Design and Implementation of HTML Code Generator using Large Language Model

CSN-352: COMPILER DESIGN

GROUP ID: **30**

NAME	ENR NO.
SHAMBHOOLAL NARWARIA	21114095
ALHAN CHARAN BESHRA	21114011
ABHISHEK RAJ	21114004

Under The Guidance Of:

Prof. Sudip Roy

TA: Anushka Joshi

Computer Science and Engineering Department

CP-2 Final Report

April 2024

Department of Computer Science and Engineering
Indian Institute of Technology Roorkee

Problem Statement

Write an HTML Code Generator that uses a Large Language Model (LLM), taking natural language prompts as input and giving output as an HTML Code.

Relation to Compiler Design

Lexical Analysis:

- The LLM system needs to parse the user's natural language prompt into relevant keywords and phrases associated with HTML elements and attributes.
- This process is similar to lexical analysis in compilers, where the input source code is broken down into tokens.

Syntax Analysis:

- The system would need to understand the structure and relationships between these elements, ensuring a valid HTML document is generated following the rules of HTML Syntax.
- This step is analogous to syntax analysis in compilers, where the parser checks the syntax of the source code.

Semantic Analysis:

- A level of semantic analysis is needed to ensure the elements and attributes are used correctly.
- This can be compared to compiler semantic analysis, which checks for correctly using variables and types.

Code Generation:

- The LLM generates the corresponding HTML code once the input is parsed and analyzed.
- This step is similar to the code generation phase in compilers, where the intermediate representation of the source code is translated into machine code.

Optimization:

- Compiler Design involves code optimization techniques to improve the efficiency and performance of the generated code.
- In this system, optimization techniques are also applied to improve the quality and efficiency of the generated HTML code.
- For example, removing redundant attributes or elements can improve the efficiency and readability of the HTML code.

Error Handling:

- Compiler design includes error handling mechanisms to detect and report errors in the input code.
- Similarly, the LLM System may handle errors in the input prompts to ensure accurate HTML generation.

Approach/Implementation

Choice of Model –

- The Falcon 7B model, a large language model containing a massive corpus of text data, is pre-trained on a large dataset, giving it a strong foundation for understanding natural language descriptions of HTML layouts and components.
- This model is selected for its ability to understand and generate human-like text.

Data Preprocessing –

- The dataset containing natural language descriptions of HTML layouts and components is pre-processed to clean HTML code, remove noise, and concatenate relevant text fields.

Tokenization –

- The text data is tokenized using the model's tokenizer.
- Tokenization breaks the input text into individual tokens, which are the basic units of input for the model.

Training Process –

- Model learns the syntactic structure and patterns in the input data.
- It optimizes the model's parameters to improve its understanding of the relationships between natural language descriptions and HTML code.

