July 1 2021

2022 Samsung AI 교육과정

Coding Practice

- KoGPT2 for text generation

Kyomin Jung

Department of Electrical and Computer Engineering
Seoul National University

About Today Class

- **Today's TA**
  - 양낙영 **(yny0506@snu.ac.kr)**
  - 곽희영 **(hykwak88@snu.ac.kr)**
  - 주성호 **(seonghojoo@snu.ac.kr)**

Tutorial highlights

- Reminder: Transformer and GPT

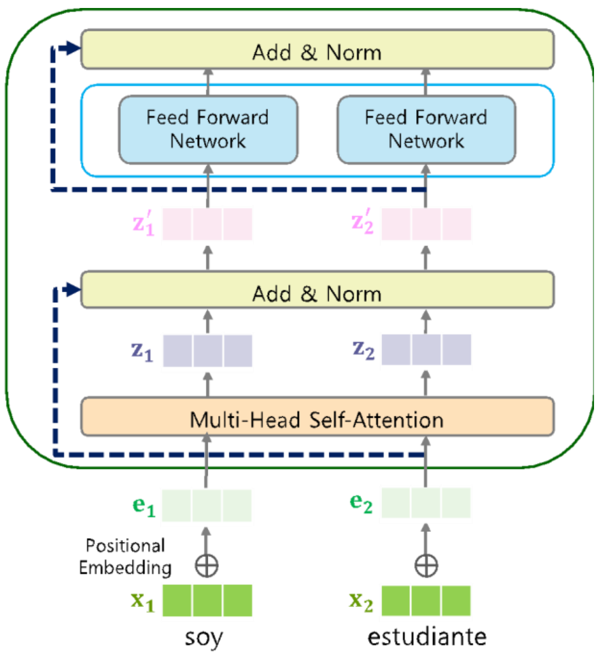
- Using the HuggingFace transformer
  - Tokenizer
  - Model
  - Using a pretrained model(KoGPT2)

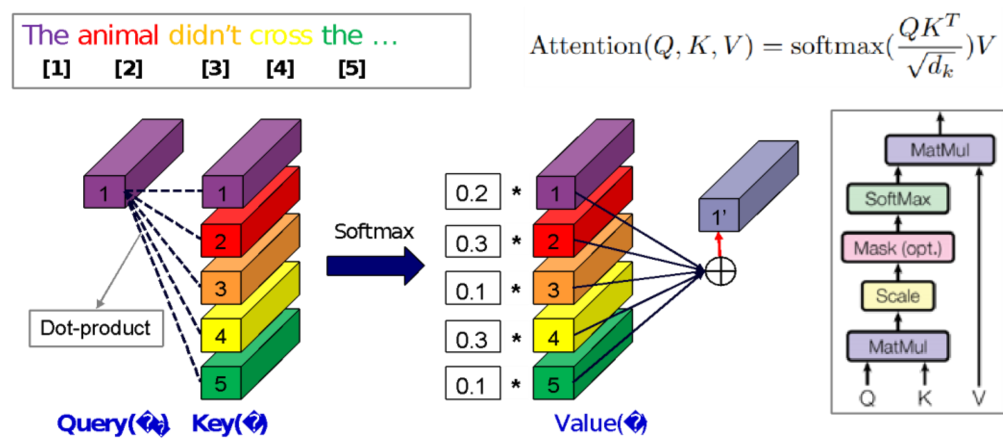
- Practice: text generation with GPT (on notebook)
  - Generating text with pretrained KoGPT2
  - Loading of the Naver Movie Review dataset
  - Fine-tuning the model for text generation with given dataset
  - Fine-tuning the model for text classification with given dataset

Reminder: Transformer

A. Vaswani et al., “Attention Is All You Need”, NIPS 2017

- Sequence is modeled by self-attention
- Can represent bidirectional context
- Originally an encoder-decoder structure

