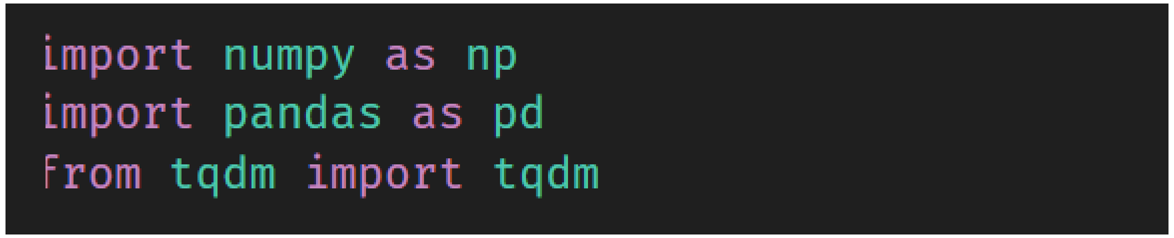
"random": create random seed for reproducibility

"argparse" (argument parser): build your own command lines

"typing": describe data types (e.g. list, dictionary, ...)

Useful Libraries:

Data Processing Libraries



```python
import numpy as np
import pandas as pd
from tqdm import tqdm
```

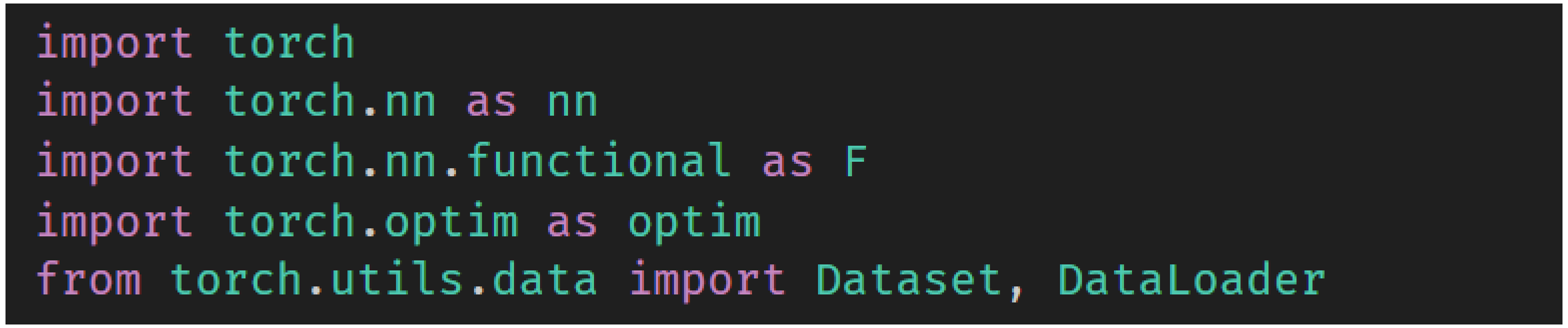
"numpy" (numerical python extensions): fast numerical (like vectors, matrices) computation

"pandas" (panel data): handle and manipulate data frame (e.g. tabular data)

"tqdm" (taqaddum): display progress bars for training loops

Useful Libraries:

Pytorch Libraries



```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
```

"torch" : machine / deep learning framework

"torch.nn" : build neural network blocks (e.g. layers, loss functions)

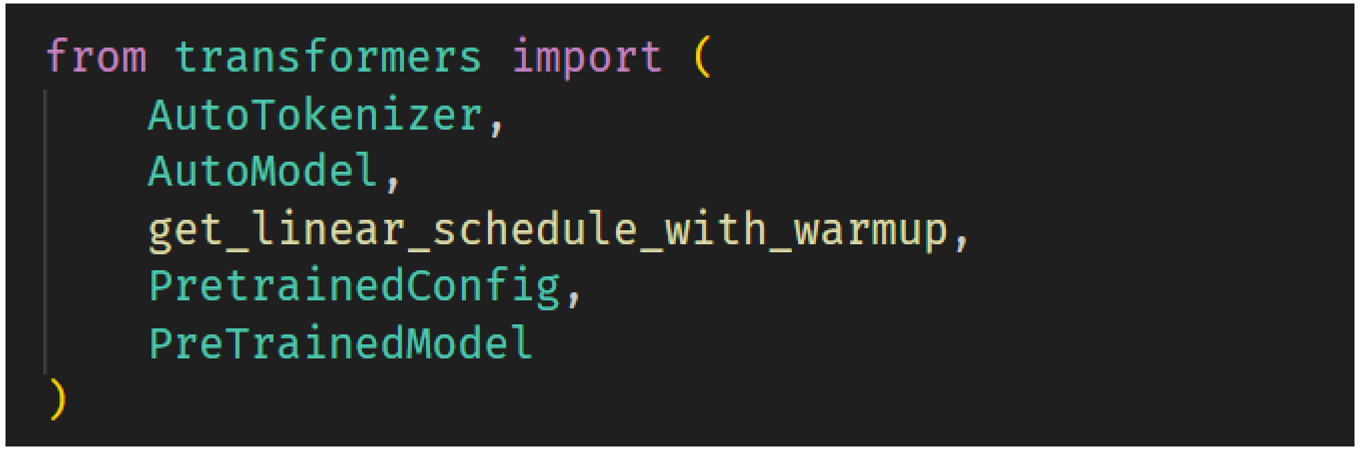
"torch.nn.functional" : call for activation functions / loss

"torch.optim" : optimization algorithm (e.g. Adam, SGD, ...)

"torch.utils.data" : Dataset -> load and preprocess data / DataLoader -> define batching and shuffling

Useful Libraries:

Hugging Face Libraries



```python
from transformers import (
    AutoTokenizer,
    AutoModel,
    get_linear_schedule_with_warmup,
    PretrainedConfig,
    PreTrainedModel
)
```

"AutoTokenizer" : convert raw text into model-readable tokens

"AutoModel" : load pretrained Transformer models

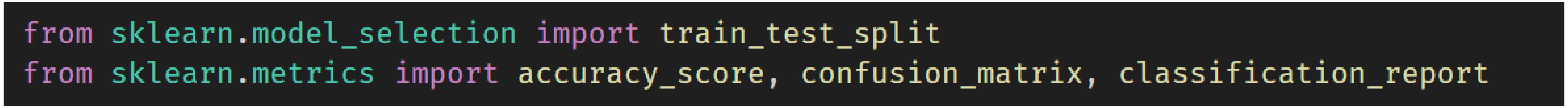
"get_linear_schedule_with_warmup" : learning rate schedular

"PretrainedConfig" : handle model configuration saving / loading (helpful for running inference)

"PreTrainedModel" : handle loading weights from checkpoint / saving model configurations

Useful Libraries:

Scikit-learn Utilities



```python
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
```

"train_test_split" : split dataset into training and validation subsets

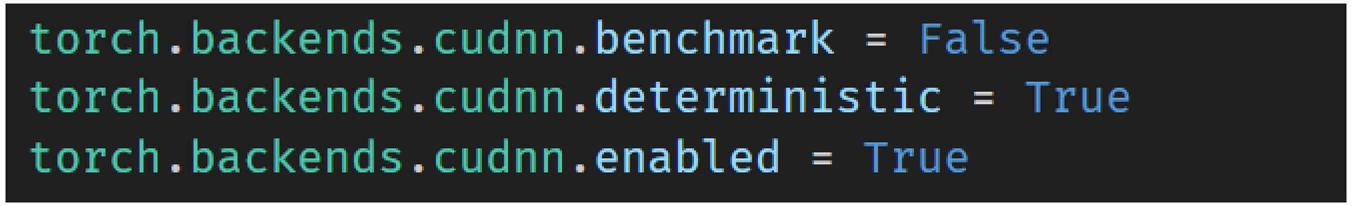
"accuracy_score" : compute percentage of correct predictions

"confusion_matrix" : show how many predictions per class are correct / wrong

"classification_report" : precision / recall / F1 metrics summary

Utility Functions:

PyTorch Backend Definition



```
torch.backends.cudnn.benchmark = False
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.enabled = True
```

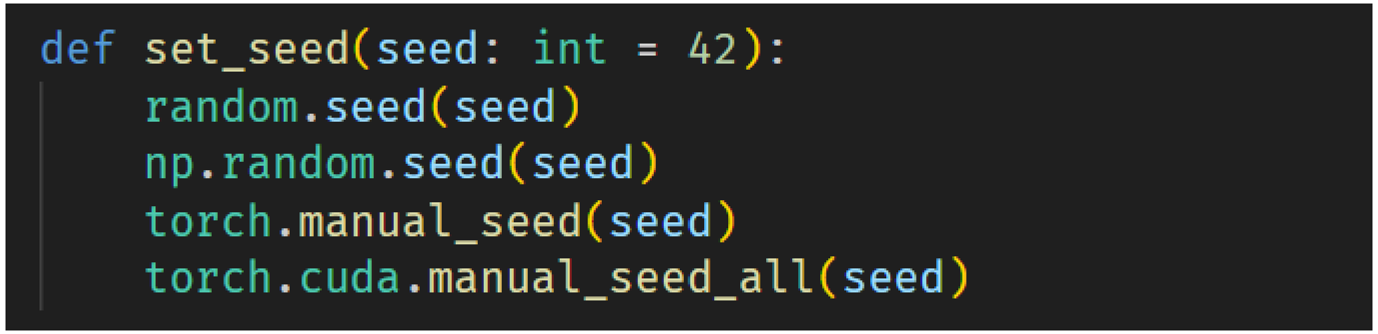
"~.benchmark" : try to find fastest algorithm for your hardware

"~.deterministic" : use only deterministic algorithms

"~.enabled" : enable acceleration on GPU

Utility Functions:

Random Seed Setup



```
def set_seed(seed: int = 42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
```

"random.seed()" : control Python built-in random operation (e.g. shuffle order)

"np.random.seed()" : control NumPy operation (used in data preprocessing)

"torch.manual_seed()" : control random behavior for CPU tensors

"torch.cuda.manual_seed()" : control random behavior for GPU setups

Utility Functions:

Why random seed = 42 ?

