15. 5. Natural Language Inference

Decomposable attention model (Parikh et al., 2016)

- Without recurrent or convolutional layers
- But, achieving the best result at the time on the SNLI dataset with much fewer parameters

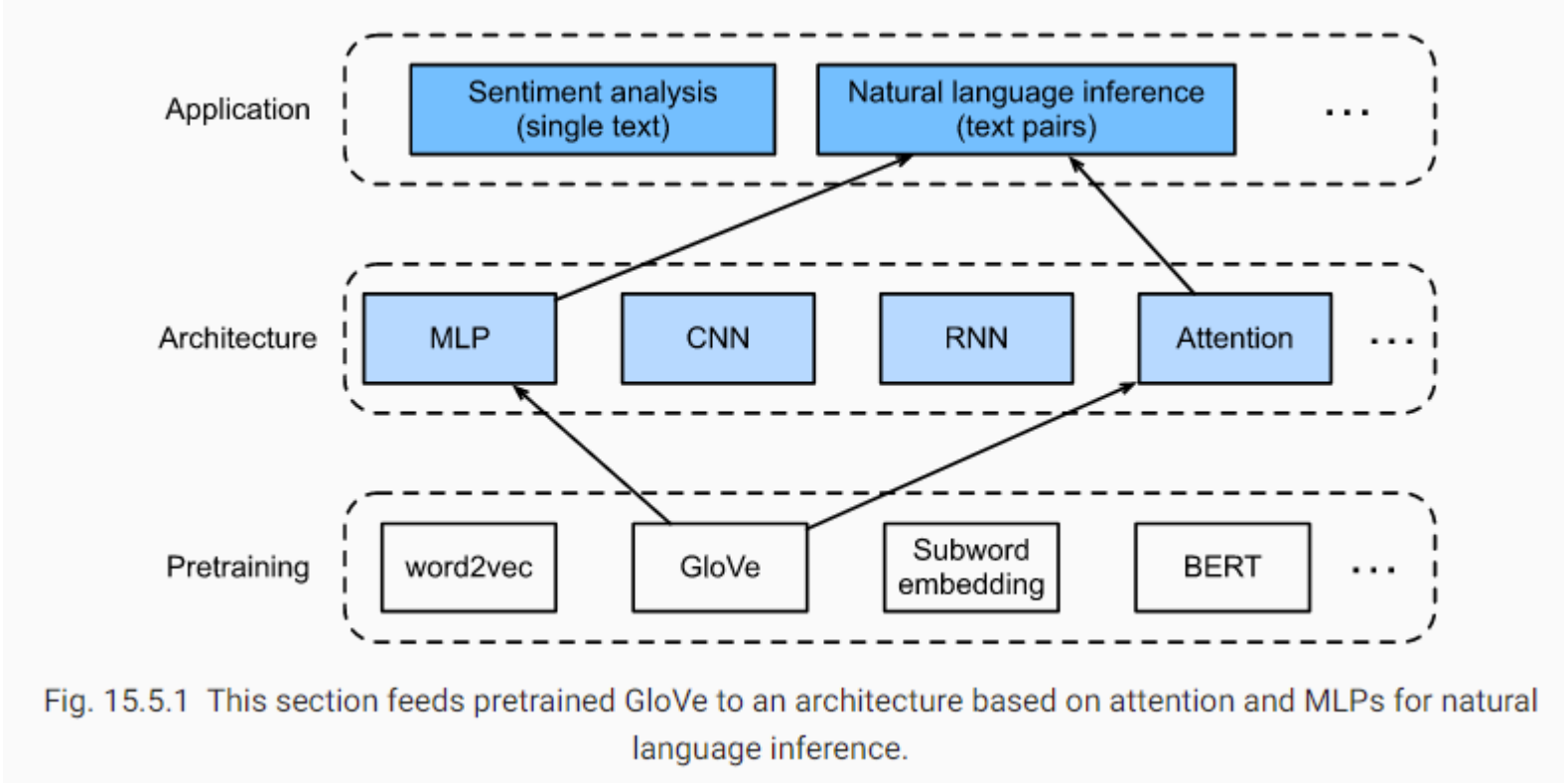


Fig. 15.5.1 This section feeds pretrained GloVe to an architecture based on attention and MLPs for natural language inference.

