[Hands-On] Head-based Text Classification with BERT

Hugman Sangkeun Jung · Follow

12 min read · Apr 14, 2024




> This code is written for educational purposes.

(You can find the Korean version of the post at this link.)

Classification techniques are incredibly diverse. In this post, we'll delve into one such technique within the realm of neural networks, specifically focusing on **transformer-based classification** methods known as **"Head-based" classification.** This article marks the first installment in our Head-based classification tutorial series, starting with *text classification* using BERT. The subsequent post will explore image classification based on ViT.

[Hands-On] Head-based Image Classification with ViT

Explore head-based image classification with Vision Transformers. Learn how ViT applies to image patches for...

medium.com

What is Head-Based Classification?

Head-based classification involves appending a “head” (a layer or set of layers of a neural network) to a pre-trained model to carry out a specific task, such as classifying the topics of texts. In this scenario, the “head” is optimized to output predictions for the classification task, effectively leveraging the rich representations learned by the network from ample data to adeptly address the problem at hand — in this case, classification.

By employing a head-based approach, we can fine-tune the pre-trained model on a relatively small dataset tailored to our specific task, making this method highly versatile and applicable across a wide range of natural language processing (NLP) challenges.

Head-Based Classification with BERT

In the BERT model, the [CLS] token plays a pivotal role in sequence-level tasks such as classification. This section will provide a step-by-step explanation of the interaction between the [CLS] token and the classification head.

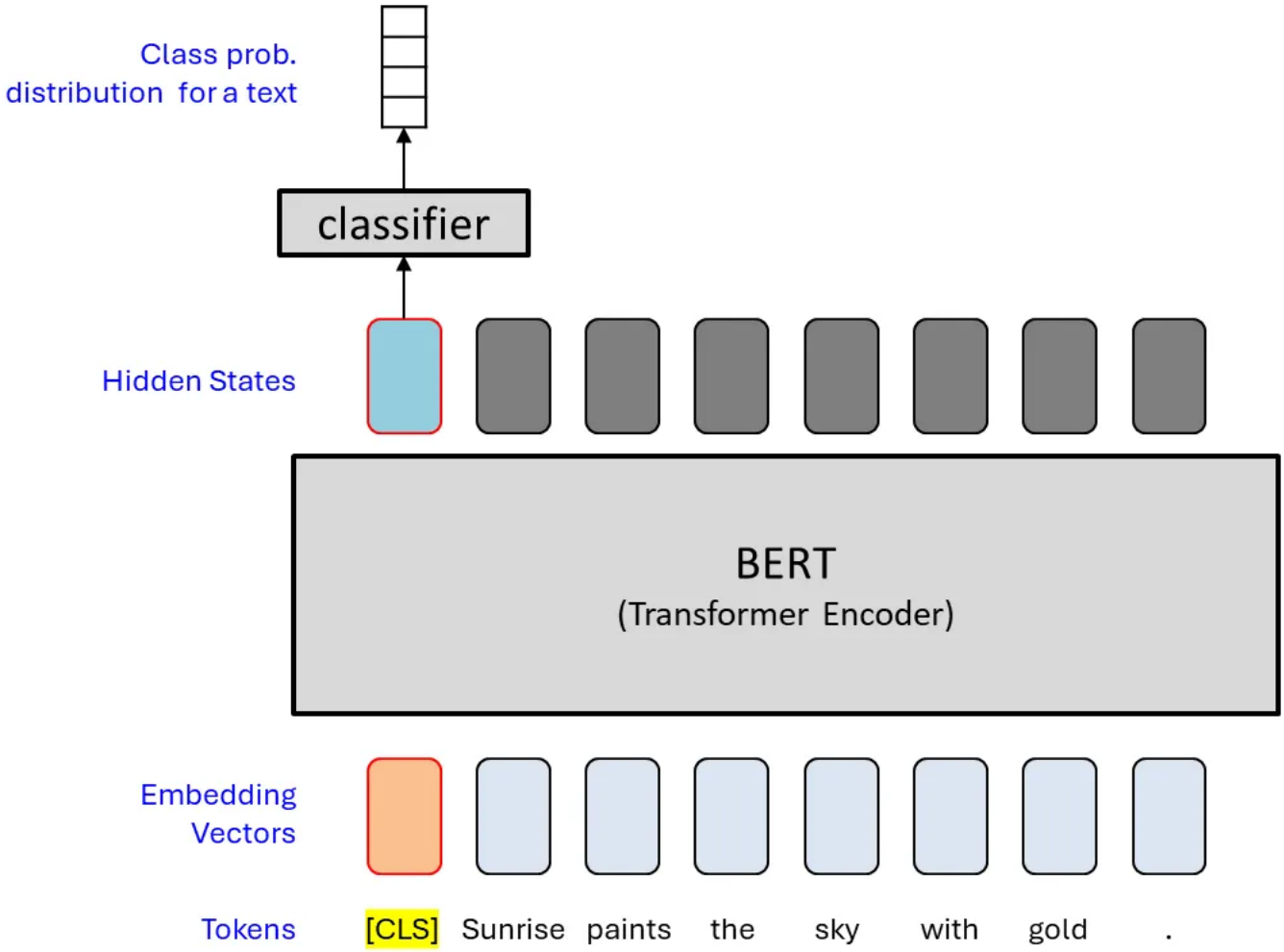
1. **Pre-trained BERT Initialization**
   Begin by preparing a BERT model that has been pre-trained on extensive text corpora, typically downloaded using tools like Hugging Face. This model already possesses the capability to understand