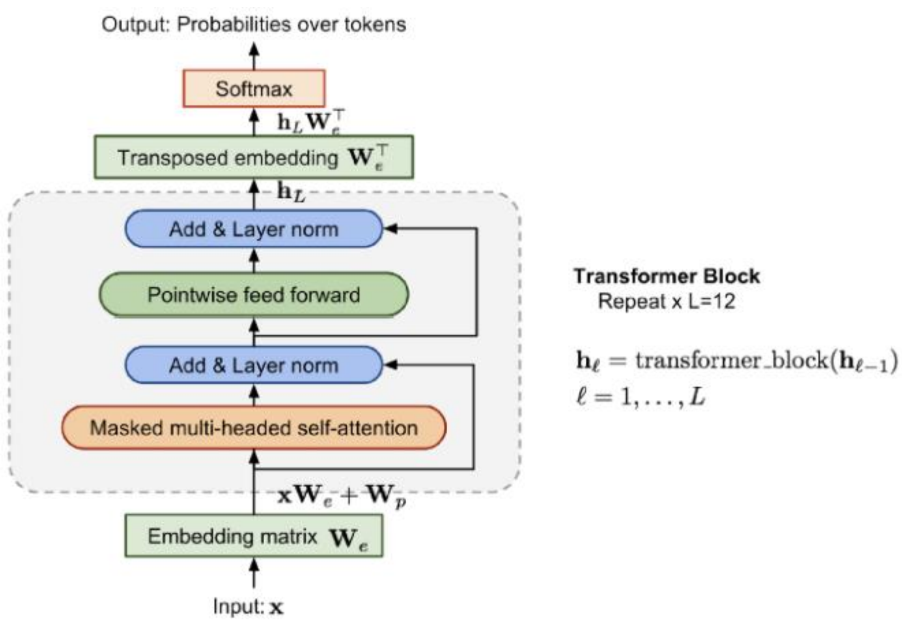




Reminder: GPT2

A. Radford et al., "Language Models are Unsupervised Multitask Learners", Open AI

- Language Model for Self-supervised representation learning
- Based on a Transformer decoder
- Trained from huge text corpus, by using Auto-regressive style word prediction objective
- Fine-tuned for downstream NLP tasks.



Using GPT2 from scratch?

- We need…
  - Dataset
  - Tokenizer (convert text into a series of numbers so the model can treat)
  - A Transformer model with extra layer(s) (for each task)
  - A set of pretrained GPT2 model parameter (available online)
  - Codes for fine-tuning
  - Codes for inference

- One can start from scratch, following the official GPT2 code
  - https://github.com/openai/gpt-2

Using GPT2 from scratch?

- **We need…**
  - **Dataset**
  - **Tokenizer (convert text into a series of numbers so the model can treat)**
  - **A Transformer model with extra layer(s) (for each task)**
  - **A set of pretrained GPT2 model parameter (available online)**
  - **Codes for fine-tuning**
  - **Codes for inference**

“Included in ”

HuggingFace Transformers

- The `transformers` package consist of
  - Tokenizers
  - Transformer-based language model architectures
    - BERT / GPT(2) / XLNet…
    - Encoder-decoder models like BART are also available
    - Efficient Transformer models like Reformer, Longformer are also available
  - Optimizers
  - High-level APIs
    - Trainer
    - Complete models like sentiment classifiers
  - Repository of pretrained models (tokenizers, …)

Tokenizer

- Text need to be converted into a sequence of numbers

- GPT2 uses subword based tokenization
  - Subword: sequence of character; smaller unit than word
    - Watching -> 'Watch' + 'ing'
  - Needs smaller vocabulary compared to word-based tokenization
  - Byte Pair Encoding (BPE; Sennrich et al.) / word-piece (Google)

- Each language model uses their unique vocabulary for tokenization; trained from the training data

Tokenizer

- **Tokenizer Implementation can be loaded and used with HuggingFace's tokenizer interface**

- **Tokenizer pre-trained with the Korean dataset is also available**

```
tokenizer =
PreTrainedTokenizerFast.from_pretrained(
                        "skt/kogpt2-base-v2")

sample_text = " 근육이 커지기 위해서는”

tokens = tokenizer.tokenize(sample_text)
token_ids = tokenizer.encode(sample_text)

print(f' Sentence: {sample_text}')
print(f'   Tokens: {tokens}')
print(f'Token IDs: {token_ids}')
```

The Model (GPT2Model)

- **HuggingFace has complete models and their pretrained parameters**

```
gpt2_model = GPT2Model.from_pretrained('skt/kogpt2-base-v2')
```

- **In case of GPT2, the model has 2 types of outputs**
  - **Sequence of the last hidden states**
  - **Sequence of the past hidden states**

```
hidden_states = gpt2_model(torch.tensor([token_ids]))

last_hidden_state = hidden_states[0]
print(last_hidden_state.shape)
```

"How to get Language Model output?"

The Model (GPT2LMHeadModel)

- **HuggingFace has complete LM models and their pretrained parameters**

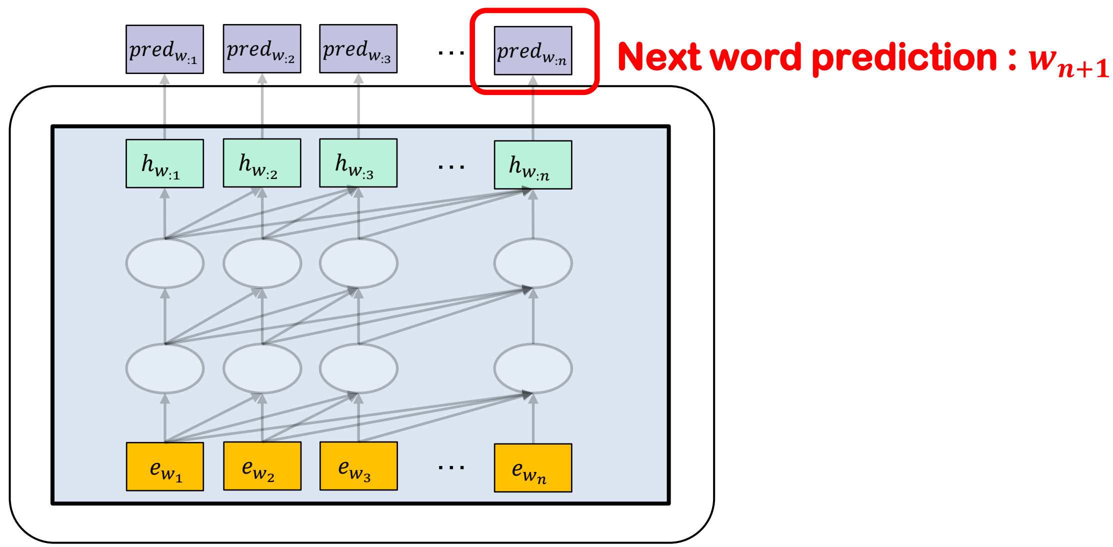
```
gpt2lm_model = GPT2LMHeadModel.from_pretrained('skt/kogpt2-base-v2')
```