15. 5. Natural Language Inference

15. 5. 1. The Model

- Align words in one text sequence to every word in the other, and vice versa
- Three jointly trained steps -> attending, comparing, and aggregating

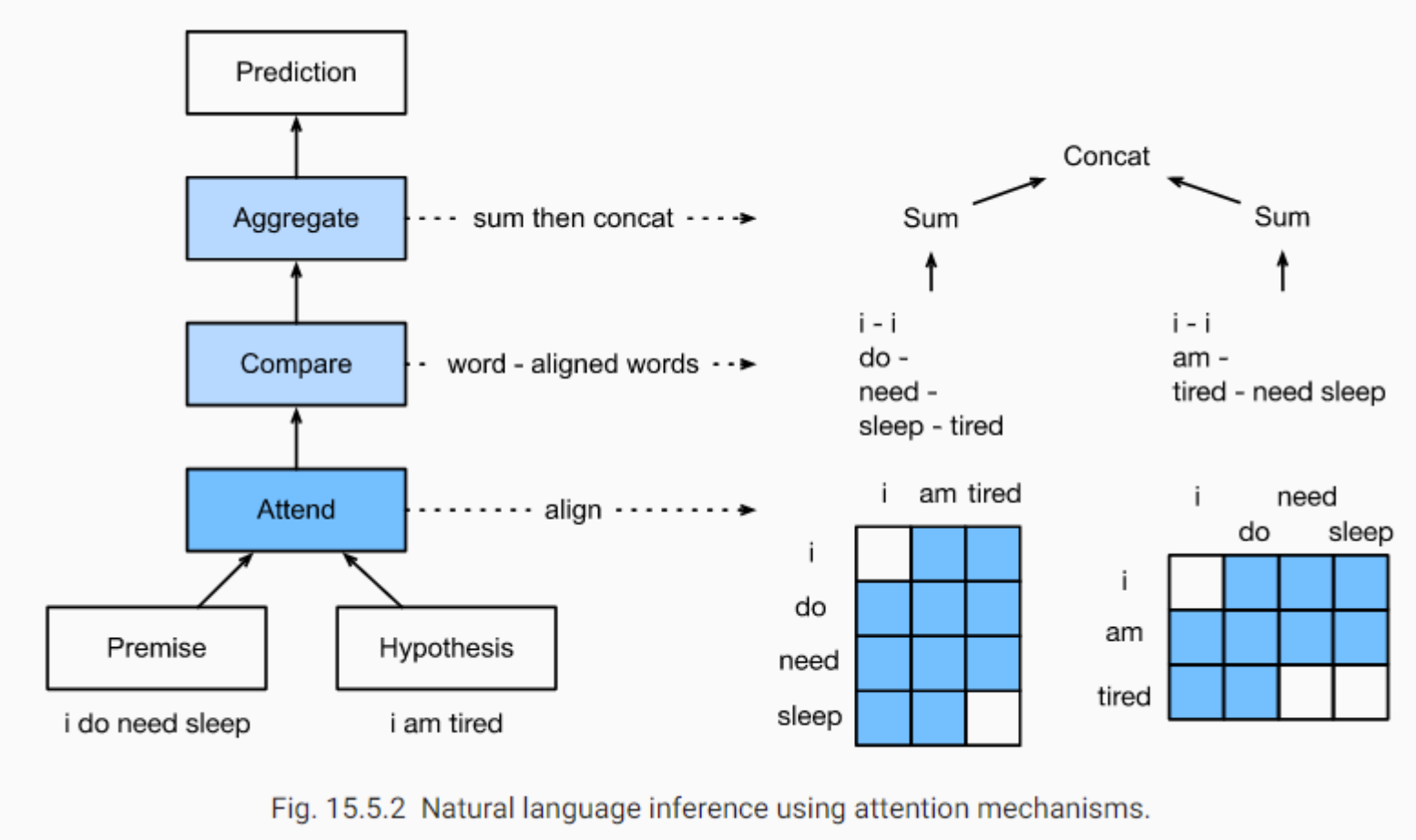


Fig. 15.5.2 Natural language inference using attention mechanisms.

15. 5. Natural Language Inference

15. 5. 1. Attending

- Premise "I do need sleep"
- Hypothesis "I am tired"
- We may wish to align "I" in the hypothesis with "I" in premise, "tired" with "sleep"
- Likewise, may wish to align "I" in the premise with "I" in the hypothesis, "need sleep" with "tired"