language to a certain extent, including grammar, context, and semantics across a broad domain.

2. **Adding the [CLS] Token**
   For each input sequence, a [CLS] token is added at the beginning. In standard BERT usage, initiating the tokenizer automatically adds a [CLS] dummy token at the forefront of the returned sequence. The hidden states corresponding to this [CLS] token, as the sequence passes through BERT's layers, aggregate contextual information from the input, condensing the entire sequence into a succinct vector representation.

3. **Designing the Classification Head**
   The vector corresponding to the [CLS] token's position from the top layer of BERT is extracted and connected to the classification head. This head is typically a simple neural network culminating in a softmax layer, mapping the rich representation of the [CLS] token to specific class labels.

4. **Fine-Tuning**
   The combined model, consisting of BERT and the classification head, is fine-tuned on a dataset tailored to the specific classification task. This stage optimizes both the parameters of BERT and the classification head to ensure optimal performance for the intended application.



Head-based classification with BERT

Implementation Overview

In this practical exercise, we'll navigate through the following key steps:

1. **Dataset Preparation and Model Initialization:** Load the AG News dataset and prepare for topic classification training with a BERT model as part of the GLUE benchmark. This stage involves setting up the necessary libraries and tools, along with initializing a pre-trained BERT model.
2. **Adding the [CLS] Token and Classification Head:** Append a [CLS] token at the beginning of each input sequence and connect a classification head to the BERT model's output corresponding to the [CLS] token. This classification head is designed to reflect the four topic categories.
3. **Fine-tuning and Evaluation:** Fine-tune the pre-trained BERT model with the added classification head using the AG News dataset. The training process utilizes a custom loop that feeds data to the model in batches, backpropagates errors to update the model's weights, and periodically assesses the model's performance using a validation set. The final model is evaluated using metrics including accuracy, precision, recall, and F1 score.
4. **Results Visualization and Analysis:** Analyze and visualize the performance of the trained model using visual tools like confusion matrices. This helps to understand how well the model classifies each topic and identify any topics where errors occur.

Setting Up the Environment

Begin by importing the necessary libraries for data loading, model building, training, and evaluating. This step is critical as it equips our working environment with the essential tools and libraries needed for the task at hand. It's akin to gathering all your ingredients before commencing to cook.

```
!pip install -qqq seaborn # for evaluation visualization
!pip install -qqq wandb   # for logging
!pip install -qqq datasets # huggingface's lib.
!pip install -qqq transformers==4.39.2
!pip install -qqq accelerate==0.28.0
!pip install -qqq shortuuid
!pip install -U accelerate
!pip install tensorboard
```

```
from transformers import BertTokenizer, BertForSequenceClassification
from transformers import Trainer, TrainingArguments
from transformers import DataCollatorWithPadding
from datasets import load_dataset
import numpy as np
from sklearn.metrics import accuracy_score, precision_recall_fscore_support,
import seaborn as sns
import matplotlib.pyplot as plt
import wandb
import torch
import random
import os

# Function to set the seed for reproducibility
def set_seed(seed_value=42):
    """Set seed for reproducibility."""
    np.random.seed(seed_value)
    torch.manual_seed(seed_value)
    torch.cuda.manual_seed(seed_value)
    torch.cuda.manual_seed_all(seed_value)  # if you are using multi-GPU.
```

```
    random.seed(seed_value)
    os.environ['PYTHONHASHSEED'] = str(seed_value)

    # The below two lines are for deterministic algorithm behavior in CUDA
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

# Set the seed
set_seed()
```