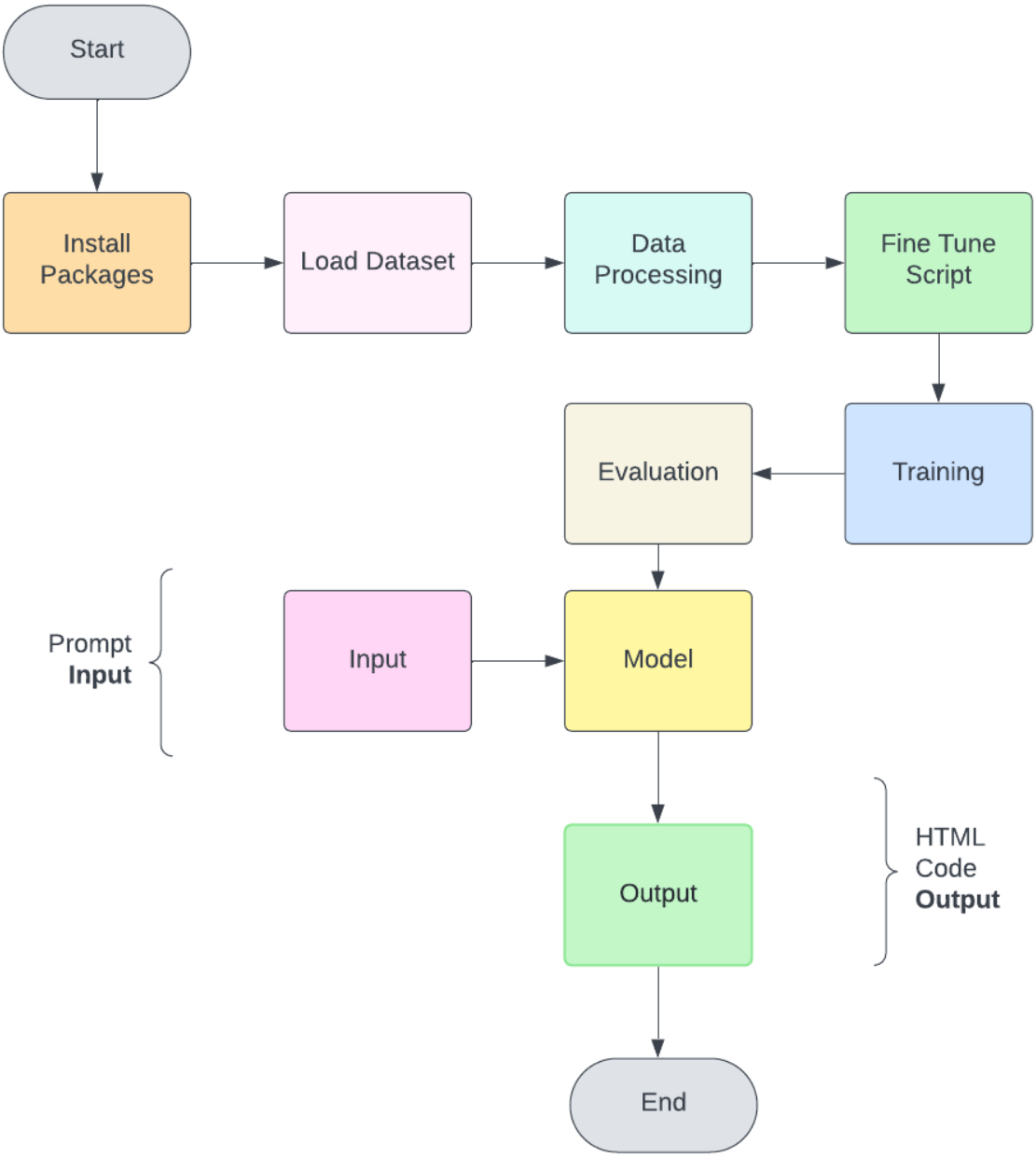
Evaluation –

- Correctness and meaningfulness of the generated output are assessed.
- It ensures the generated HTML code accurately reflects the intended layout and components described in the input text.

Code Generation –

- Once the model is trained and evaluated, it generates HTML code from natural language prompts.
- This code generation process involves feeding a natural language prompt to the model and receiving the corresponding HTML code as output.

Project Flow Chart



Code Implementation

Load Dataset

```
dataset = load_dataset("ttbui/html_alpaca")
print(dataset)
```

Data Preprocessing

```
from bs4 import BeautifulSoup
html_data = dataset['train']
# Function to clean and validate HTML
def clean_html(html):
    try:
        soup = BeautifulSoup(html, 'html5lib')  # Using html5lib for lenient parsing
        cleaned_html = soup.prettify()
    except Exception as e:
        print(f"An error occurred while cleaning HTML: {e}")
        cleaned_html = html  # Keep original if error occurs
    return cleaned_html

# Function to remove script and style tags
def remove_noise(html):
    soup = BeautifulSoup(html, 'html5lib')
    for script_or_style in soup(['script', 'style']):
        script_or_style.decompose()
    return soup.prettify()

def preprocess_html(example):
    example['output'] = clean_html(example['output'])
    example['output'] = remove_noise(example['output'])
    return example

# Apply preprocessing
my_dataset = html_data.map(preprocess_html)
for i in range(5):  # Adjust the range as needed
    print()
    print(f"Original HTML:\n{html_data['output'][i]}")
    print()
    print(f"Cleaned HTML:\n{my_dataset['output'][i]}")
```

- The code snippet demonstrates how to preprocess HTML data.
- It defines functions to clean and remove noise from HTML content and applies these functions to a dataset containing HTML data.
- The preprocessed HTML content is stored in a new dataset, and the code prints the original and cleaned HTML.