- **GPT2LMHead has 2 types of outputs**
  - **Sequence of the next word predictions**
  - **Sequence of the past hidden states**

```
outputs = gpt2lm_model(torch.tensor([token_ids]))

next_word_predictions = outputs[0]
print(next_word_predictions.shape)
```

Next word Generation

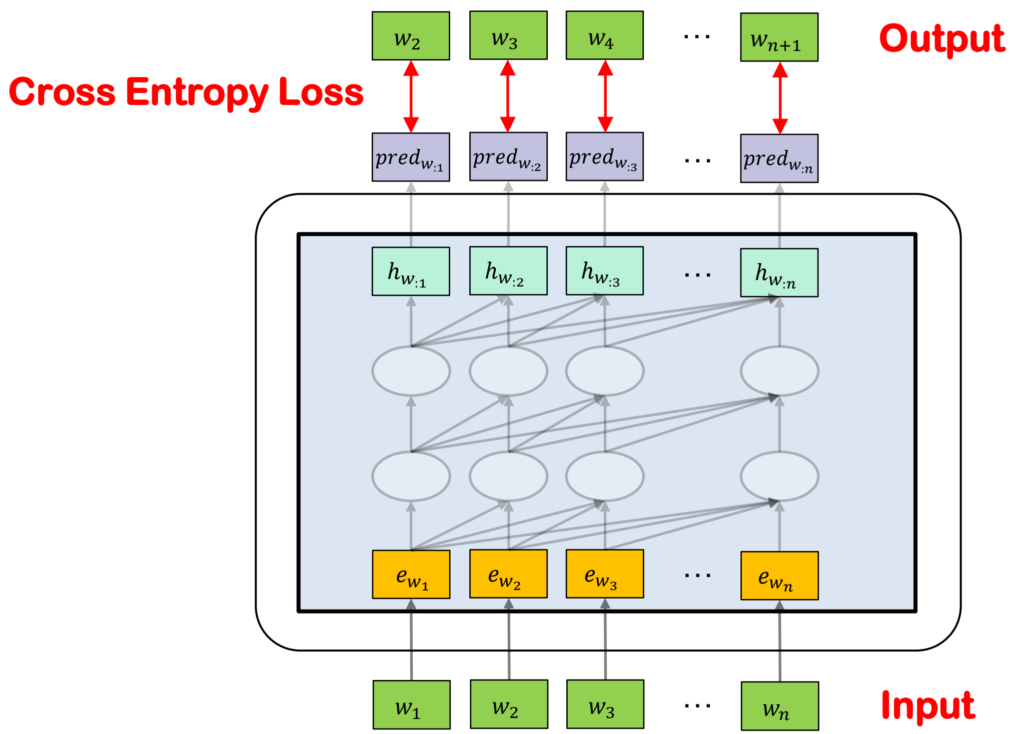
- **The last output corresponds to the next word prediction**




```
next_word_distribution = next_word_predictions[0, -1, :]
next_word_id = ??? // To-do
next_word = ??? // To-do
print(f' Next word: {next_word}')
```