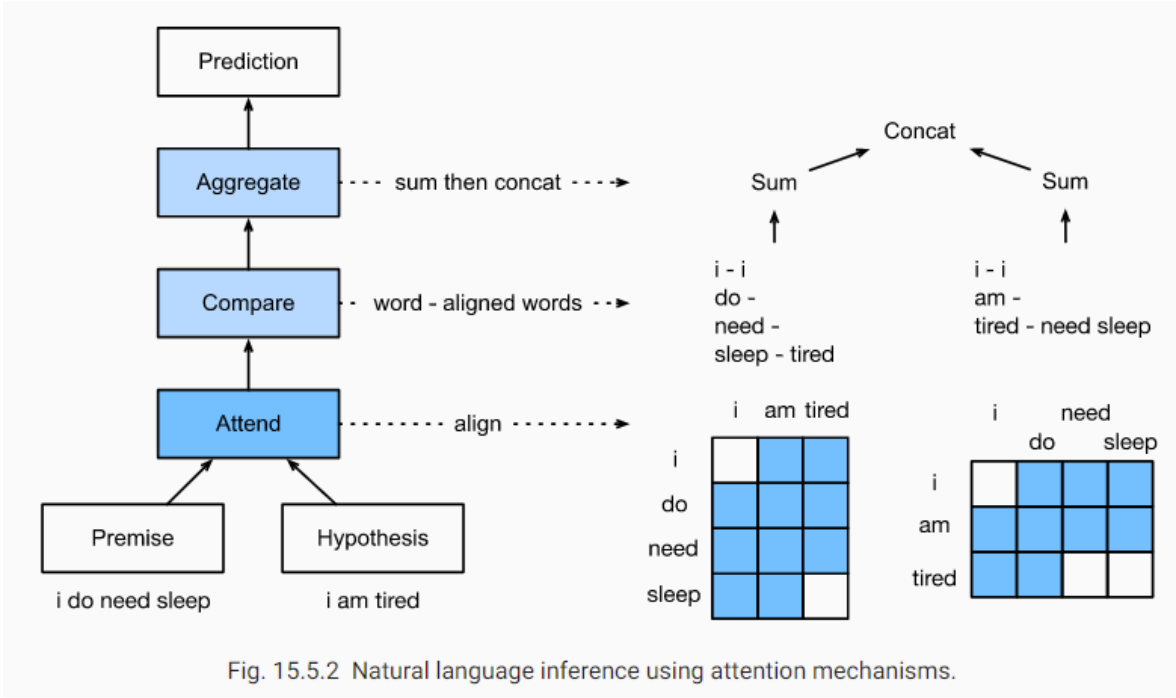


Fig. 15.5.2 Natural language inference using attention mechanisms.

$$\mathbf{a}_i, \mathbf{b}_j \in \mathbb{R}^d \ (i = 1, \ldots, m, j = 1, \ldots, n)$$

$$e_{ij} = f(\mathbf{a}_i)^\top f(\mathbf{b}_j),$$

Weighted average of all the word embeddings

$$\boldsymbol{\beta}_i = \sum_{j=1}^{n} \frac{\exp(e_{ij})}{\sum_{k=1}^{n} \exp(e_{ik})} \mathbf{b}_j.$$

$$\boldsymbol{\alpha}_j = \sum_{i=1}^{m} \frac{\exp(e_{ij})}{\sum_{k=1}^{m} \exp(e_{kj})} \mathbf{a}_i.$$

Beta (softly aligned i in the hypothesis)

Alpha (softly aligned j in the premise)

15. 5. Natural Language Inference

15. 5. 2. Comparing

- Compare a word in one sequence with the other sequence that is **softly aligned**
- For example, "need sleep" in the premise are both aligned with "tired" in the hypothesis, the pair "tired – need sleep" will be compared

$$\mathbf{v}_{A,i} = g([\mathbf{a}_i, \boldsymbol{\beta}_i]), i = 1, \ldots, m$$
$$\mathbf{v}_{B,j} = g([\mathbf{b}_j, \boldsymbol{\alpha}_j]), j = 1, \ldots, n.$$