DON'T PANIC!
道格拉斯・亞當斯 著
丁世佳 譯
銀河便車指南
想離開本行星嗎？你會需要「銀河便車指南」
"IRRESISTIBLE!" —— The Boston Globe
DOUGLAS NOËl ADAMS
THE HITCHHIKER'S GUIDE TO THE GALAXY

Ref: https://www.eslite.com/product/1001128312485185?srsltid=AfmBOoqeID55hmRT-rfZJXUntXXQuDMbnXNAdwc8dTzd1iU0Edz0599V

Utility Functions:

Device Setup

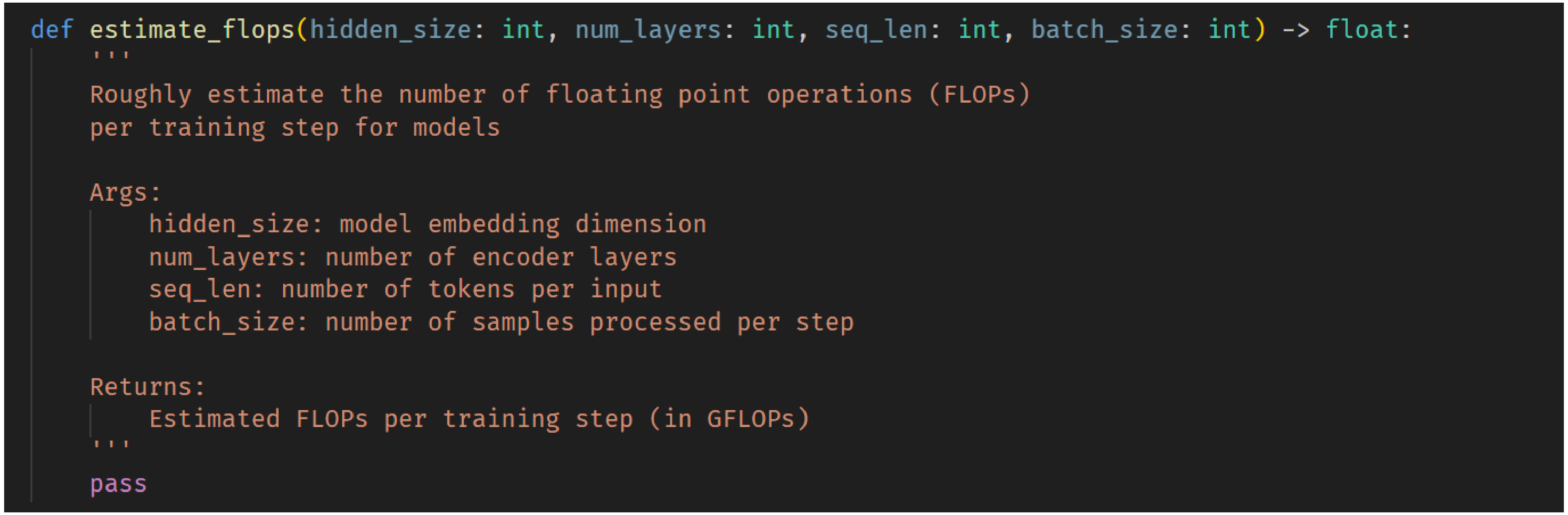
```
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

"torch.device()" : *create a device object representing where tensors and models are stored and execute*

"torch.cuda.is_available()" : *return True if CUDA-compatible GPU is detected*

Utility Functions:

Computational Cost Estimation



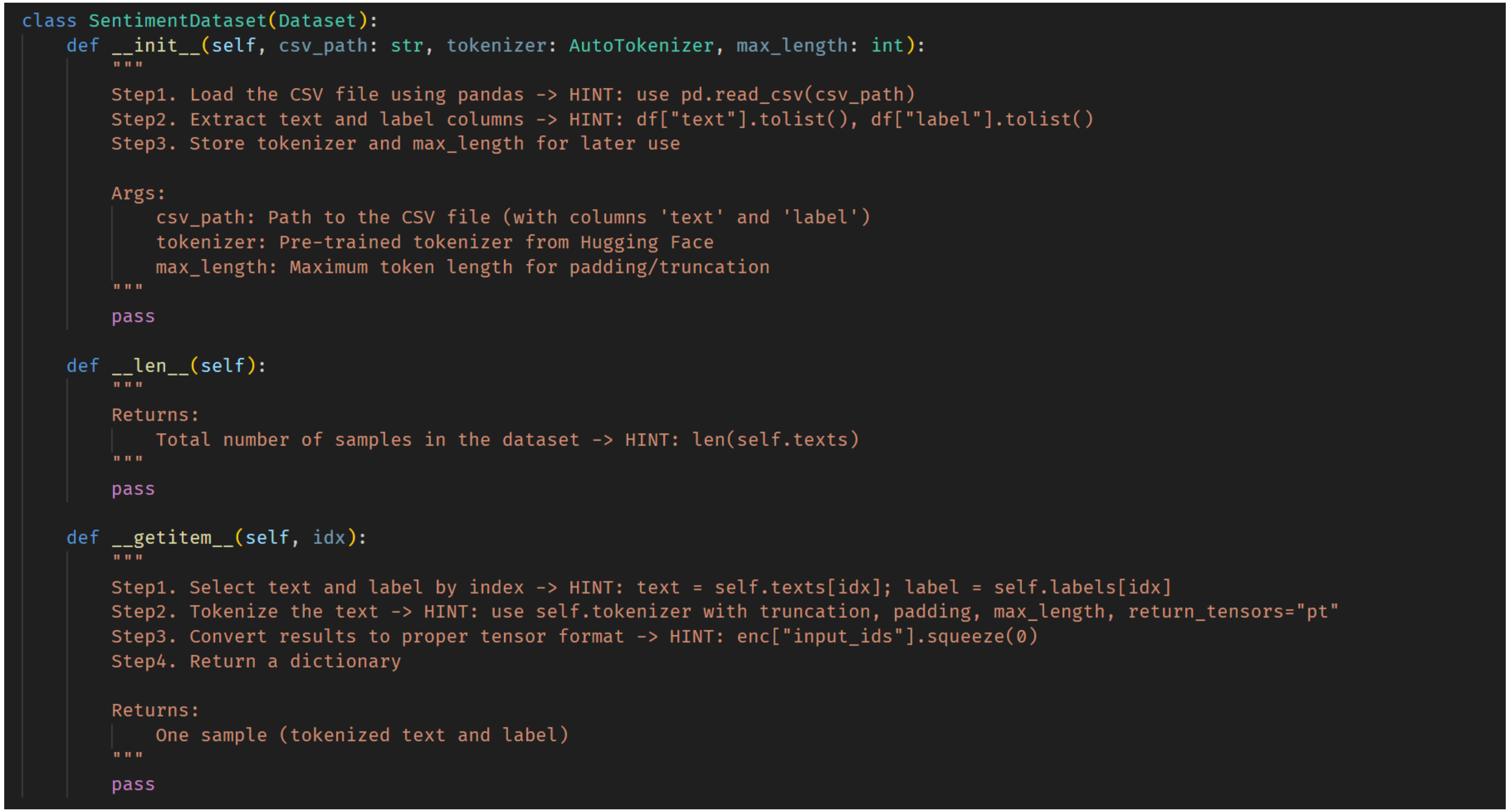
```python
def estimate_flops(hidden_size: int, num_layers: int, seq_len: int, batch_size: int) -> float:
    '''
    Roughly estimate the number of floating point operations (FLOPs)
    per training step for models

    Args:
        hidden_size: model embedding dimension
        num_layers: number of encoder layers
        seq_len: number of tokens per input
        batch_size: number of samples processed per step

    Returns:
        Estimated FLOPs per training step (in GFLOPs)
    '''
    pass
```

Hint:

Compute cost per layer, then multiply by number of layers and batch size. Divide by 10^9 at last (convert to GFLOPs)

Utility Functions:

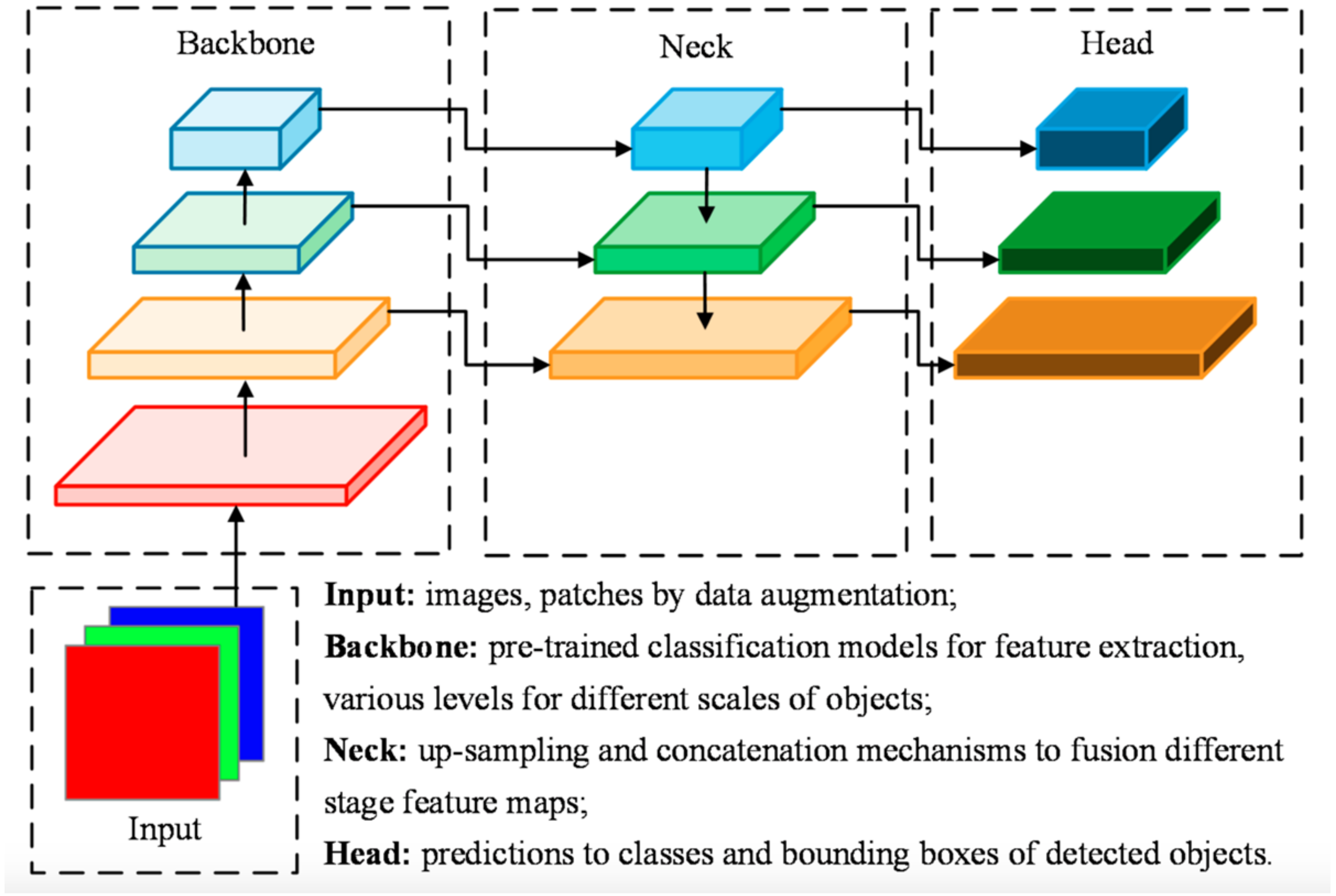
Build Custom Dataset



```python
class SentimentDataset(Dataset):
    def __init__(self, csv_path: str, tokenizer: AutoTokenizer, max_length: int):
        """
        Step1. Load the CSV file using pandas -> HINT: use pd.read_csv(csv_path)
        Step2. Extract text and label columns -> HINT: df["text"].tolist(), df["label"].tolist()
        Step3. Store tokenizer and max_length for later use

        Args:
            csv_path: Path to the CSV file (with columns 'text' and 'label')
            tokenizer: Pre-trained tokenizer from Hugging Face
            max_length: Maximum token length for padding/truncation
        """
        pass

    def __len__(self):
        """
        Returns:
            Total number of samples in the dataset -> HINT: len(self.texts)
        """
        pass

    def __getitem__(self, idx):
        """
        Step1. Select text and label by index -> HINT: text = self.texts[idx]; label = self.labels[idx]
        Step2. Tokenize the text -> HINT: use self.tokenizer with truncation, padding, max_length, return_tensors="pt"
        Step3. Convert results to proper tensor format -> HINT: enc["input_ids"].squeeze(0)
        Step4. Return a dictionary