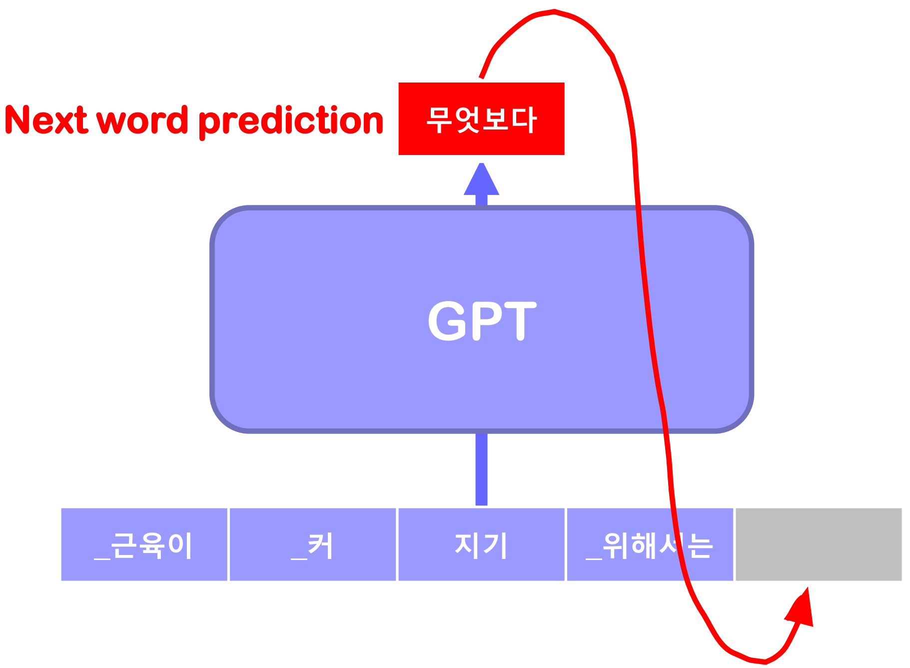
GPT Training

- We can train GPT using cross entropy loss
- We can understand the process of GPT Training as a classification problem



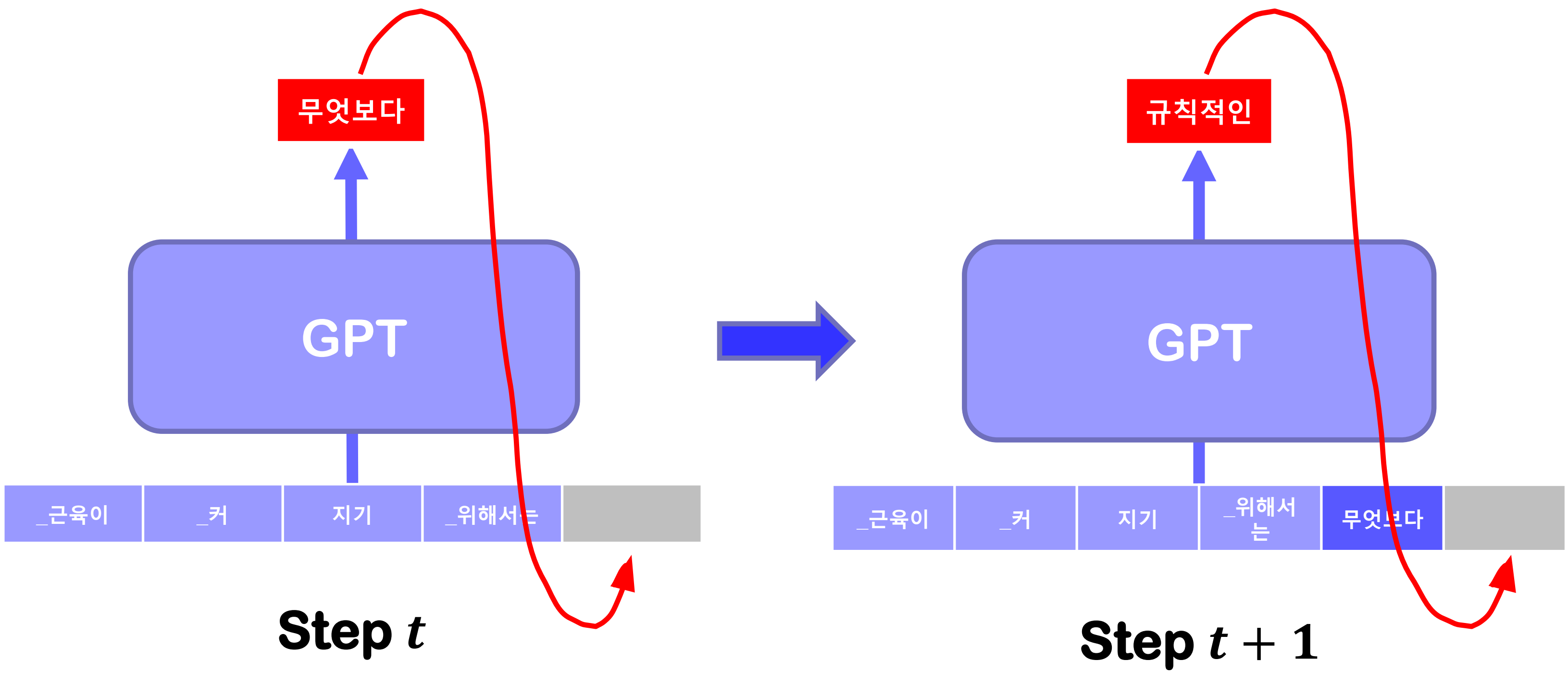
Text Generation (Inference)

- The generation of text is performed by repeatedly predicting the next word
- The predicted word is inserted at the end of the sentence and used as the input of the GPT model