Loading the Dataset

For this project, we will use the AG News dataset, accessible through the Hugging Face's `datasets` library. The loaded dataset comprises training and validation data, with each news article's text and its corresponding topic label. After loading, we'll inspect the dataset's structure and a few samples to ensure it has been correctly loaded. For more detailed information, please refer to the provided link.

```
dataset = load_dataset("ag_news")
print(dataset)
```

```
DatasetDict({
    train: Dataset({
        features: ['text', 'label'],
        num_rows: 120000
    })
    test: Dataset({
        features: ['text', 'label'],
        num_rows: 7600
    })
```

```
})
```

```
from pprint import pprint # Using Python's pprint (pretty-print) library to |

print(type(dataset)) # Data type
print(dataset) # Data structure and count
print("\n"*2+ "Train dataset:")
pprint(dataset["train"][1000]) # Print to check the content of train data

# Explanation of labels - 4 classes # https://huggingface.co/datasets/ag_news
# 1 class: World news
# 2 class: Sports news
# 3 class: Business news
# 4 class: Science/Technology news
```

```
class 'datasets.dataset_dict.DatasetDict'>
DatasetDict({
    train: Dataset({
        features: ['text', 'label'],
        num_rows: 120000
    })
    test: Dataset({
        features: ['text', 'label'],
        num_rows: 7600
    })
})


Train dataset:
{'label': 3,
 'text': 'European Union Extends Microsoft-Time Warner Review BRUSSELS, '
         'Belgium (AP) -- European antitrust regulators said Monday they have
         'extended their review of a deal between Microsoft Corp. (MSFT) and
         'Time Warner Inc...'}
```

Initializing the Tokenizer and Preprocessing Data

The tokenizer converts text into tokens that can be fed into the BERT

model, preparing our data in a format that the model can interpret.

```
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

def tokenize_function(examples):
    return tokenizer(examples['text'], padding=True, truncation=True)

tokenized_datasets = dataset.map(tokenize_function, batched=True)
```

Data Collator

The data collator automatically preprocesses data to fit a specific model. A fundamental function includes *padding* the sentence lengths of each sample in a batch to match the input length through padding.

```
data_collator = DataCollatorWithPadding(tokenizer=tokenizer)
```

In fact, the data collator is one of the data processing patterns commonly used in Hugging Face. As this will be frequently used later, let's take a look at the flow of how data is processed within Hugging Face.

Data Handling Flow in Hugging Face

In most cases, the flow in Hugging Face follows `dataset` → `mapping function` → `data collator` → `data loader`.

1. `dataset`: Manages data objects like the AG News dataset.

2. `mapping function` : Implements functions to transform raw text into a format understandable by the model through tokenization and preprocessing, or other preprocessing functions. Usually, developers customize this according to their data.
3. `data collator` : Essential for handling variable-length sequences, dynamically padding batches to ensure uniformity and facilitate smooth model input. Typically, Hugging Face provides data processing functionalities corresponding to the model type (e.g., Seq2Seq model, TokenClassification, Causual_LM, etc.), which are already implemented and distributed. Developers can also write their own if necessary.
4. `data loader` : The final step that prepares data for the model by batching, shuffling, and creating an optimized iterable for training or evaluation.

Following this flow for coding, developers can expect high code readability and reusability when utilizing different datasets or models, becoming familiar with this pattern.

Training the Model

Now that our data is ready, let's proceed with training. The process will follow these steps:

1. **Define the Model**: Load BERT and add a classification layer suited to our topics.
2. **Training Arguments**: Set parameters like epochs, batch size, and

learning rate.

3. **Training Loop:** Iterate through the data, adjusting parameters to enhance classification accuracy.
4. **Evaluation:** Test the model's performance on separate data to assess its generalization ability.