        Returns:
            One sample (tokenized text and label)
        """
        pass
```

Model Architecture:

Big Picture



Input: images, patches by data augmentation;

Backbone: pre-trained classification models for feature extraction, various levels for different scales of objects;

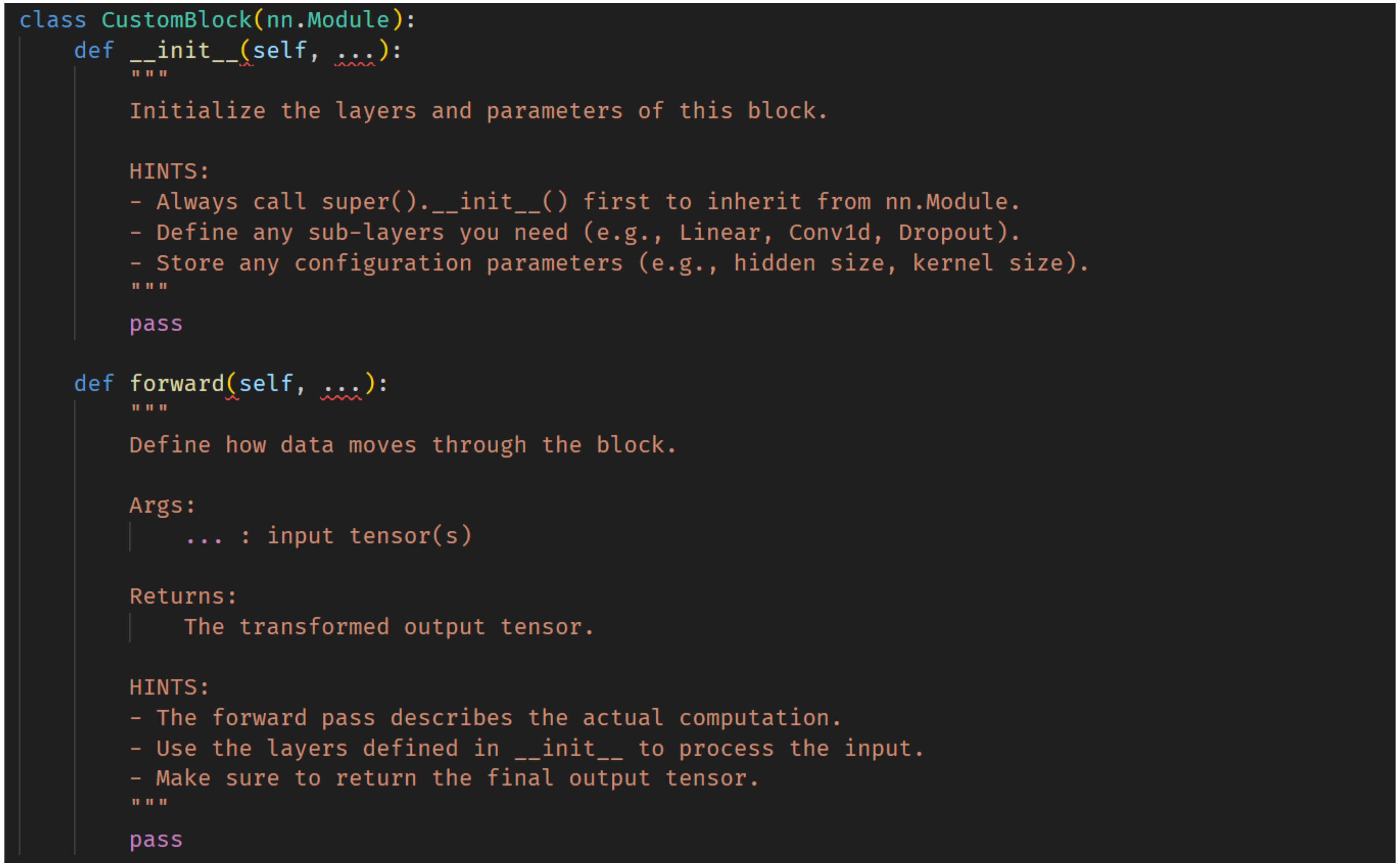
Neck: up-sampling and concatenation mechanisms to fusion different stage feature maps;

Head: predictions to classes and bounding boxes of detected objects.

Ref: https://velog.io/@peterkim/Object-Detection%EC%97%90%EC%84%9C-%EB%A7%90%ED%95%98%EB%8A%94-Backbone-Neck-Head

Model Architecture:

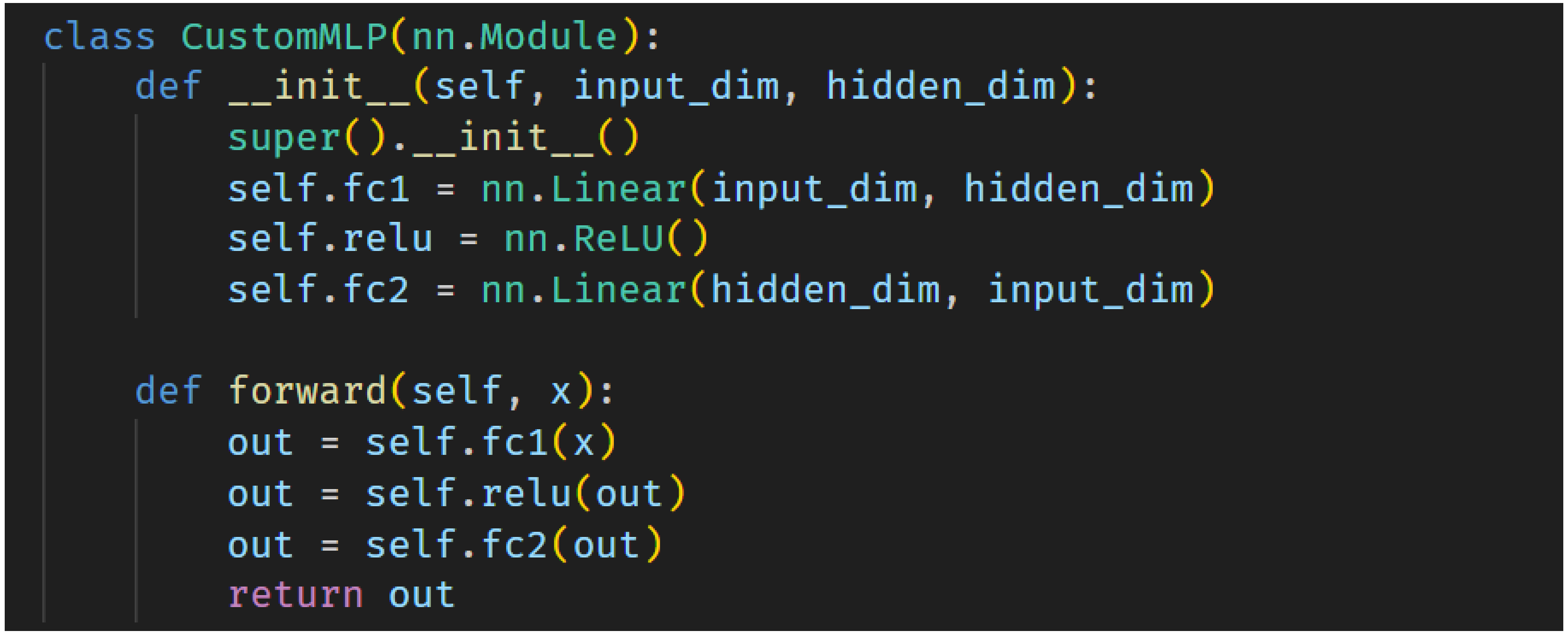
Design Custom Blocks



```python
class CustomBlock(nn.Module):
    def __init__(self, ...):
        """
        Initialize the layers and parameters of this block.

        HINTS:
        - Always call super().__init__() first to inherit from nn.Module.
        - Define any sub-layers you need (e.g., Linear, Conv1d, Dropout).
        - Store any configuration parameters (e.g., hidden size, kernel size).
        """
        pass

    def forward(self, ...):
        """
        Define how data moves through the block.

        Args:
            ... : input tensor(s)

        Returns:
            The transformed output tensor.

        HINTS:
        - The forward pass describes the actual computation.
        - Use the layers defined in __init__ to process the input.
        - Make sure to return the final output tensor.
        """
        pass
```

Model Architecture:

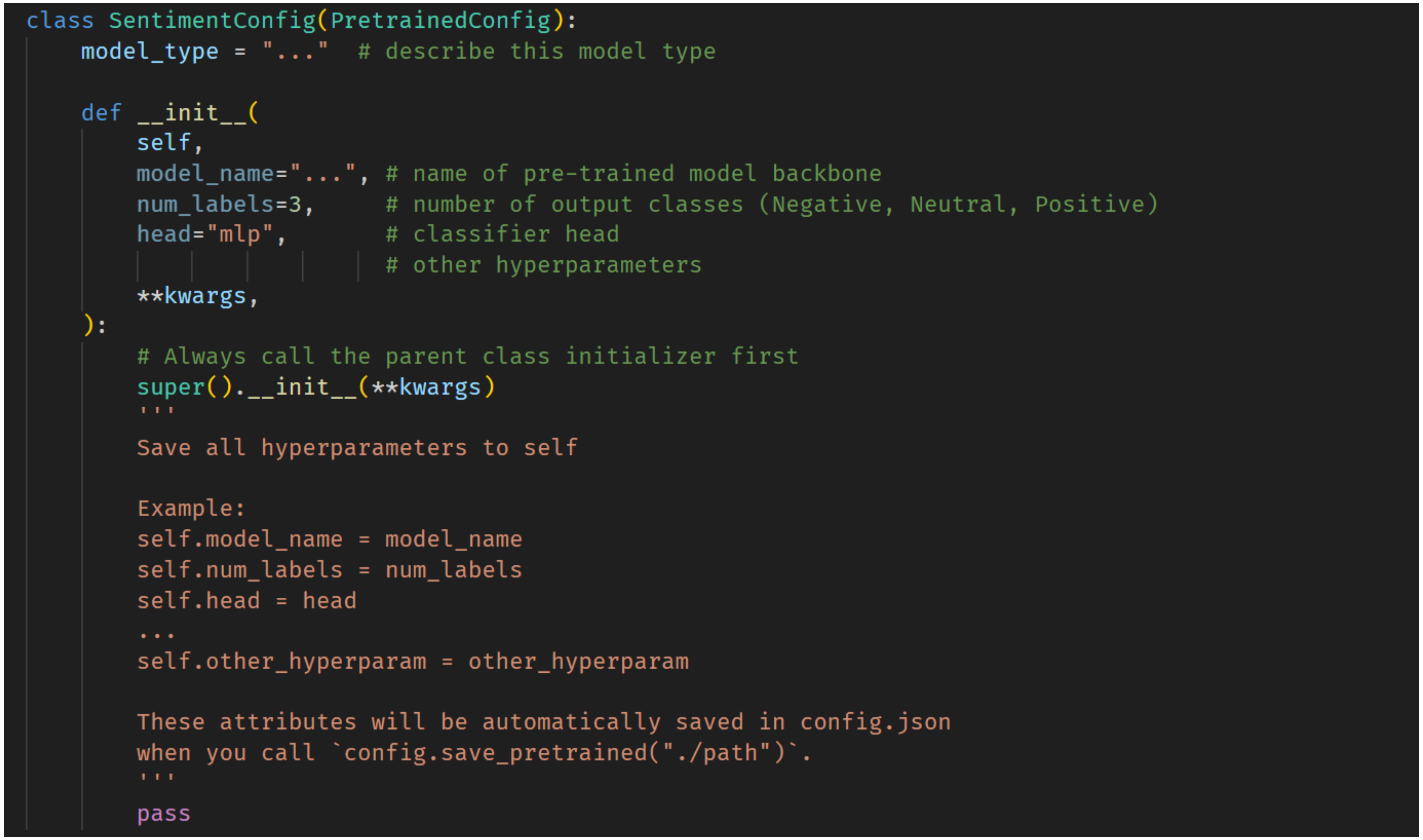
Simple MLP Example