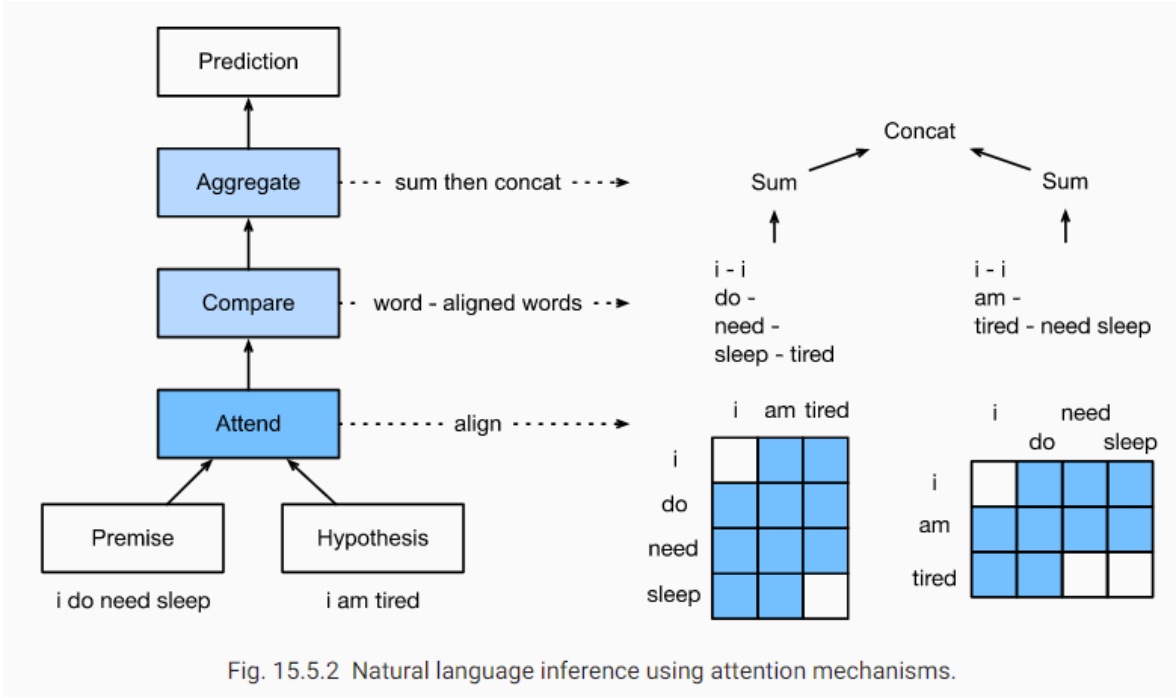


Fig. 15.5.2 Natural language inference using attention mechanisms.

- V_{A, i} is the comparison between word i in the premise and all the hypothesis words softly aligned with word i
- V_{B, j} is the comparison between word j in the hypothesis and all the premise words softly aligned with word j

15. 5. Natural Language Inference

15. 5. 2. Aggregating

- Aggregate such information to infer logical relationship
- Begin by summing up both sets

$$\mathbf{v}_{A,i}\ (i = 1, \ldots, m) \quad \mathbf{v}_{B,j}\ (j = 1, \ldots, n)$$

$$\mathbf{v}_A = \sum_{i=1}^{m} \mathbf{v}_{A,i}, \quad \mathbf{v}_B = \sum_{j=1}^{n} \mathbf{v}_{B,j}.$$

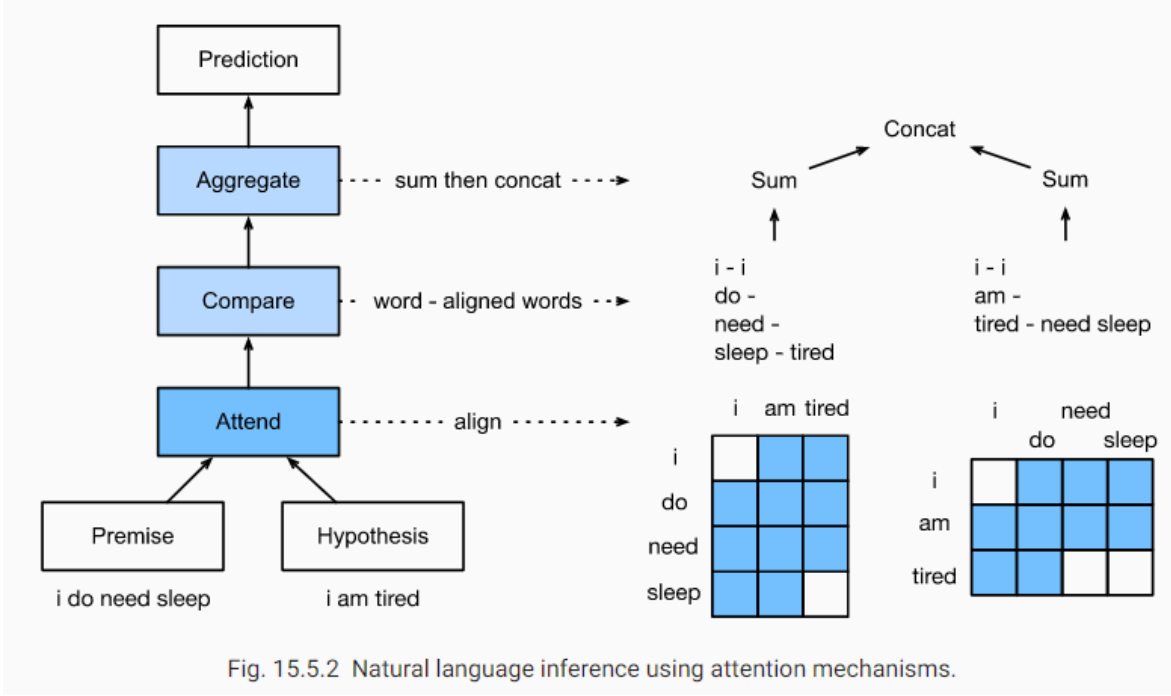


Fig. 15.5.2 Natural language inference using attention mechanisms.