Defining the Model

We will initialize the BERT model for sequence classification here. The `num_labels=4` indicates we're dealing with *four* possible outcomes, matching the number of topics in the AG News dataset. Thus, the loaded BERT model automatically constructs a structure with a head capable of outputting four classification labels.

```
model = BertForSequenceClassification\
        .from_pretrained('bert-base-uncased', num_labels=4)
```

Defining Training Arguments

Training arguments define various parameters for the training process, such as the number of epochs, batch size, learning rate adjustments, and where to store logs and outputs.

```
training_args = TrainingArguments(
    output_dir='./results-bert-topic-cls',
    num_train_epochs=3,
    per_device_train_batch_size=32,
    per_device_eval_batch_size=8,
    warmup_steps=500,
```

```
    weight_decay=0.01,
    logging_dir='./logs',

    evaluation_strategy='epoch',  # Evaluate at the end of each epoch
    logging_steps=10,
    ## ----
    report_to="tensorboard",
)
```

Defining Evaluation Functions

Before defining the training loop, we usually add a feature to automatically evaluate during training by defining a metric function.

```
def compute_metrics(p):
    predictions, labels = p
    predictions = np.argmax(predictions, axis=1)
    precision, recall, f1, _ = precision_recall_fscore_support(labels, predi
    acc = accuracy_score(labels, predictions)
    return {'accuracy': acc, 'f1': f1, 'precision': precision, 'recall': rec
```

Since we're dealing with a classification problem, we'll utilize **Precision, Recall, F1 Score**, and **Accuracy**, commonly used in classification tasks. We won't delve into each metric's meaning here; refer to this link for a deeper understanding.

Training Loop

```
# Select the first N samples from the tokenized training dataset
subset_train_dataset = tokenized_datasets['train'].select(range(6000)) # 1/2

trainer = Trainer(
```

```
    model=model,
    args=training_args,
    train_dataset=subset_train_dataset,
    eval_dataset=tokenized_datasets['test'],
    data_collator=data_collator,
    compute_metrics=compute_metrics,
)
```

In traditional training processes, a for-loop is used to perform forward and backward passes repeatedly. While this approach is fundamental in PyTorch, using Hugging Face's `Trainer` class can simplify this process. `Trainer` proovides a high-level interface for training, evaluation, and prediction, making complex training loops straightforward. Users can adjust training settings through `TrainingArguments` and use Trainer's methods to train and evaluate the model. This tool is very useful when managing the training process efficiently.

With everything set up, we execute the following command to start training:

```
trainer.train()
```

[564/564 21:47, Epoch 3/3]

Epoch	Training Loss	Validation Loss	Accuracy	F1	Precision	Recall
1	0.289600	0.302556	0.898684	0.898765	0.899624	0.898684
2	0.217800	0.296935	0.903947	0.903986	0.906630	0.903947
3	0.140100	0.346284	0.904605	0.904311	0.909090	0.904605

```
TrainOutput(global_step=564, training_loss=0.23599527131581138, metrics={'train_runtime': 1308.7238,
'train_samples_per_second': 13.754, 'train_steps_per_second': 0.431, 'total_flos': 3163210797786624.0,
'train_loss': 0.23599527131581138, 'epoch': 3.0})
```

For a quick learning demonstration, we've conducted training for only up to 3 epochs. Even without fine-tuning hyperparameters, we can observe that the training achieved approximately 90% accuracy.

Saving the Model

```
# Specify the directory where you want to save your model
output_dir = './bert-topic-cls'

# Save the model
model.save_pretrained(output_dir)
# Save the tokenizer
tokenizer.save_pretrained(output_dir)
```

We'll save the recently trained model and tokenizer. By saving them, we can easily use them later in a pipeline, which I'll explain further in the later.