```python
class CustomMLP(nn.Module):
    def __init__(self, input_dim, hidden_dim):
        super().__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_dim, input_dim)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return out
```

Model Architecture:

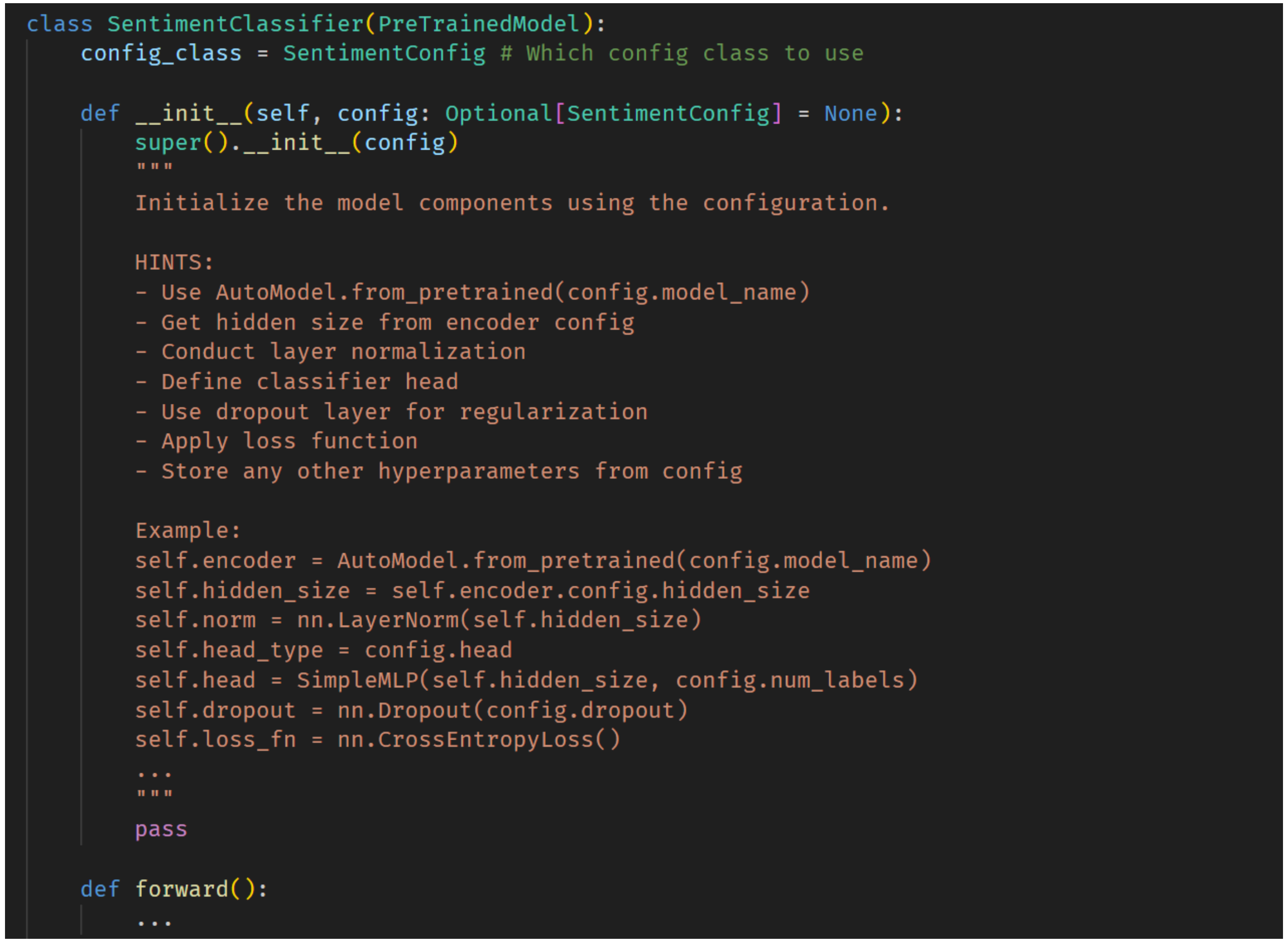
Model Configuration



```python
class SentimentConfig(PretrainedConfig):
    model_type = "..."  # describe this model type

    def __init__(
        self,
        model_name="...", # name of pre-trained model backbone
        num_labels=3,     # number of output classes (Negative, Neutral, Positive)
        head="mlp",       # classifier head
                          # other hyperparameters
        **kwargs,
    ):
        # Always call the parent class initializer first
        super().__init__(**kwargs)
        '''
        Save all hyperparameters to self

        Example:
        self.model_name = model_name
        self.num_labels = num_labels
        self.head = head
        ...
        self.other_hyperparam = other_hyperparam

        These attributes will be automatically saved in config.json
        when you call `config.save_pretrained("./path")`.
        '''
        pass
```

Model Architecture:

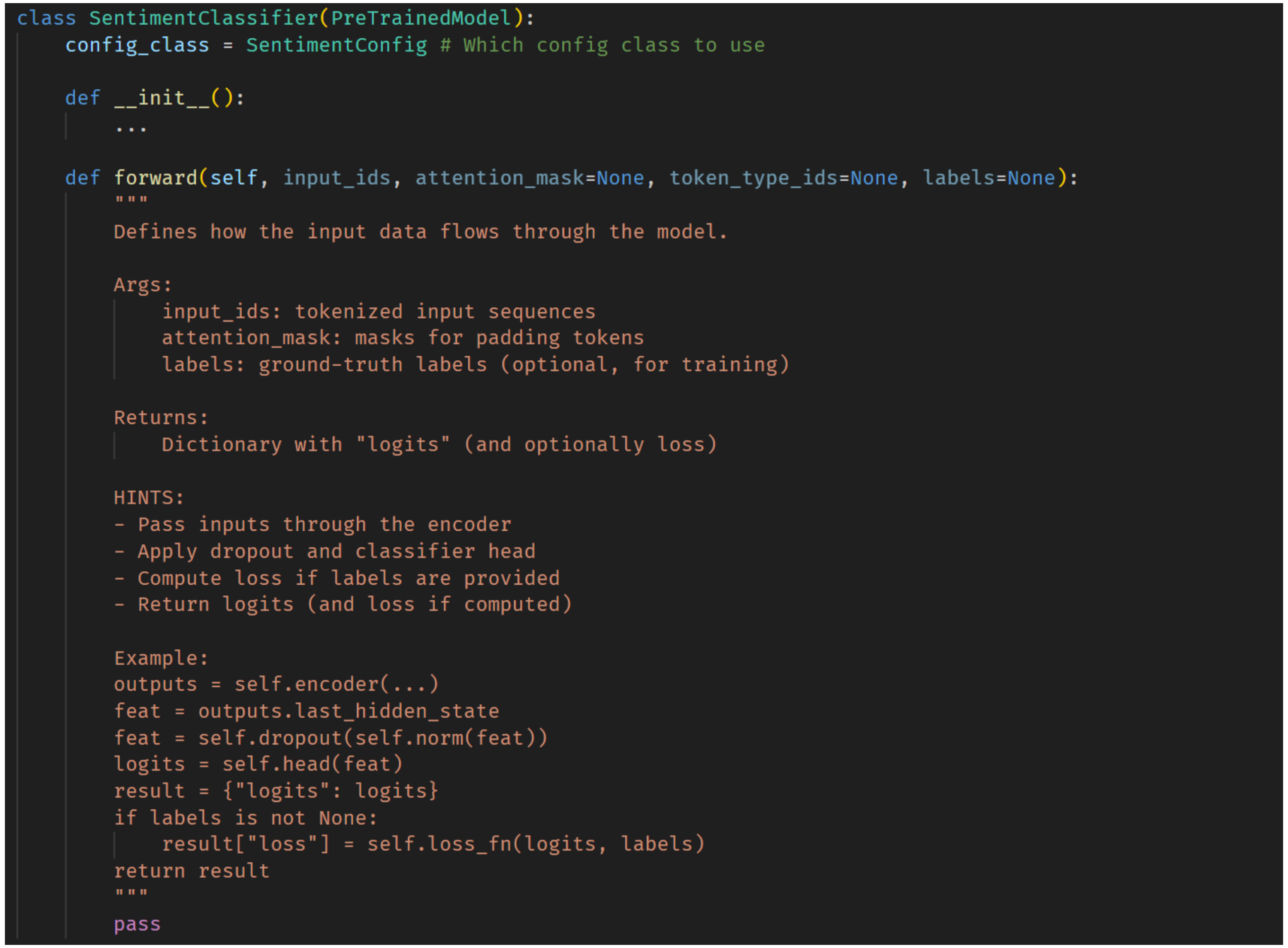
Classification Model



```python
class SentimentClassifier(PreTrainedModel):
    config_class = SentimentConfig # Which config class to use

    def __init__(self, config: Optional[SentimentConfig] = None):
        super().__init__(config)
        """
        Initialize the model components using the configuration.

        HINTS:
        - Use AutoModel.from_pretrained(config.model_name)
        - Get hidden size from encoder config
        - Conduct layer normalization
        - Define classifier head
        - Use dropout layer for regularization
        - Apply loss function
        - Store any other hyperparameters from config