Splitting the Dataset:

```
from sklearn.model_selection import train_test_split

train_test_val_split = dataset["train"].train_test_split(test_size=0.3, seed=42)
test_val_split = train_test_val_split['test'].train_test_split(test_size=0.5, seed=42)

# Assign the splits to variables
train_dataset = train_test_val_split['train']
test_dataset = test_val_split['test']
val_dataset = test_val_split['train']
```

- The code snippet demonstrates how to perform a hierarchical split of a dataset into training, testing, and validation sets using the train_test_split function from sci-kit-learn.
- The dataset is initially split into a training set and a combined test-validation set, which is further split into separate test and validation sets.
- The resulting splits are assigned to variables for further training and evaluation of machine learning models.

Concatenating Text Field:

```
def concat_fields(example):

    if example['input'] is not None:
        return {'text': example['instruction'] + ' ' + example['input']}
    else:
        return {'text': example['instruction']}

# Apply this function to the dataset
train_dataset = train_dataset.map(concat_fields)
val_dataset = val_dataset.map(concat_fields)
test_dataset = test_dataset.map(concat_fields)

print(train_dataset)
print(test_dataset)
```

```
print(val_dataset)
```

- The code snippet demonstrates how to concatenate specific fields (instruction and input) from each example in a dataset and store the result in a new text field.
- The concat_fields function is applied to the training, validation, and testing datasets using the map method, resulting in modified datasets with concatenated text fields.
- The modified datasets are then printed to inspect the changes and verify that the concatenation was performed correctly.

Load the Pre-trained Sharded Falcon-7b Model:

```
model_name = "ybelkada/falcon-7b-sharded-bf16"

bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_use_double_quant=True
    bnb_4bit_compute_dtype=torch.bfloat16,)
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    quantization_config=bnb_config,
    device_map="auto",
    trust_remote_code=True)
```

- The code snippet demonstrates loading a pre-trained sharded falcon-7b model using the Hugging Face Transformers library with specific quantization configurations.
- The code applies a BitsAndBytesConfig object with 4-bit quantization settings to the model and loads the model with these configurations.
- The loaded model is stored in the model variable and can be used for various natural language processing tasks, such as causal language modeling, with optimized memory and computational efficiency due to the applied quantization.

Tokenization:

```
tokenizer = AutoTokenizer.from_pretrained(model_name, trust_remote_code=True)

tokenizer.pad_token = tokenizer.eos_token
```

- The code snippet demonstrates how to load a tokenizer corresponding to a pre-trained sharded falcon-7b model using the Hugging Face Transformers library.
- Additionally, it sets the pad_token of the tokenizer to its eos_token to leverage the benefits of end-of-sequence treatment and prevent the model from being influenced by padding tokens during training and text generation.

Fine Tune Script

```
model = prepare_model_for_kbit_training(model)

lora_alpha = 32 # scaling factor for the weight matrices-->  scaling the influence of the learnt params during training
lora_dropout = 0.05 # dropout probability of the LoRA layers
lora_rank = 32 # dimension of the low-rank matrices--> low dim means lower complexity and fewer params

peft_config = LoraConfig(
    lora_alpha=lora_alpha,
    lora_dropout=lora_dropout,
    r=lora_rank,
    bias="none",  # setting to 'none' for only training weight params instead of biases
    task_type="CAUSAL_LM",
    target_modules=[        # Setting names of layers where we want to apply lora to
        "query_key_value",
        "dense",
        "dense_h_to_4h",
        "dense_4h_to_h",
    ]
)
peft_model = get_peft_model(model, peft_config)
```

- The code snippet demonstrates the preparation and configuration of a model for training with k-bit quantization using LoRA layers.
- The original model is prepared for k-bit training, and a new LoRA-integrated model is obtained based on the specified LoRA configuration. This LoRA-integrated model is designed for causal language modeling tasks.
- It has LoRA layers applied to specific target modules, making it suitable for efficient and optimized training with k-bit quantization and low-rank adaptation.