Evaluating the Model

Now, let's evaluate the model we've trained.

```
# Evaluate the model
results = trainer.evaluate()
```

When `trainer.evaluate()` is executed, it returns the evaluation results as 'results', which detail the performance of the model based on the evaluation metrics specified.

```
print( results )
```

```
{'eval_loss': 0.3104916214942932,
 'eval_accuracy': 0.9064473684210527,
 'eval_f1': 0.9062005426924208,
 'eval_precision': 0.9085149708994765,
 'eval_recall': 0.9064473684210527,
 'eval_runtime': 244.8868,
 'eval_samples_per_second': 31.035,
 'eval_steps_per_second': 3.879,
 'epoch': 3.0}
```

Confusion Matrix

Another way to closely examine classification performance is through the **Confusion Matrix**. The Confusion Matrix is a highly useful tool for evaluating the performance of models in classification problems. It visually represents the accuracy of predictions by basing on the actual and the model-predicted classes. The matrix is structured with actual classes as rows and predicted classes as columns, where the value in each cell represents the number of samples belonging to the corresponding

class combination.

The principal diagonal of the Confusion Matrix (diagonal from the top left to the bottom right) represents the cases where the model has made correct predictions, i.e., True Positives (TP) and True Negatives (TN). In contrast, the off-diagonal elements represent errors: False Positives (FP) indicate cases where the model incorrectly predicted a negative class as positive, and False Negatives (FN) indicate cases where the model incorrectly predicted a positive class as negative.

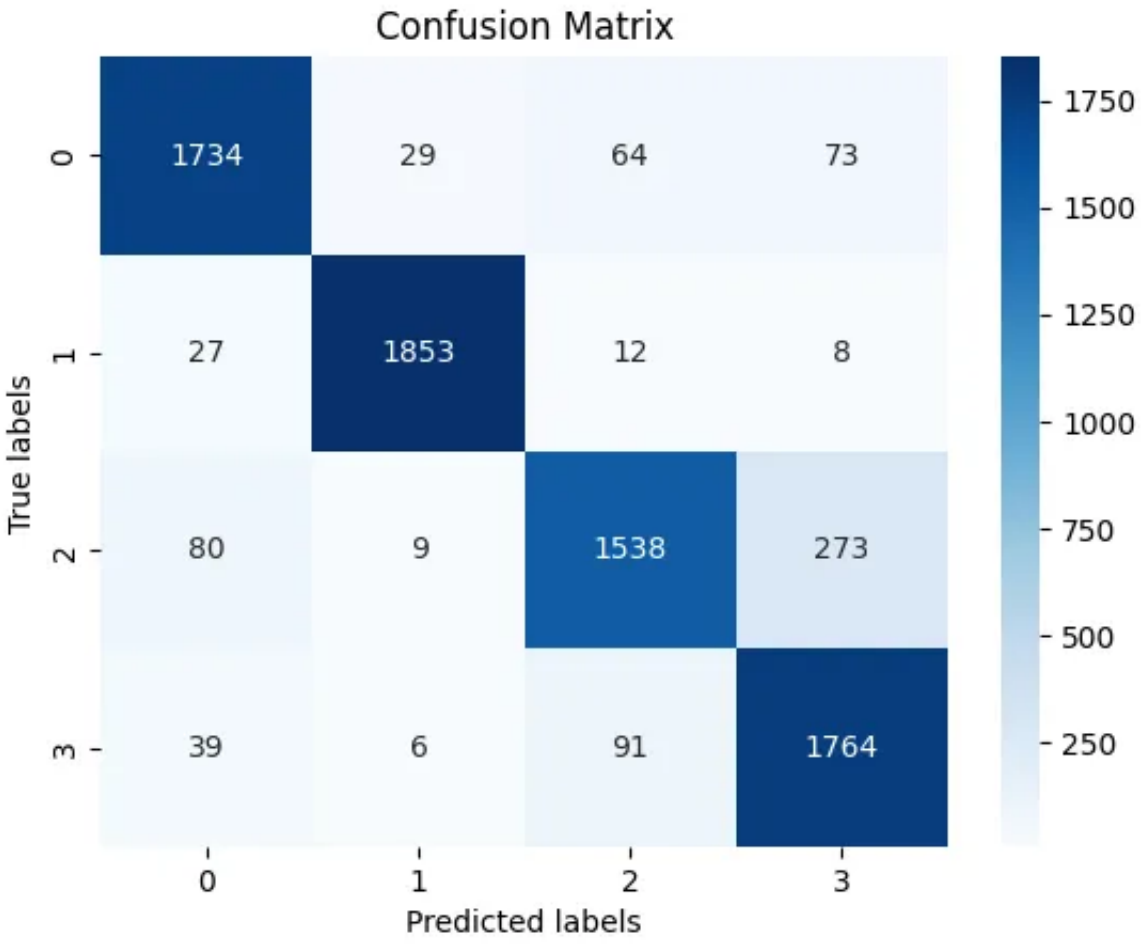
Through the Confusion Matrix, not only the accuracy but also other important performance metrics such as Precision, Recall, and F1 score can be calculated. These metrics offer a more detailed evaluation of the model's performance and are particularly helpful in understanding the performance of models on imbalanced datasets.