        Example:
        self.encoder = AutoModel.from_pretrained(config.model_name)
        self.hidden_size = self.encoder.config.hidden_size
        self.norm = nn.LayerNorm(self.hidden_size)
        self.head_type = config.head
        self.head = SimpleMLP(self.hidden_size, config.num_labels)
        self.dropout = nn.Dropout(config.dropout)
        self.loss_fn = nn.CrossEntropyLoss()
        ...
        """
        pass

    def forward():
        ...
```

Model Architecture:

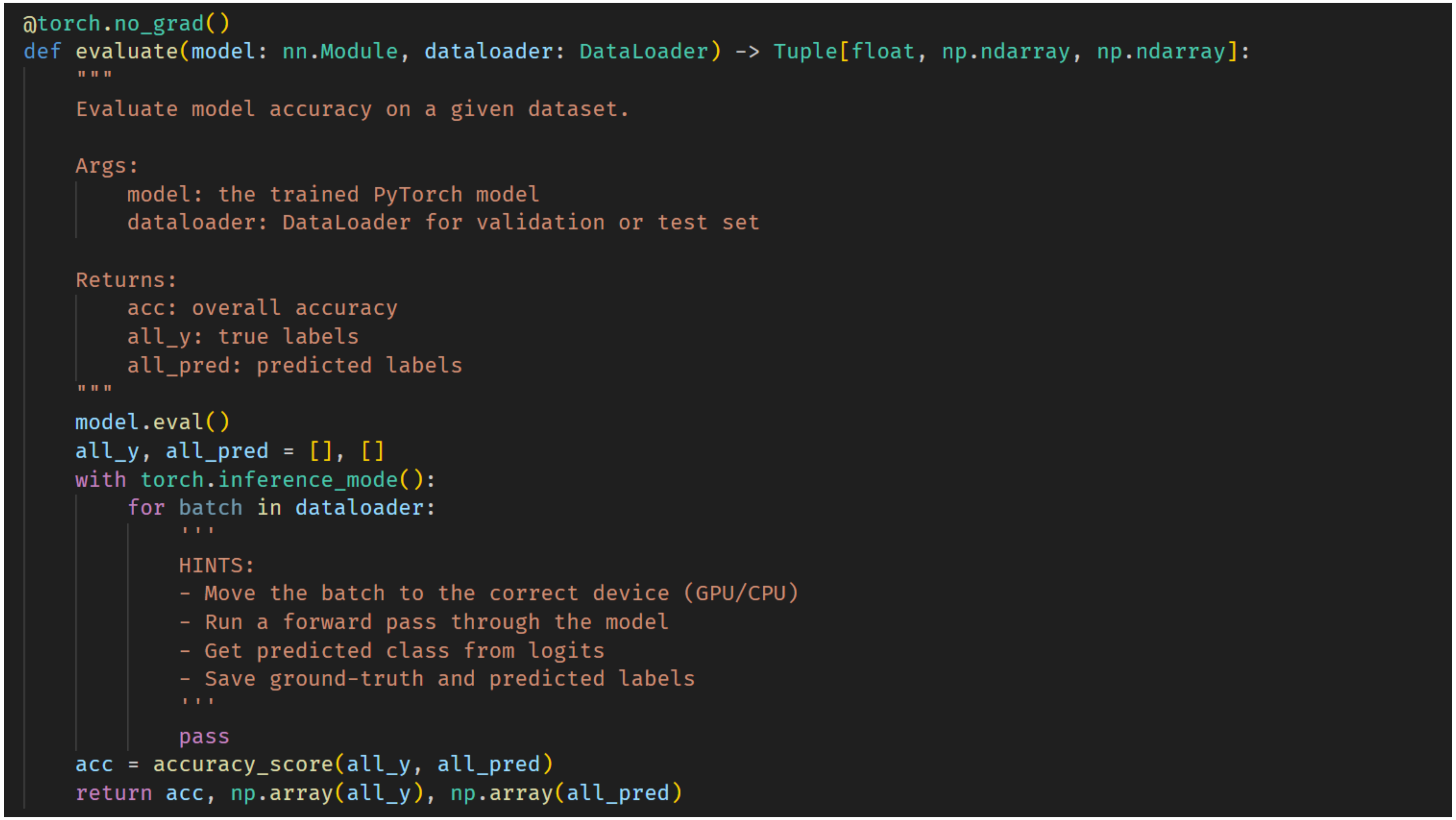
Classification Model



```python
class SentimentClassifier(PreTrainedModel):
    config_class = SentimentConfig # Which config class to use

    def __init__():
        ...

    def forward(self, input_ids, attention_mask=None, token_type_ids=None, labels=None):
        """
        Defines how the input data flows through the model.

        Args:
            input_ids: tokenized input sequences
            attention_mask: masks for padding tokens
            labels: ground-truth labels (optional, for training)

        Returns:
            Dictionary with "logits" (and optionally loss)

        HINTS:
        - Pass inputs through the encoder
        - Apply dropout and classifier head
        - Compute loss if labels are provided
        - Return logits (and loss if computed)

        Example:
        outputs = self.encoder(...)
        feat = outputs.last_hidden_state
        feat = self.dropout(self.norm(feat))
        logits = self.head(feat)
        result = {"logits": logits}
        if labels is not None:
            result["loss"] = self.loss_fn(logits, labels)
        return result
        """
        pass
```

Evaluation:

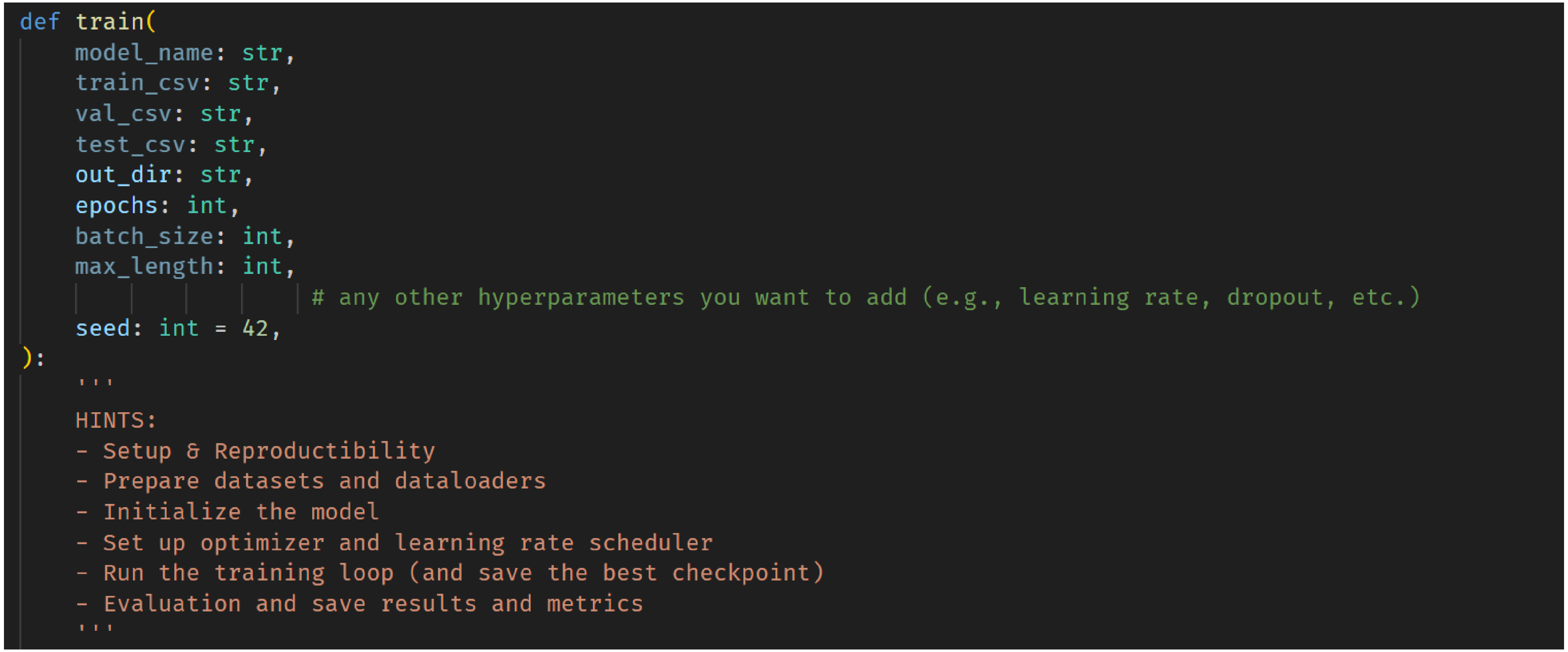
Model Evaluation



```python
@torch.no_grad()
def evaluate(model: nn.Module, dataloader: DataLoader) -> Tuple[float, np.ndarray, np.ndarray]:
    """
    Evaluate model accuracy on a given dataset.

    Args:
        model: the trained PyTorch model
        dataloader: DataLoader for validation or test set

    Returns:
        acc: overall accuracy
        all_y: true labels
        all_pred: predicted labels
    """
    model.eval()
    all_y, all_pred = [], []
    with torch.inference_mode():
        for batch in dataloader:
            '''
            HINTS:
            - Move the batch to the correct device (GPU/CPU)
            - Run a forward pass through the model
            - Get predicted class from logits
            - Save ground-truth and predicted labels
            '''
            pass
    acc = accuracy_score(all_y, all_pred)
    return acc, np.array(all_y), np.array(all_pred)
```

Training Loop:

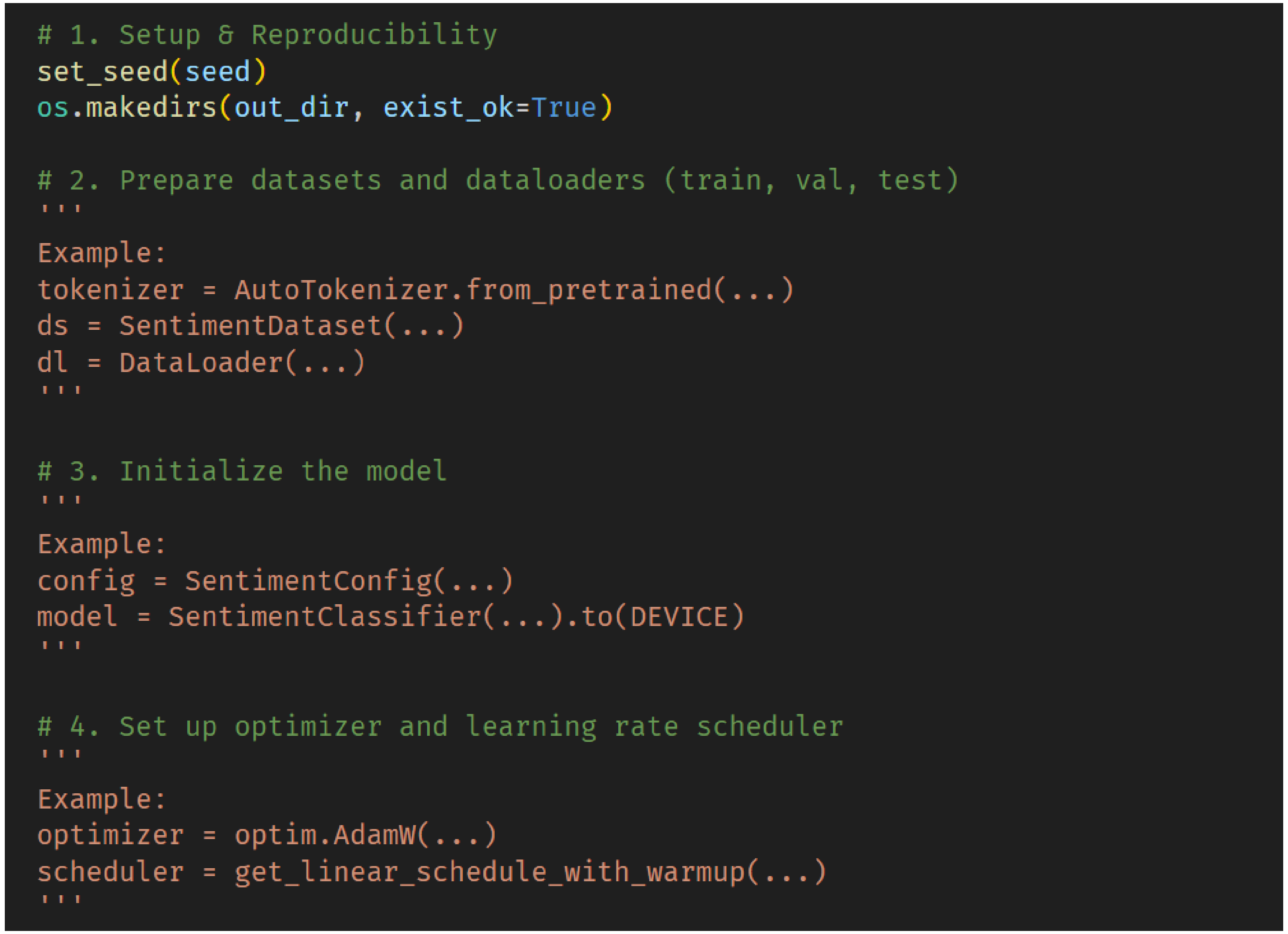
Train Function