- Concat both summarization results into function MLP to obtain classification

$$\hat{\mathbf{y}} = h([\mathbf{v}_A, \mathbf{v}_B]).$$

15. 6. Fine-Tuning BERT

For Sequence-Level and Token-Level Applications

- Crafting a specific model for every NLP task is practically infeasible (based on RNNs, CNNs, attention…)
- BERT improved the state of the art on various NLP tasks
  - Come with 110 million and 340 million parameters
- We may consider fine-tuning BERT for downstream NLP applications!

- Generalize a subset of NLP apps as **sequence-level** and **token-level**
- How to transform the BERT representation of the text input to the output label in
  - Single text classification
  - Text pair classification
  - Regression
- Token level -> text tagging, question answering

15. 6. Fine-Tuning BERT

15. 6. 1. Single Text Classification

- <cls> is used for sequence classification
- <sep> marks the end of single text or separates a pair of text
- <cls> encodes the information of the entire input text sequence
- Will be fed into MLP generating output

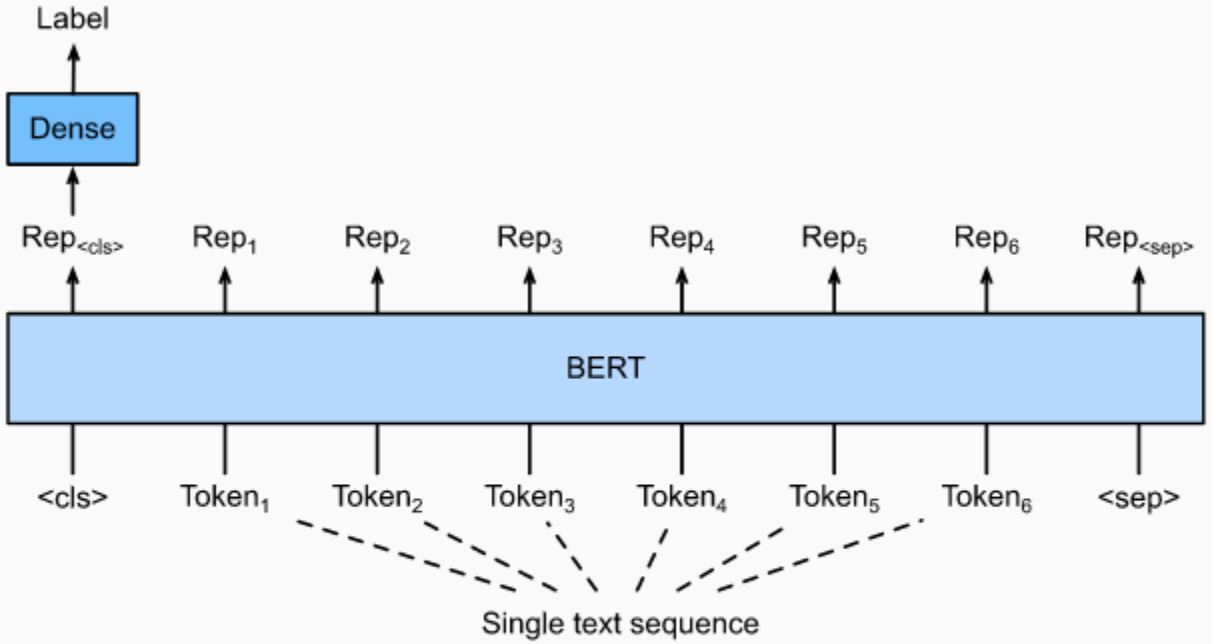


Fig. 15.6.1 Fine-tuning BERT for single text classification applications, such as sentiment analysis and testing linguistic acceptability. Suppose that the input single text has six tokens.

15. 6. Fine-Tuning BERT

15. 6. 2. Text Pair Classification or Regression

- Taking a pair of text as the input but outputting a continuous value (such as semantic textual similarity)
- Datasets are consisting of (sentence 1, sentence 2, similarity score)
- From 0 (no meaning overlap) to 5 (meaning equivalence)

- A Plain is taking off <-> An air plane is taking off => 5
- A woman is eating something <-> A woman is eating meat => 3
- A woman is dancing <-> A man is talking => 0