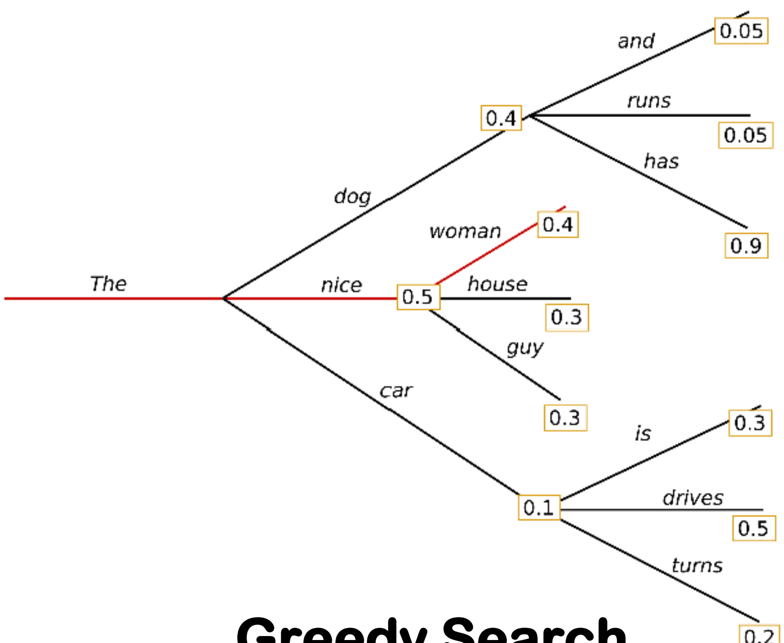
Text Generation (Inference)

- The generation of text is performed by repeatedly predicting the next word
- The predicted word is inserted at the end of the sentence and used as the input of the GPT model

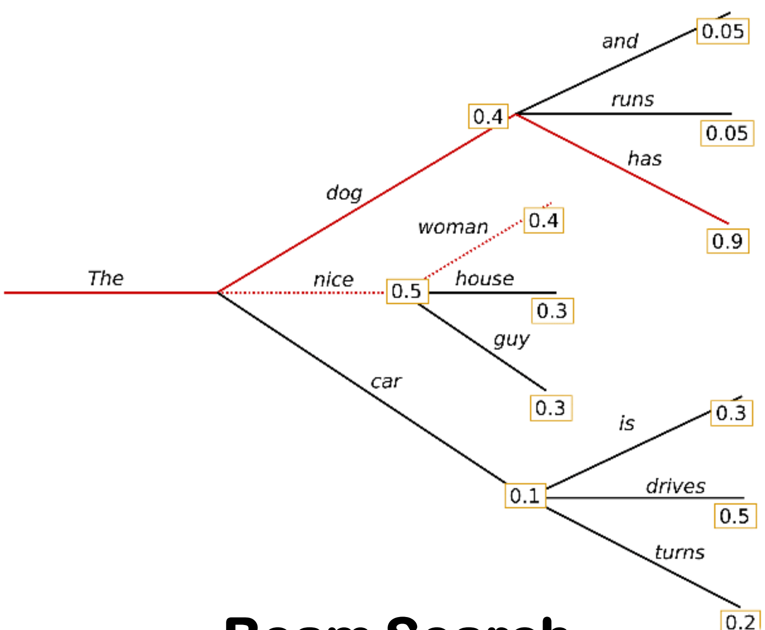


Greedy search vs Beam search

- Greedy search just selects the word with the highest probability
- Beam search keeps the most likely `num_beams` of hypotheses at each time step and eventually choosing the highest probability path