```python
def train(
    model_name: str,
    train_csv: str,
    val_csv: str,
    test_csv: str,
    out_dir: str,
    epochs: int,
    batch_size: int,
    max_length: int,
                    # any other hyperparameters you want to add (e.g., learning rate, dropout, etc.)
    seed: int = 42,
):
    '''
    HINTS:
    - Setup & Reproductibility
    - Prepare datasets and dataloaders
    - Initialize the model
    - Set up optimizer and learning rate scheduler
    - Run the training loop (and save the best checkpoint)
    - Evaluation and save results and metrics
    '''
```

Training Loop:

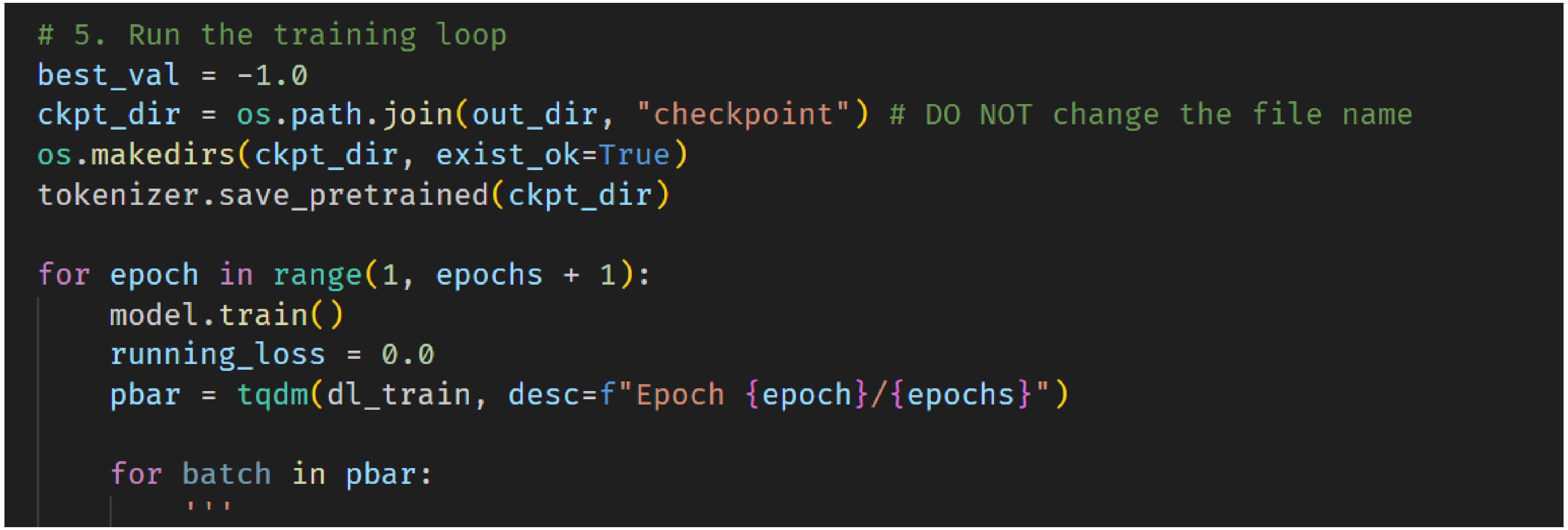
Train Function



```
# 1. Setup & Reproducibility
set_seed(seed)
os.makedirs(out_dir, exist_ok=True)

# 2. Prepare datasets and dataloaders (train, val, test)
'''
Example:
tokenizer = AutoTokenizer.from_pretrained(...)
ds = SentimentDataset(...)
dl = DataLoader(...)
'''


# 3. Initialize the model
'''
Example:
config = SentimentConfig(...)
model = SentimentClassifier(...).to(DEVICE)
'''


# 4. Set up optimizer and learning rate scheduler
'''
Example:
optimizer = optim.AdamW(...)
scheduler = get_linear_schedule_with_warmup(...)
'''
```

Training Loop:

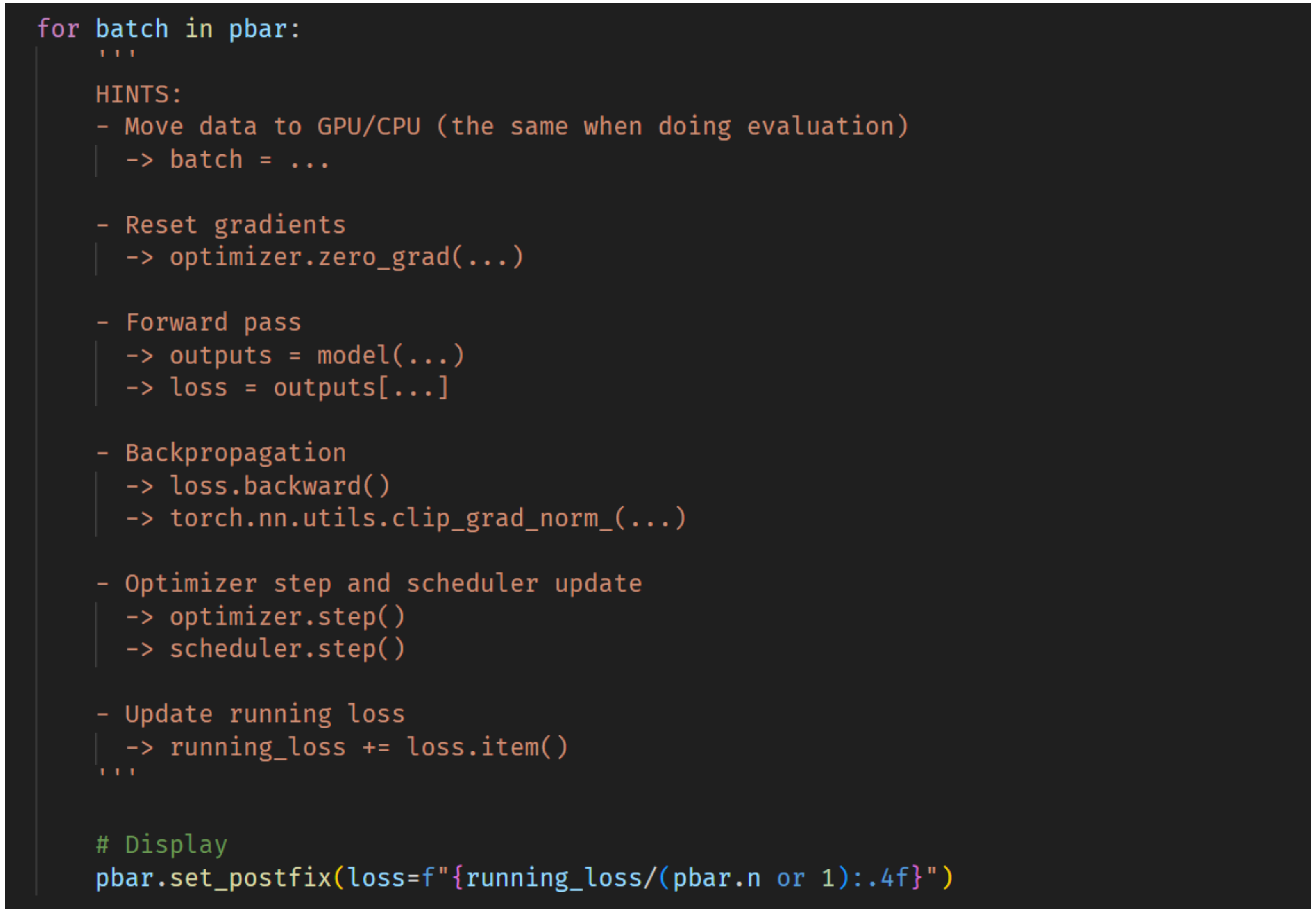
Train Function



```
# 5. Run the training loop
best_val = -1.0
ckpt_dir = os.path.join(out_dir, "checkpoint") # DO NOT change the file name
os.makedirs(ckpt_dir, exist_ok=True)
tokenizer.save_pretrained(ckpt_dir)

for epoch in range(1, epochs + 1):
    model.train()
    running_loss = 0.0
    pbar = tqdm(dl_train, desc=f"Epoch {epoch}/{epochs}")

    for batch in pbar:
        '''
```

Training Loop:

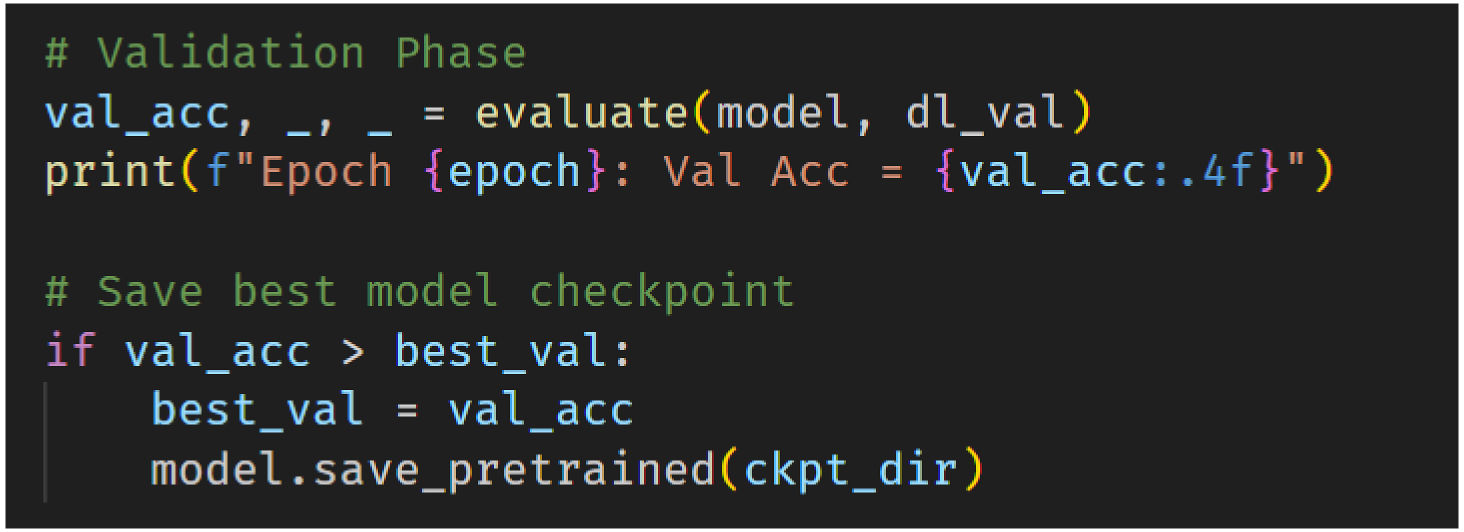
Train Function



```
for batch in pbar:
    '''
    HINTS:
    - Move data to GPU/CPU (the same when doing evaluation)
      -> batch = ...

    - Reset gradients
      -> optimizer.zero_grad(...)

    - Forward pass
      -> outputs = model(...)
      -> loss = outputs[...]