Let's draw a confusion matrix using our results and the correct answer data.

```
# Predictions to get the confusion matrix
predictions = trainer.predict(tokenized_datasets['test'])
preds = np.argmax(predictions.predictions, axis=-1)
```

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix

label_map = {
```

```
    'LABEL_0': 'World',
    'LABEL_1': 'Sports',
    'LABEL_2': 'Business',
    'LABEL_3': 'Sci/Tech'
}

cm = confusion_matrix(predictions.label_ids, preds)

# label_map to labels
labels = [label_map[f'LABEL_{i}'] for i in range(len(label_map))]

# Confusion Matrix
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=labels, ytick
plt.xlabel('Predicted labels')
plt.ylabel('True labels')
plt.title('Confusion Matrix with Label Names')
plt.show()
```



Confusion Matrix

A good Confusion Matrix should have most of the correct predictions concentrated along the *diagonal*. This indicates that the model is accurately distinguishing between different classes and minimizing errors. The higher the values on the diagonal and the lower the values off the diagonal, the better the model's performance is considered.

For instance, in the illustration above, although the classification is generally well-performed, it can be observed that sentences from the 'World' topic are being confused with the 'Sci/Tech' topic. If this model were to be refined, more attention would need to be paid to this area.

Prediction

Instead of using the entire dataset, individual 'sentences' can also be fed into the model for prediction, similar to the data preparation process. This involves first *tokenizing* the sentence using a tokenizer. If necessary, after performing a certain level of preprocessing, the sentence is fed forward through the model to obtain the output of the classification layer. This output is then analyzed to derive the final class label.

```
# Example sentence
sentence = "The stock market is reaching new heights."

# Tokenize the sentence
inputs = tokenizer(sentence, padding=True, truncation=True, max_length=512,
```

```
import torch
# Move inputs to the same device as the model
inputs = {k: v.to(model.device) for k, v in inputs.items()}

# Make prediction
model.eval()  # Set the model to evaluation mode
with torch.no_grad():
    outputs = model(**inputs)
    predictions = outputs.logits.argmax(-1).item()  # Get the predicted clas

# Map the prediction index to the class name (if you have a label map)
simple_label_map = {0: "World", 1: "Sports", 2: "Business", 3: "Sci/Tech"}
predicted_label = simple_label_map[predictions]

print(f"Sentence: '{sentence}'")
print(f"Predicted Label: '{predicted_label}'")
```

```
Sentence: 'The stock market is reaching new heights.'
Predicted Label: 'Business'
```

Predicting with Hugging Face Pipeline

Although writing prediction code from scratch is possible, the prediction process is generally similar and can be patterned to a certain extent. Hugging Face simplifies this with what's called a pipeline. Executing the code below will process it directly.

```
from transformers import pipeline

# Specify the path to your fine-tuned model or use a pre-trained model from
model_path = './bert-topic-cls'  # Change this to your model's path or a Hug

# Load the pipeline for text classification
classifier = pipeline("text-classification", model=model_path, tokenizer=mod
```

We have prepared a classifier by loading our trained model and tokenizer. Now, this classifier can easily perform classification on multiple sentences.

```
# Example sentences
sentences = [
    "The stock market is reaching new heights.",
    "The new sports car has been unveiled at the auto show.",
    "The tech company announced its latest gadget yesterday."
]

# Make predictions
predictions = classifier(sentences)

# Print the predictions using the label map
for sentence, prediction in zip(sentences, predictions):
    # Map the predicted label to the actual class name
    class_name = label_map[prediction['label']]
    print(f"Sentence: '{sentence}'")
    print(f"Predicted Label: '{class_name}' with score {prediction['score']:
```

```
Sentence: 'The stock market is reaching new heights.'
Predicted Label: 'Business' with score 0.9829