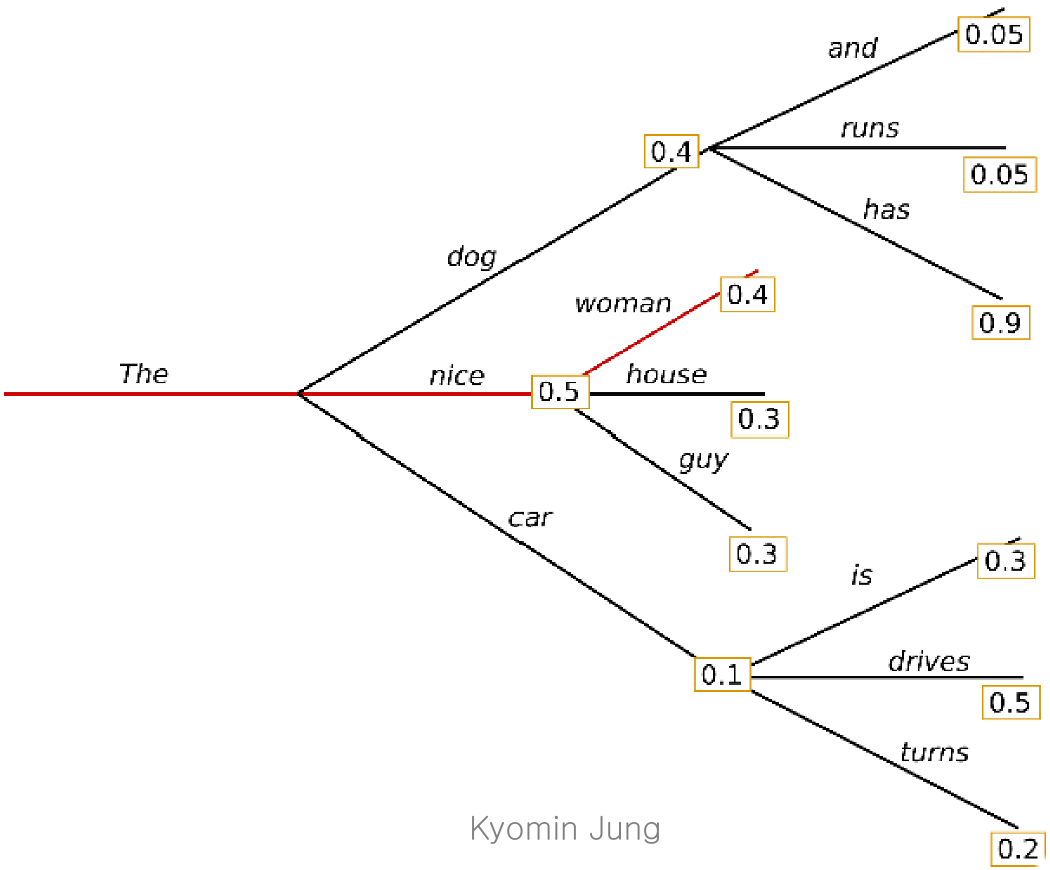
Greedy Search



Beam Search

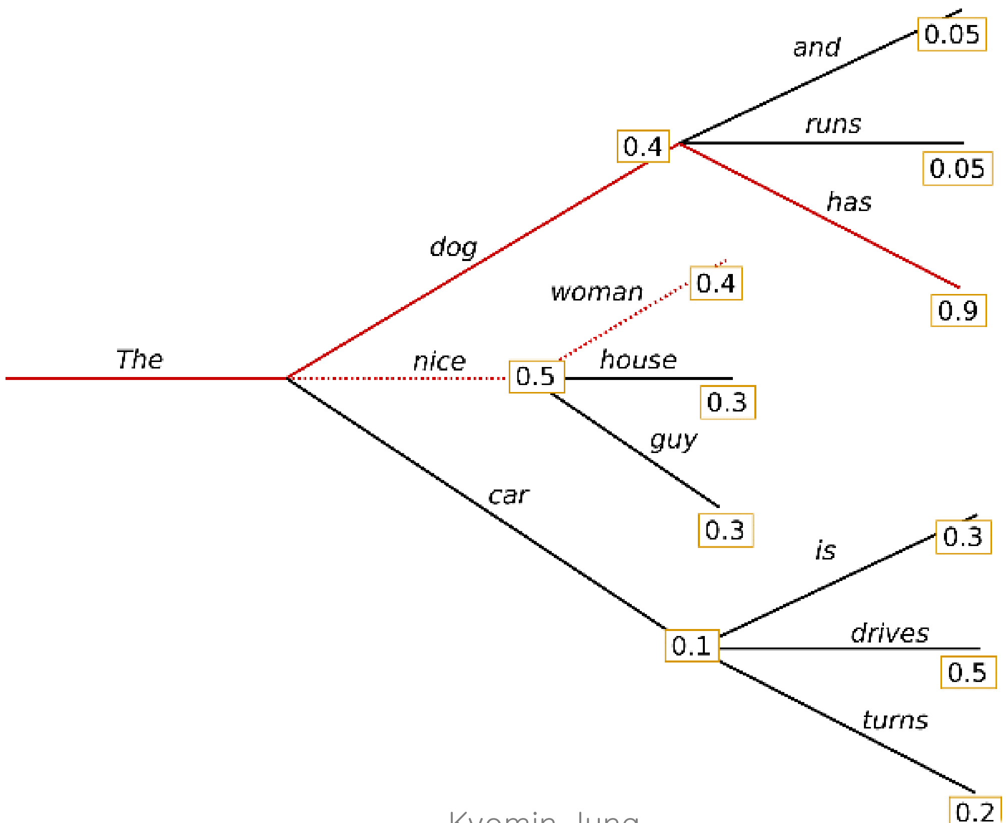
Greedy search

- **The final generated word sequence is "The nice woman" in greedy search.**
- $\mathbf{P(\text{"nice"} \mid \text{"The"})\, P(\text{"woman"} \mid \text{"The"}, \text{"nice"}) = 0.5 \times 0.4 = 0.2}$



Beam search

- **The final generated word sequence is "The dog has" in Beam search.**
- $\mathbf{P(\text{"dog"} \mid \text{"The"})\, P(\text{"has"} \mid \text{"The"}, \text{"dog"}) \;=\; 0.4 \times 0.9 = 0.36}$



Text Generation (Greedy search)