    - Backpropagation
      -> loss.backward()
      -> torch.nn.utils.clip_grad_norm_(...)

    - Optimizer step and scheduler update
      -> optimizer.step()
      -> scheduler.step()

    - Update running loss
      -> running_loss += loss.item()
    '''

    # Display
    pbar.set_postfix(loss=f"{running_loss/(pbar.n or 1):.4f}")
```

Training Loop:

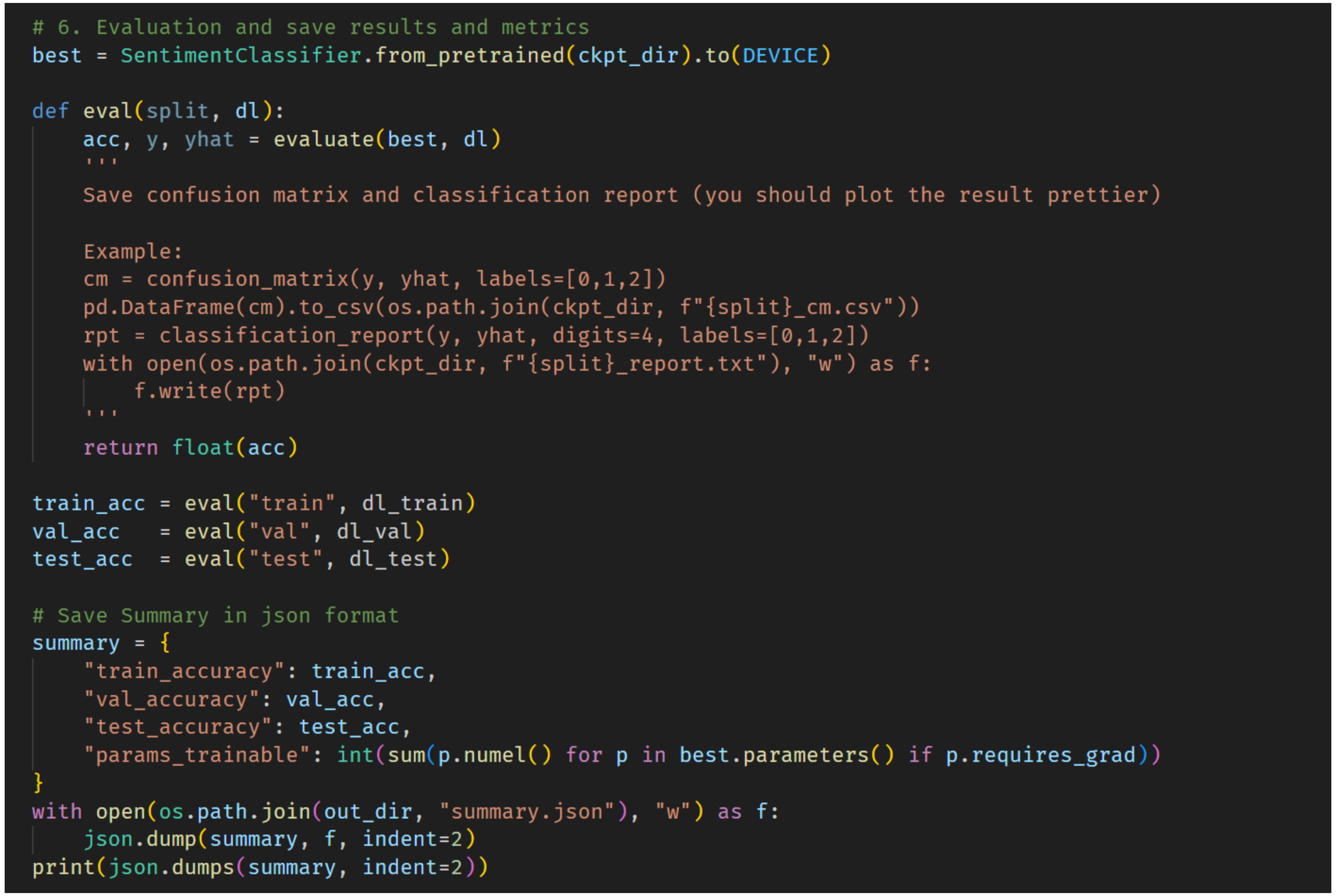
Train Function



```python
# Validation Phase
val_acc, _, _ = evaluate(model, dl_val)
print(f"Epoch {epoch}: Val Acc = {val_acc:.4f}")

# Save best model checkpoint
if val_acc > best_val:
    best_val = val_acc
    model.save_pretrained(ckpt_dir)
```

Training Loop:

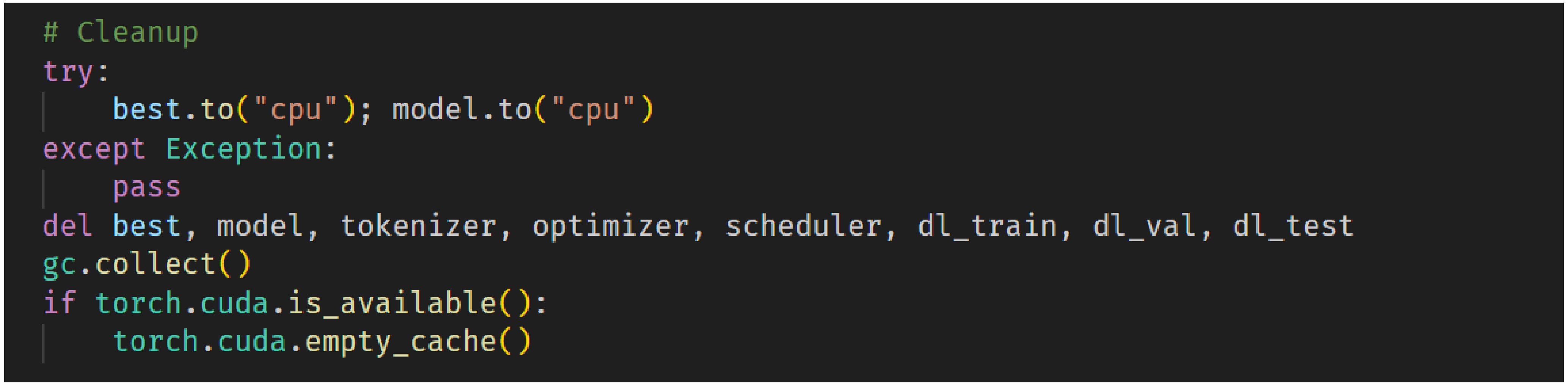
Train Function



```python
# 6. Evaluation and save results and metrics
best = SentimentClassifier.from_pretrained(ckpt_dir).to(DEVICE)

def eval(split, dl):
    acc, y, yhat = evaluate(best, dl)
    '''
    Save confusion matrix and classification report (you should plot the result prettier)

    Example:
    cm = confusion_matrix(y, yhat, labels=[0,1,2])
    pd.DataFrame(cm).to_csv(os.path.join(ckpt_dir, f"{split}_cm.csv"))
    rpt = classification_report(y, yhat, digits=4, labels=[0,1,2])
    with open(os.path.join(ckpt_dir, f"{split}_report.txt"), "w") as f:
        f.write(rpt)
    '''
    return float(acc)

train_acc = eval("train", dl_train)
val_acc   = eval("val", dl_val)
test_acc  = eval("test", dl_test)

# Save Summary in json format
summary = {
    "train_accuracy": train_acc,
    "val_accuracy": val_acc,
    "test_accuracy": test_acc,
    "params_trainable": int(sum(p.numel() for p in best.parameters() if p.requires_grad))
}
with open(os.path.join(out_dir, "summary.json"), "w") as f:
    json.dump(summary, f, indent=2)
print(json.dumps(summary, indent=2))
```

Training Loop:

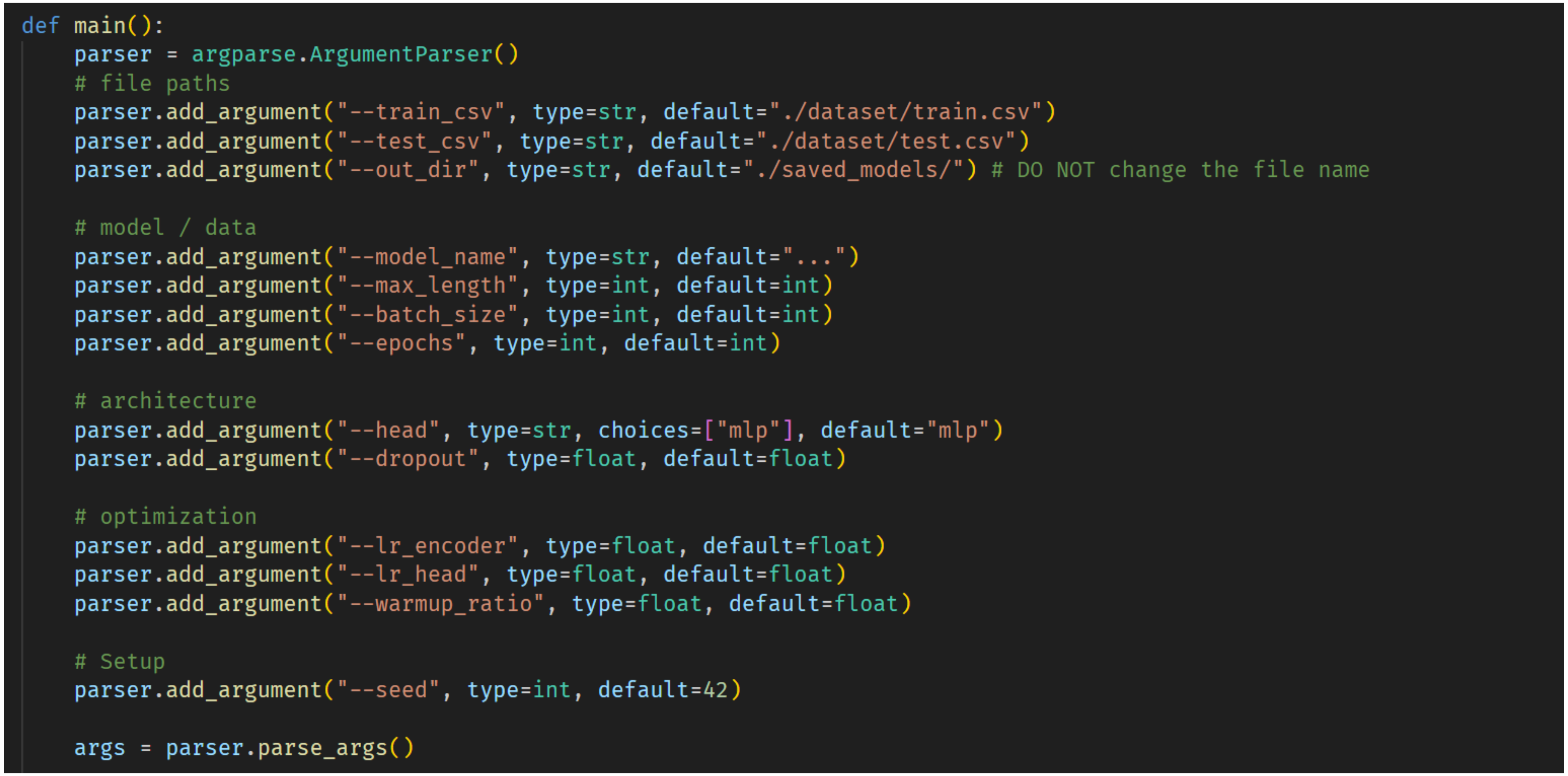
Train Function