Setting up Training-arguments:

```
output_dir = "/content/gdrive/MyDrive/LLM/falcon-7b-sharded-bf16-finetuned-html-code-generation"

per_device_train_batch_size = 2
gradient_accumulation_steps = 2
optim = "paged_adamw_32bit
save_strategy="steps"
save_steps = 20
logging_steps = 20
learning_rate = 2e-4
max_grad_norm = 0.3
max_steps = 320
warmup_ratio = 0.03
lr_scheduler_type = "cosine"

training_arguments = TrainingArguments(
    output_dir=output_dir,
    per_device_train_batch_size=per_device_train_batch_size,
    gradient_accumulation_steps=gradient_accumulation_steps,
    optim=optim,
    save_steps=save_steps,
    logging_steps=logging_steps,
    learning_rate=learning_rate,
    bf16=False,
    max_grad_norm=max_grad_norm,
    max_steps=max_steps,
    warmup_ratio=warmup_ratio,
    group_by_length=True,
    lr_scheduler_type=lr_scheduler_type,
    push_to_hub= True,
    tf32=False
    evaluation_strategy="steps",
    eval_steps=20
    load_best_model_at_end=True,)
```

- The provided code snippet defines a comprehensive set of training hyperparameters and settings using the Training-Arguments object for training a model with specific configurations.
  These settings include batch size, optimizer type, learning rate, gradient accumulation steps, checkpoint saving and logging strategies, and evaluation

settings. This configuration prepares the model for efficient and optimized training with the specified hyperparameters and strategy.

Instantiate Trainer:

```
trainer = SFTTrainer(

    model=peft_model,
    train_dataset=train_dataset,
    peft_config=peft_config,
    dataset_text_field="text",  # The concatenated field
    max_seq_length=1024,
    tokenizer=tokenizer,
    args=training_arguments,
    eval_dataset=val_dataset,
    callbacks=[EarlyStoppingCallback(early_stopping_patience=3)])
for name, module in trainer.model.named_modules():
    if "norm" in name:
        module = module.to(torch.float32)
```

- The code snippet initializes an SFTTrainer for training a LoRA-integrated model with k-bit quantization.
- The trainer is configured with the model, datasets, tokenizer, training arguments, and an EarlyStoppingCallback for early stopping during training.
- Additionally, the code adjusts the model's data type of normalization layers to torch.float32 to accommodate reduced precision during training with k-bit quantization.

Loading the Original and The Fine-Tuned Model:

```
model_name = "ybelkada/falcon-7b-sharded-bf16"

bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_use_double_quant=True,
    bnb_4bit_compute_dtype=torch.float16,
)

model = AutoModelForCausalLM.from_pretrained(
    model_name,
    quantization_config=bnb_config,
    device_map="auto",
    trust_remote_code=True,
)
```

```
tokenizer = AutoTokenizer.from_pretrained(model_name, trust_remote_code=True)
tokenizer.pad_token = tokenizer.eos_token
```

- The provided code snippet loads an original pre-trained model (falcon-7b-sharded-bf16) and its tokenizer using the AutoModelForCausalLM.from_pretrained() and AutoTokenizer.from_pretrained() methods, respectively.
- The model has specific bits and bytes quantization configurations, including 4-bit quantization, double quantization, and float16 computation datatype.
- The tokenizer's pad_token is set to the eos_token, which is beneficial for sequence padding and inference.

Evaluation:

```
from datasets import load_metric

bleu_metric = load_metric("bleu")

# Define the evaluation function
def evaluate_model(model, tokenizer, test_dataset):
    model.eval()
    predictions, references = [], []

    for example in test_dataset:
        inputs = tokenizer(example["text"], return_tensors="pt", padding=True,
truncation=True, max_length=512)
        if "token_type_ids" in inputs:
            inputs.pop("token_type_ids")
        outputs = model.generate(**inputs, max_length=32)
        generated_html = tokenizer.decode(outputs[0], skip_special_tokens=True)
        predictions.append(tokenizer.tokenize(generated_html))
        references.append([tokenizer.tokenize(example["output"])])
    return bleu_metric.compute(predictions=predictions, references=references)
results = evaluate_model(peft_model, peft_tokenizer, test_dataset)
print("BLEU Score:", results['bleu'])
```

- The code snippet defines an evaluation function to assess the performance of a fine-tuned PEFT model using the BLEU metric.
- The function tokenizes the input text, generates HTML output using the model, and computes the BLEU score based on the test dataset's generated and expected HTML sequences. Finally, the BLEU score is printed to evaluate the model's performance.

- The code snippet defines several evaluation metrics and a combined evaluation function to assess the quality and accuracy of HTML outputs generated by a model. It evaluates the model's performance on a test dataset and computes average scores for each metric to provide an overall evaluation result. The evaluation metrics include relevant tags accuracy, structure accuracy, completeness, styling, and scripting elements presence, semantic accuracy, and overall score.