- **GPT2LMHeadModel has** `generate` **method which is used to generate text**
- `generate` **method predicts the next word repeatedly**

```
gen_ids = gpt2lm_model.generate(torch.tensor([token_ids]),
                                max_length=127,
                                repetition_penalty=2.0)

generated = tokenizer.decode(gen_ids[0,:].tolist())
print(generated)
```

Text Generation (Beam search)

- **GPT2LMHeadModel has** `generate` **method which is used to generate text**

- `generate` **method predicts the next word repeatedly**

```
gen_ids = gpt2lm_model.generate(torch.tensor([token_ids]),
                                max_length=127,
                                repetition_penalty=2.0,
                                num_beams=5)

generated = tokenizer.decode(gen_ids[0,:].tolist())
print(generated)
```

Practice: Summing Up

- Dataset loader
- Model Class
- Optimization
- Training (fine-tuning) & Evaluation Loop

Naver Movie Review Dataset

- Large Movie Review dataset
  - https://github.com/e9t/nsmc

- 150,000 movie reviews for training, and 50,000 for testing

- We only use “test” set
- We further split the set as:
  - 80% for training
  - 10% for development
  - 10% for test

Practice: Loading Dataset

- **Each review is stored as a plain text file**
- **Use pandas's DataFrame to store the data**

```
def get_naver_review_examples():
    url = "https://raw.githubusercontent.com/e9t/nsmc/master/ratings_test.txt"
    urllib.request.urlretrieve(url, filename="ratings_test.txt")
    data = pd.read_table('ratings_test.txt')
    return data
```

Practice: Loading Dataset

- **Exploit pytorch's Dataset class**
- `encode_plus` **method to preprocess input for the model**

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

    def __getitem__(self, item):
        text = str(self.texts[item])
        label = self.labels[item]

        encoding = self.tokenizer.encode_plus(
          text,
          add_special_tokens=True,
          max_length=self.max_len,
          return_token_type_ids=False,
          padding='max_length',
          return_attention_mask=True,
          return_tensors='pt',
          truncation=True,
        )

        return {
          'text': text,
          'input_ids': encoding['input_ids'].flatten(),
          'attention_mask': encoding['attention_mask'].flatten(),
          'labels': torch.tensor(label, dtype=torch.long)
        }

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

Get texts and labels

Encode texts with PAD token

Return data

Practice: Loading Dataset

- Data loader for training and testing
- We split the set as: train:valid:test = 8:1:1

```
batch_size = 16

naver_data = get_naver_review_examples()  Load Naver Review Text Data

dataset = NaverReviewDataset(naver_data['document’],
                             naver_data['label'], tokenizer, 100)  Make Dataset

train_set, valid_set, test_set =
torch.utils.data.random_split(dataset, [40000, 5000, 5000])  Split Data

train_dataloader = DataLoader(train_set,
                              batch_size=batch_size,
                              shuffle=True, num_workers=4)

valid_dataloader = DataLoader(valid_set,
                              batch_size=batch_size,  Make DataLoader
                              shuffle=True, num_workers=4)

test_dataloader = DataLoader(test_set,
                             batch_size=batch_size,
                             shuffle=True, num_workers=4)
```

Practice: Model Class & Optimization

- Adam as an optimizer
- Cross entropy loss with true label as objective

```
gpt2_model = GPT2Model.from_pretrained('skt/kogpt2-base-v2')

gpt2lm_model.train()

learning_rate = 1e-5

criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(gpt2lm_model.parameters(),
                             lr=learning_rate)

device = 'cuda'

epochs = 1
count = 0
```