```
# Cleanup
try:
    best.to("cpu"); model.to("cpu")
except Exception:
    pass
del best, model, tokenizer, optimizer, scheduler, dl_train, dl_val, dl_test
gc.collect()
if torch.cuda.is_available():
    torch.cuda.empty_cache()
```

Pipeline Wrapper:

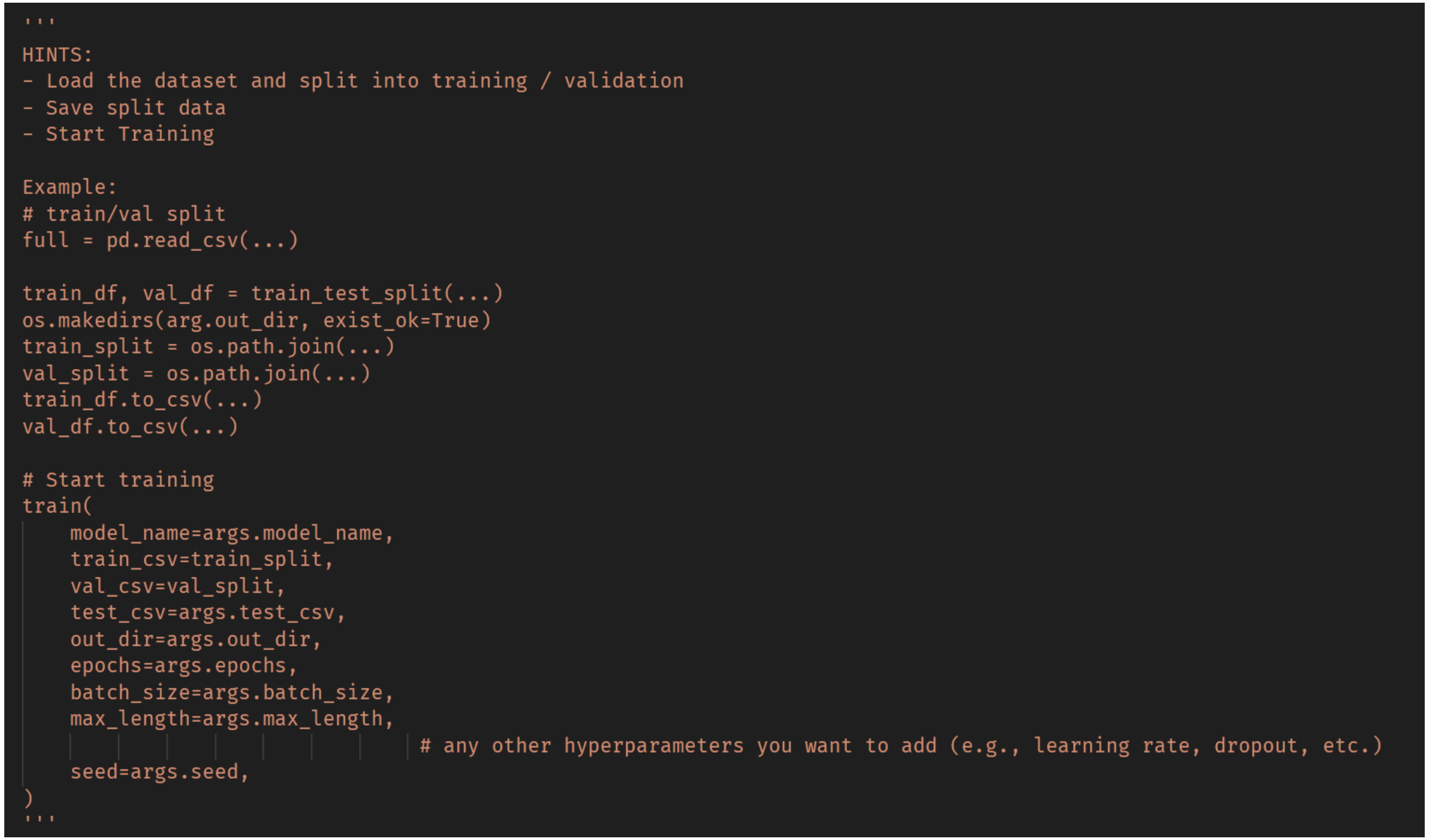
Main Function



```
def main():
    parser = argparse.ArgumentParser()
    # file paths
    parser.add_argument("--train_csv", type=str, default="./dataset/train.csv")
    parser.add_argument("--test_csv", type=str, default="./dataset/test.csv")
    parser.add_argument("--out_dir", type=str, default="./saved_models/") # DO NOT change the file name

    # model / data
    parser.add_argument("--model_name", type=str, default="...")
    parser.add_argument("--max_length", type=int, default=int)
    parser.add_argument("--batch_size", type=int, default=int)
    parser.add_argument("--epochs", type=int, default=int)

    # architecture
    parser.add_argument("--head", type=str, choices=["mlp"], default="mlp")
    parser.add_argument("--dropout", type=float, default=float)

    # optimization
    parser.add_argument("--lr_encoder", type=float, default=float)
    parser.add_argument("--lr_head", type=float, default=float)
    parser.add_argument("--warmup_ratio", type=float, default=float)

    # Setup
    parser.add_argument("--seed", type=int, default=42)

    args = parser.parse_args()
```

Pipeline Wrapper:

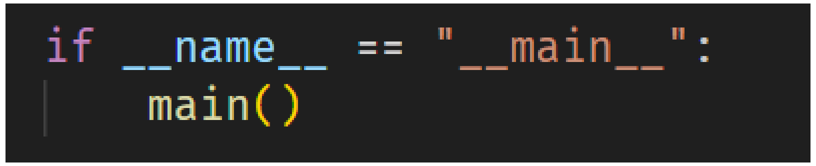
Main Function



```
'''
HINTS:
- Load the dataset and split into training / validation
- Save split data
- Start Training

Example:
# train/val split
full = pd.read_csv(...)

train_df, val_df = train_test_split(...)
os.makedirs(arg.out_dir, exist_ok=True)
train_split = os.path.join(...)
val_split = os.path.join(...)
train_df.to_csv(...)
val_df.to_csv(...)

# Start training
train(
    model_name=args.model_name,
    train_csv=train_split,
    val_csv=val_split,
    test_csv=args.test_csv,
    out_dir=args.out_dir,
    epochs=args.epochs,
    batch_size=args.batch_size,
    max_length=args.max_length,
                            # any other hyperparameters you want to add (e.g., learning rate, dropout, etc.)
    seed=args.seed,
)
'''
```

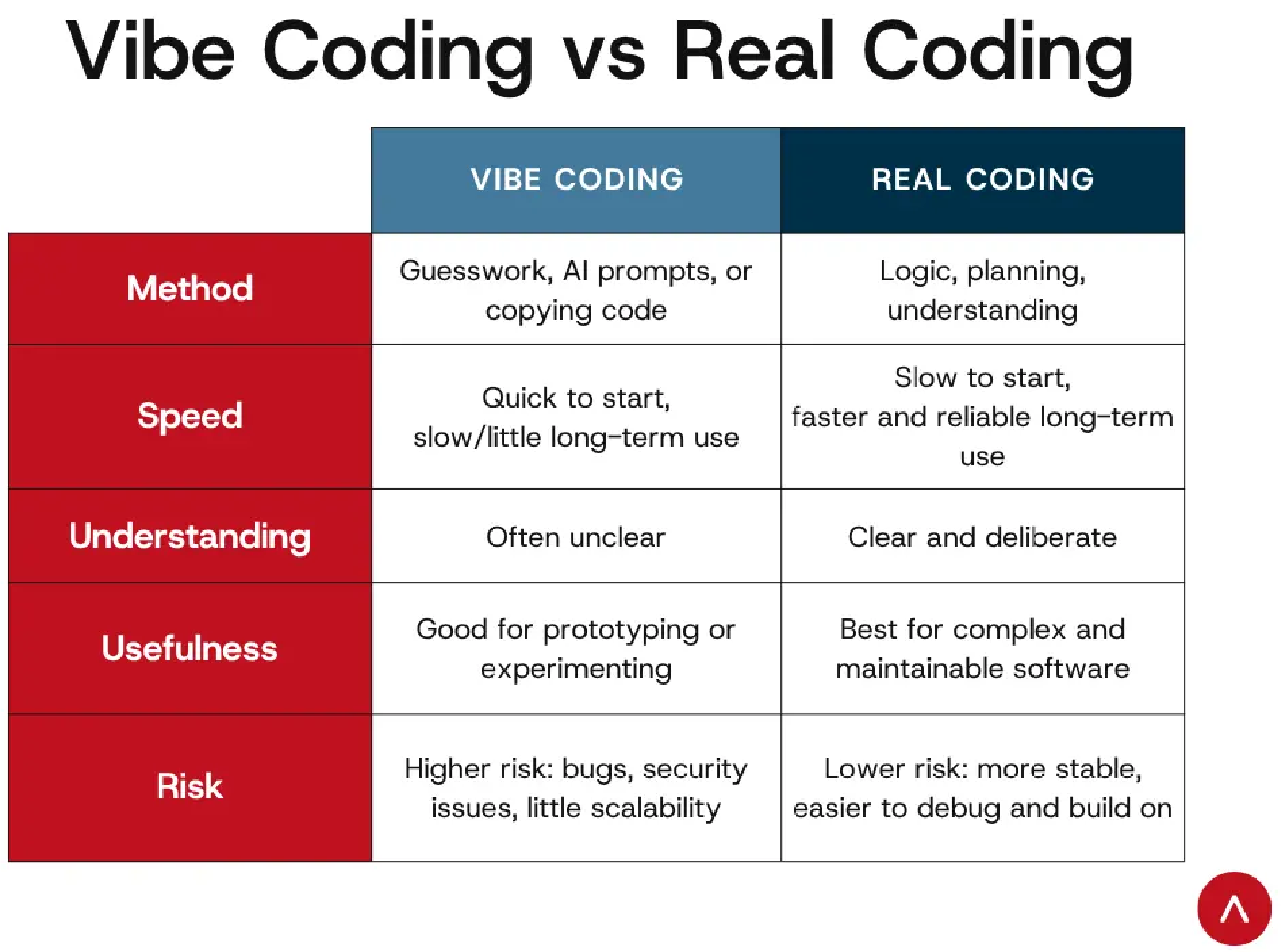
Pipeline Wrapper:

Main Function



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

Supplementary:



Vibe Coding vs Real Coding

	VIBE CODING	REAL CODING
Method	Guesswork, AI prompts, or copying code	Logic, planning, understanding
Speed	Quick to start, slow/little long-term use	Slow to start, faster and reliable long-term use
Understanding	Often unclear	Clear and deliberate
Usefulness	Good for prototyping or experimenting	Best for complex and maintainable software
Risk	Higher risk: bugs, security issues, little scalability	Lower risk: more stable, easier to debug and build on

Ref: https://www.northcoders.com/blog/what-is-vibe-coding-and-can-it-replace-traditional-coding/

Supplementary:

Need practical demonstration?

Contact & Information:

- Please post your question on the E3 forum.
- [TA] Cheng-Tsung Lee (李承璁): ctlee.ee14@nycu.edu.tw
- [TA hours] 11:00-13:00 Tue. ED-716 (Please make an appointment by email first)