Inference Model

```
def generate_html_code(prompt, model, peft_model, tokenizer, peft_tokenizer):

    dashline = "-" * 50

    print(f'Prompt: {prompt}')
    print(dashline)
    encoding = tokenizer(prompt, return_tensors="pt")
    outputs = model.generate(input_ids=encoding["input_ids"],
attention_mask=encoding["attention_mask"],
                             max_length=256, pad_token_id=tokenizer.eos_token_id,
                             eos_token_id=tokenizer.eos_token_id, temperature=0.7, top_p=0.9)
    original_model_html = tokenizer.decode(outputs[0], skip_special_tokens=True)
    print(f'Original Model HTML Output:\n{original_model_html}')
    print(dashline)
    peft_encoding = peft_tokenizer(prompt, return_tensors="pt")
    peft_outputs = peft_model.generate(input_ids=peft_encoding["input_ids"],
attention_mask=peft_encoding["attention_mask"],
                                       max_length=256, pad_token_id=peft_tokenizer.eos_token_id,
                                       eos_token_id=peft_tokenizer.eos_token_id, temperature=0.7,
top_p=0.9)
    peft_model_html = peft_tokenizer.decode(peft_outputs[0], skip_special_tokens=True)
    print(f'PEFT Model HTML Output:\n{peft_model_html}')
    print(dashline)

prompt = "Generate HTML code for sign up page with login options"
prompt = "Create an HTML page that includes a navigation bar with links to 'Home', 'About',
'Services', and 'Contact'. Below the navigation bar, add a hero section with a welcoming
message and a call-to-action button labeled 'Learn More'. Ensure the page is structured with a
header, main content area, and footer. The footer should contain copyright information and social
media links."
generate_html_code(prompt, model, peft_model, tokenizer, peft_tokenizer)
```

Function Steps –

1. **Print Prompt –**
   - Prints the prompt for which the HTML code needs to be generated.

2. **Generate HTML with Original Model –**
   - Tokenizes the prompt using the original tokenizer.
   - Generates HTML code using the original model based on the tokenized input.
   - Decodes the generated output into readable HTML code.
   - Prints the HTML output from the original model.

3. **Generate HTML with PEFT Model –**
   - Tokenizes the prompt using the PEFT tokenizer.
   - Generates HTML code using the fine-tuned PEFT model based on the tokenized input.
   - Decodes the generated output into readable HTML code.
   - Prints the HTML output from the PEFT model.

4. **Print Dashline –**
   Prints a dashed line separator to differentiate between the original and PEFT models' outputs.

Code Execution

The provided prompt is used as input to generate HTML code using the original and fine-tuned PEFT models. The generated HTML outputs from both models are printed for comparison.

Testcase 1

Prompt:

Create an HTML website with navigation, logo, and footer.

Output:

```
<!DOCTYPE html>
<html lng="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>My Website</title>
</head>

<body>
    <div class="navbar"> <!-- navigation links here --> </div>
    <div class="logo"> <!-- logo here --> </div>
    <div class="main-content"> <!-- page content here --> </div>
    <div class="footer"> <!-- footer here --> </div>
</body>

</html>
```

Testcase 2

Prompt:

Create an HTML page with a form to capture the user's name and age.

Output:

```
<!DOCTYPE html>
<html>
<head>
  <title>User Information Form</title>
</head>
<body>
  <h2>User Information Form</h2>
  <form action="user_information.php" method="post">
    <p>Name: <input type="text" name="name"></p>
    <p>Age: <input type="number" name="age"></p> <input type="submit" value="Submit">
  </form>
</body>
</html>
```

Testcase 3

Prompt:

Design an HTML page with a table of products and their prices.

Output:

```
<!DOCTYPE html>
<html>
<head>
  <title>Product Prices</title>
</head>
<body>
  <h1>Product Prices</h1>
  <table>
    <tr>
      <th>Product Name</th>
      <th>Price</th>
    </tr>
    <tr>
      <td>Laptop</td>
      <td>1000</td>
    </tr>
    <tr>
      <td>Keyboard</td>
      <td>50</td>
    </tr>
    <tr>
      <td>Monitor</td>
      <td>200</td>
    </tr>
    <tr>
      <td>Mouse</td>
      <td>30</td>
    </tr>
  </table>
</body>
</html>
```