Practice: Fine tuning & Evaluation Loop

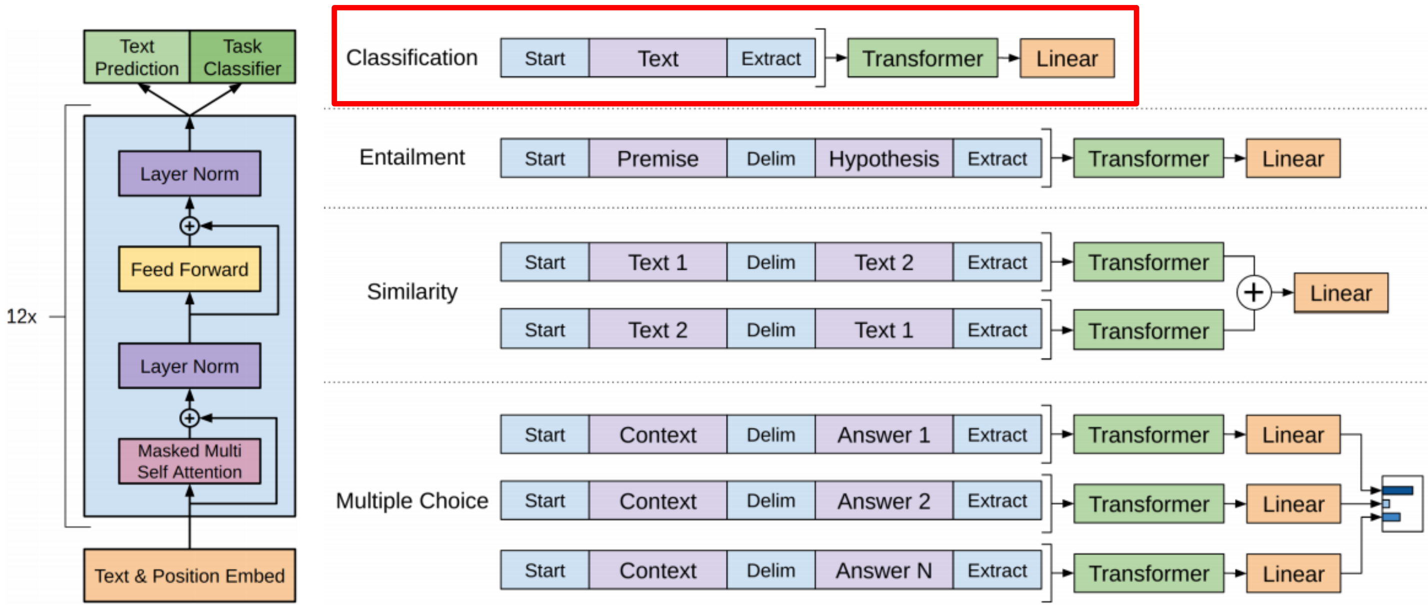
- **Train loop consists following step, as training other deep neural network models**
  - **Prepare input**
  - **Get the output from model**
  - **Calculate loss function**
  - **Backpropagation**
  - **Update parameters**

- **Evaluation is done similarly, except omitting optimization and adding metrics**

```
for epoch in range(epochs):
    for batch, train_data in enumerate(train_dataloader):
        train_inputs = train_data['input_ids'].to(device)

        # To-do : train_outputs and train_loss

        valid_inputs = valid_data['input_ids'].to(device)

        # To-do : valid_outputs and valid_loss

        # To-do : optimization

        tot_train_loss += train_loss.item()
        tot_valid_loss += valid_loss.item()

        …
```

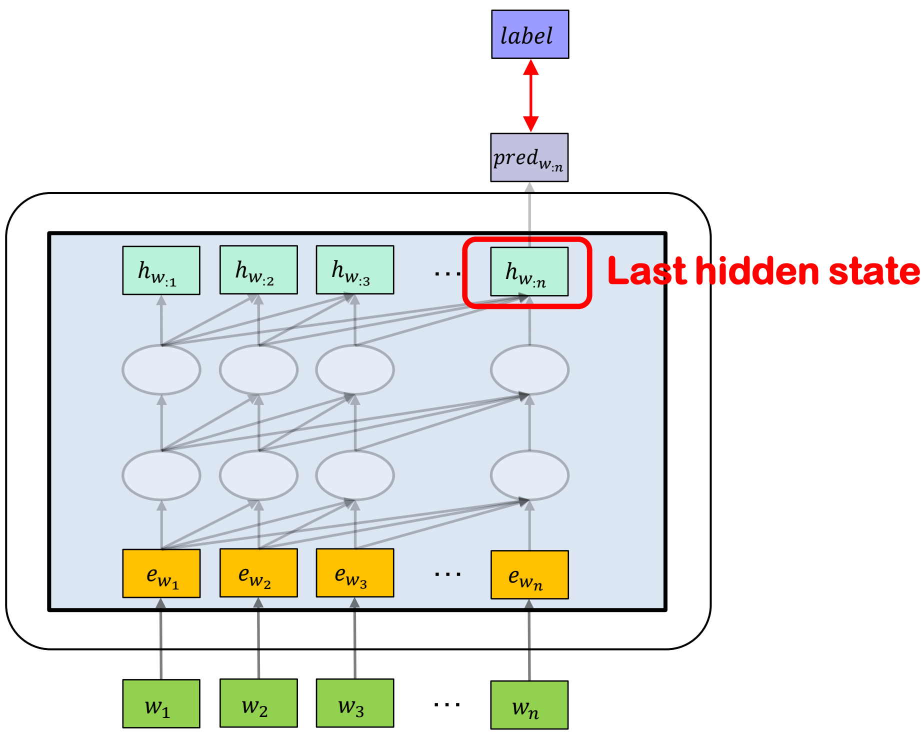
Exercise: Finetuning (other task)

- GPT2 model can be used for other tasks
  - Classification, Entailment, Similarity, …
- Let's implement "Sentiment Classifier" with pretrained KoGPT2
  - You should use additional layer to implement "Sentiment Classifier"



Exercise: Finetuning (other task)

- **Hint 1**
  - You should use last hidden state to classify sentiment
  - The last hidden state has the full information of the input sentence



Exercise: Finetuning (other task)

- Hint 2
  - You should use **forward-style zero padding**
  - If you use backward-style zero padding, you may not take full advantage of the information in the input sentence