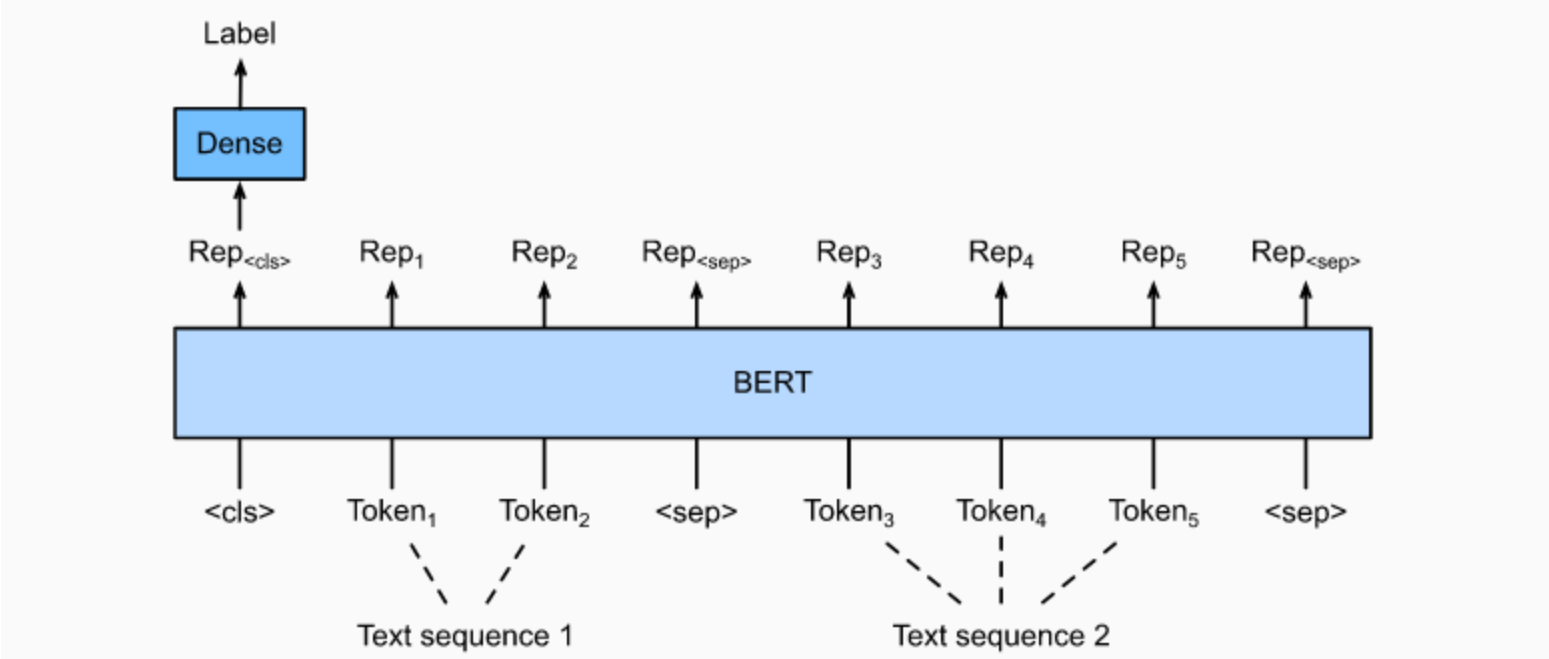


Fig. 15.6.2 Fine-tuning BERT for text pair classification or regression applications, such as natural language inference and semantic textual similarity. Suppose that the input text pair has two and three tokens.

15. 6. Fine-Tuning BERT

15. 6. 3. Text Tagging

- Each token needs to be assigned a label
- John Smith's car is new => NNP(noun), NNP, POS, NN, VB(verb), JJ (adjective)

- Every token of the input text is fed into the same extra fully-connected layers to output the label of the token