Sentence: 'The new sports car has been unveiled at the auto show.'
Predicted Label: 'World' with score 0.6967

Sentence: 'The tech company announced its latest gadget yesterday.'
Predicted Label: 'Sci/Tech' with score 0.9329
```

Conclusion

Through this post, we've explored **head-based classification** techniques and applied them to a specific problem in the field of natural language processing: topic classification. In particular, we learned how to utilize the vector corresponding to the [CLS] token in the BERT model to attach an additional classification layer. Using this technique, we trained and evaluated a model on the AG News dataset and visualized the results to analyze the model's performance.

This practice code can be downloaded or executed directly in Colab.

Google Colaboratory

[Hands-On] Head-based Text Classification with BERT

Hugman Sangkeun Jung

Move on to the next article!

[Hands-On] Head-based Image Classification with ViT

Explore head-based image classification with Vision Transformers. Learn how ViT applies to image patches for...

medium.com

Bert Text Classification Sequence Classifier Head Based Classification Classification Head

Written by Hugman Sangkeun Jung

157 Followers · 2 Following

Hugman Sangkeun Jung is a professor at Chungnam National University, with expertise in AI, machine learning, NLP, and medical decision support.

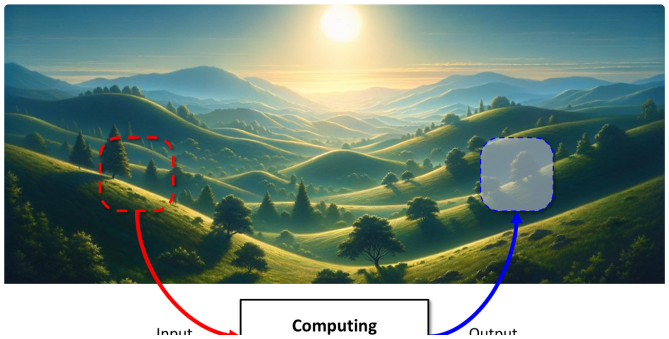
Follow

No responses yet

What are your thoughts?

Respond

More from Hugman Sangkeun Jung

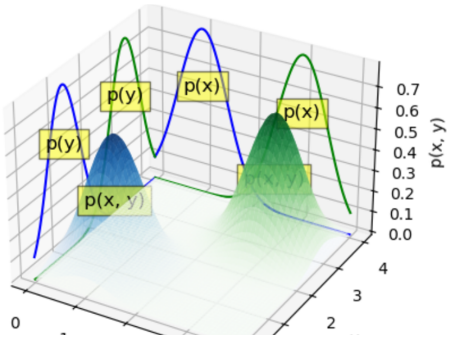


Hugman Sangkeun Jung

Transformer의 큰 그림 이해: 기술적 복잡함 없이 핵심 아이디어 파악하기

트랜스포머의 핵심 아이디어를 이해하세요. 코딩 및 복잡한 이론 없이, 트랜스포머의 핵심만 이해...

Apr 5 61

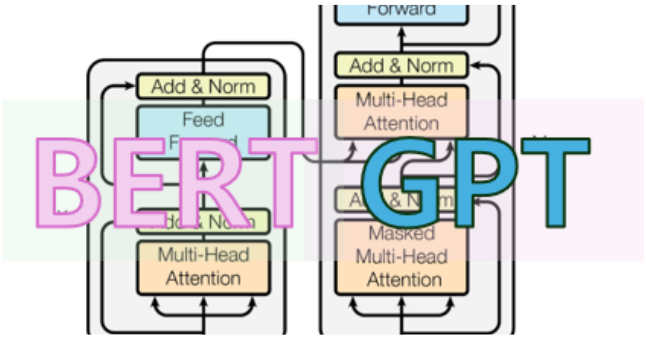


Hugman Sangkeun Jung

Autoencoder 와 Variational Autoencoder의 직관적인 이해

오토인코더와 변분 오토인코더의 구조, 잠재 공간, 생성 학습에서의 활용 방법을 알아보세요. 실습 ...

May 17 60

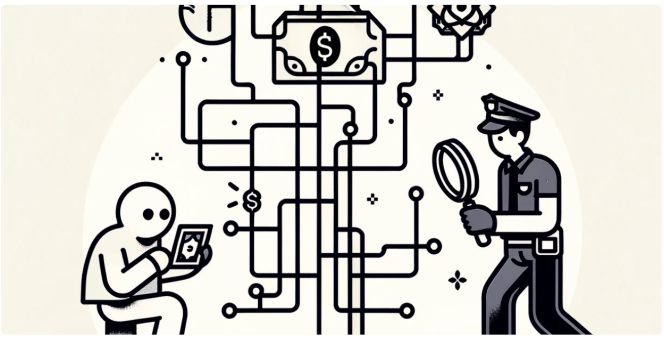


Hugman Sangkeun Jung

가장 성공적인 트랜스포머의 변형: BERT와 GPT 소개

자기지도 학습과 독특한 구조를 활용한 Transformer 의 변형 구조인 BERT와 GPT에 ...

Apr 23 6



Hugman Sangkeun Jung

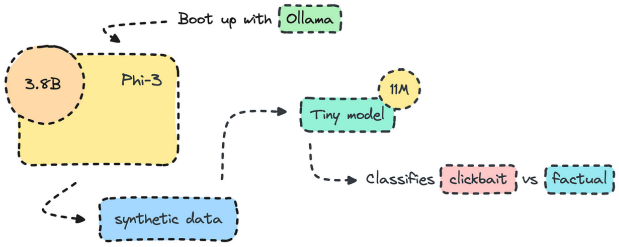
GAN에 대한 이해

생성적 적대 신경망(GAN)의 기본 개념, 훈련 방법, 응용 사례를 실용적인 예제와 구현 팁과 함께 살...

Jun 3 5

See all from Hugman Sangkeun Jung

Recommended from Medium

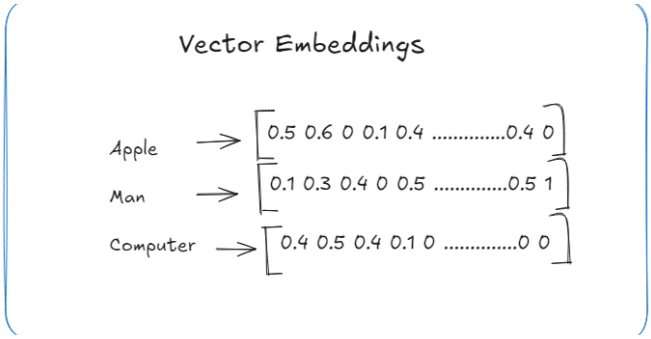


In Towards Data Science by Ida Silfverskiöld

Fine-Tune Smaller Transformer Models: Text Classification

Using Microsoft's Phi-3 to generate synthetic data

May 28 814 7



In Towards AI by Mdabdullahalhasib

A Complete Guide to Embedding For NLP & Generative AI/LLM

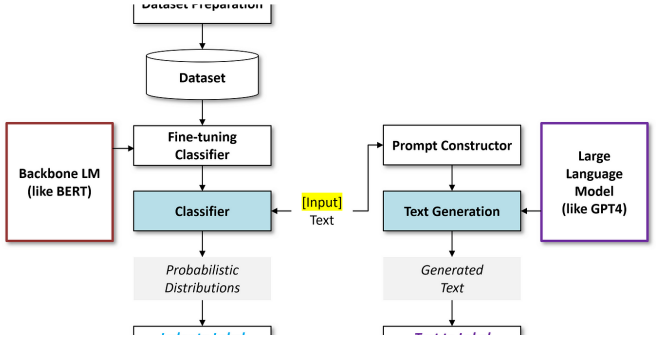
Understand the concept of vector embedding, why it is needed, and...

Oct 18 146

Lists

Natural Language Processing

1851 stories · 1476 saves



Hugman Sangkeun Jung

[Hands-On] Prompt-based Text Classification Using Large...

Learn how to implement prompt-based text classification using large language model...

Jun 30

In AI-Enthusiast by Deepankar Singh

Building a Multi-Class Text Classifier with BERT: A Step-by-...

Unlock the power of BERT for multi-class text classification! Dive into its architectur...

Sep 7 92 2

In CodeX by Rachit

Fine tune BERT for text classification

Jun 20 2

Rahul Kumar

NLP Hands-On with Causal Language Modelling

This post is a part of the NLP Hands-on series and consists of the following tasks: 1..

Oct 27

See more recommendations