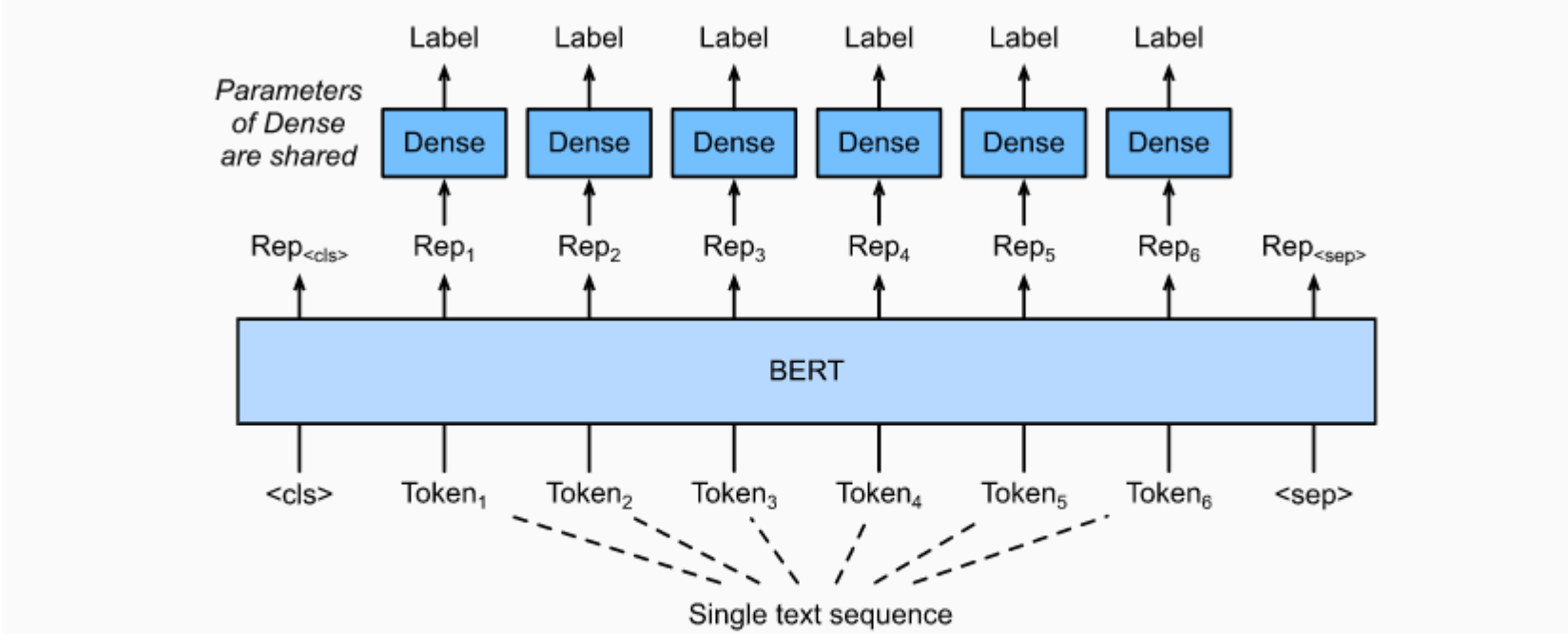


Fig. 15.6.3 Fine-tuning BERT for text tagging applications, such as part-of-speech tagging. Suppose that the input single text has six tokens.

15. 6. Fine-Tuning BERT

15. 6. 4. Question Answering

- **Passage:** Some experts report that a mask's efficacy is inconclusive. However, mask makers insist that their products, such as N95 respirator masks, can guard against the virus
- **Question:** Who say that N95 respirator masks can guard against the virus?
- Expected answer -> Mask maker (predict the start and end of the text span in the passage)
- Maximize log-likelihoods the score

  start-i + end-j (text span, i <= j)

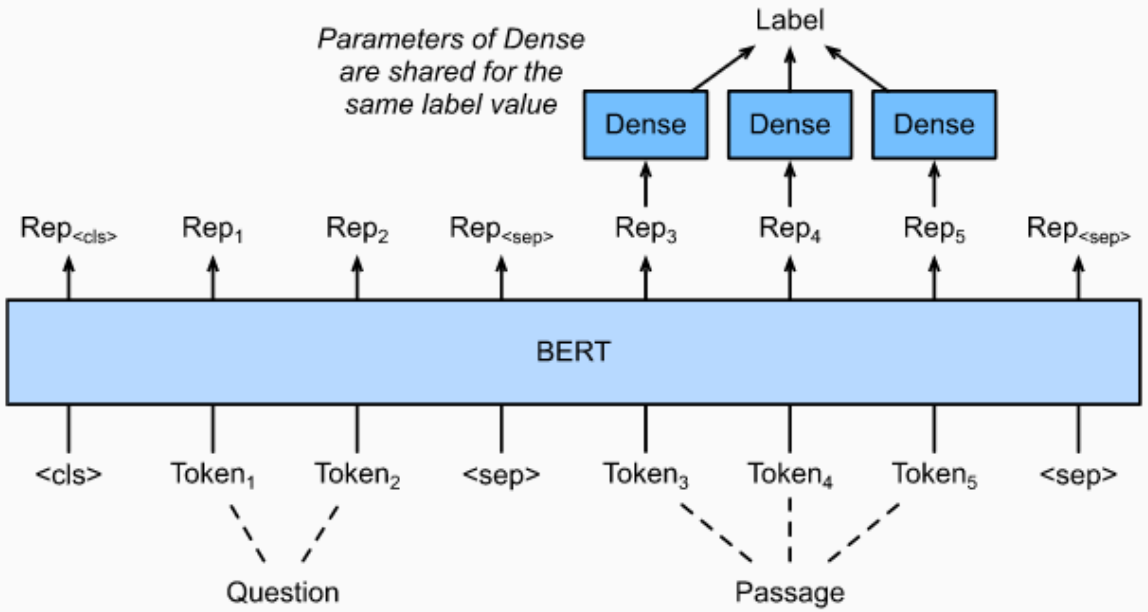
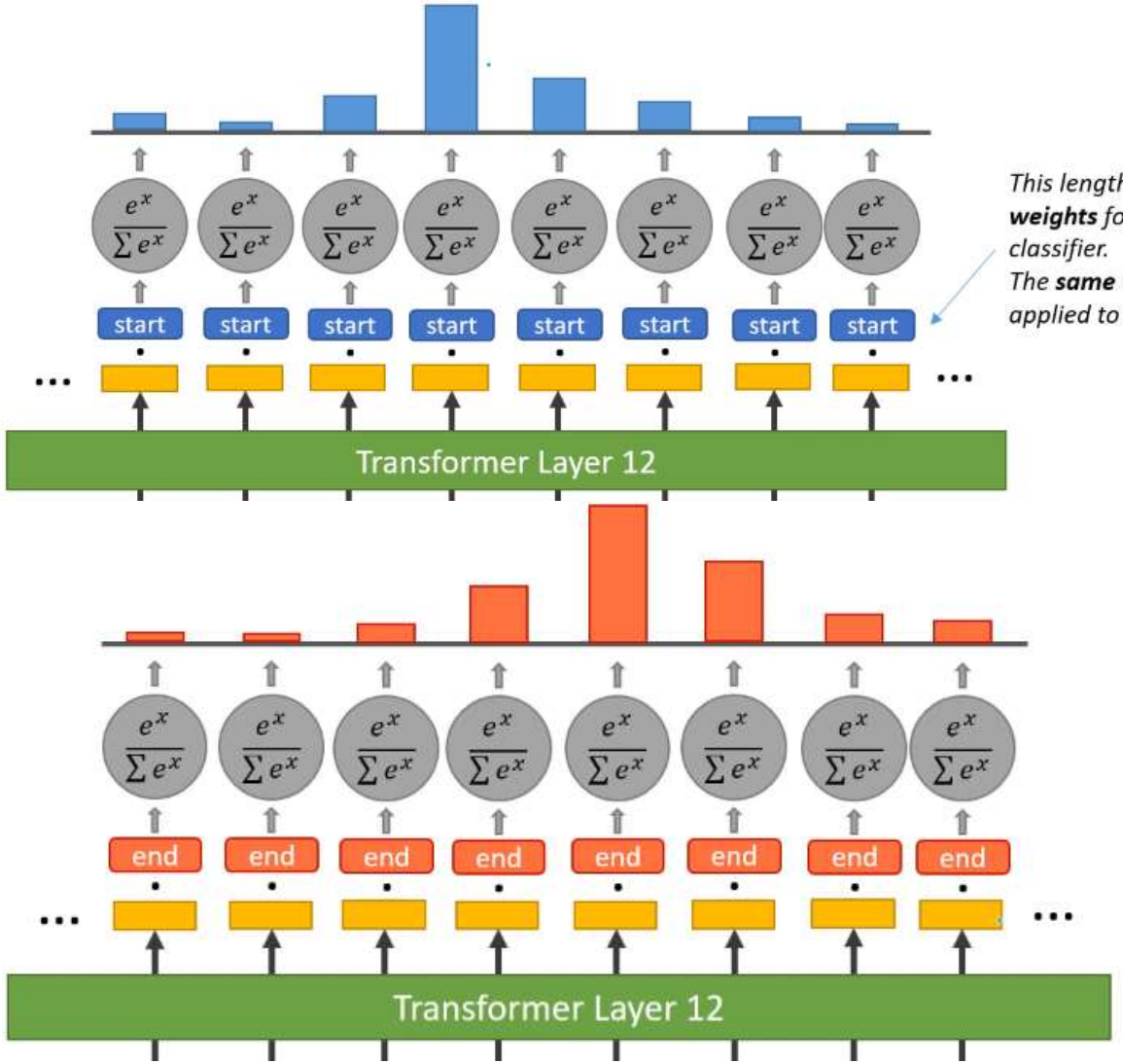


Fig. 15.6.4 Fine-tuning BERT for question answering. Suppose that the input text pair has two and three tokens.

15. 6. Fine-Tuning BERT

15. 6. 4. Question Answering



Applying BERT to Question Answering (SQuAD v1.1) - YouTube

15. 7. Natural Language Inference

Fine-Tuning BERT

- Fine-tuning BERT only requires an additional MLP-based architecture

```
class BERTClassifier(nn.Block):
    def __init__(self, bert):
        super(BERTClassifier, self).__init__()
        self.encoder = bert.encoder
        self.hidden = bert.hidden
        self.output = nn.Dense(3)

    def forward(self, inputs):
        tokens_X, segments_X, valid_lens_x = inputs
        encoded_X = self.encoder(tokens_X, segments_X, valid_lens_x)
        return self.output(self.hidden(encoded_X[:, 0, :]))
```

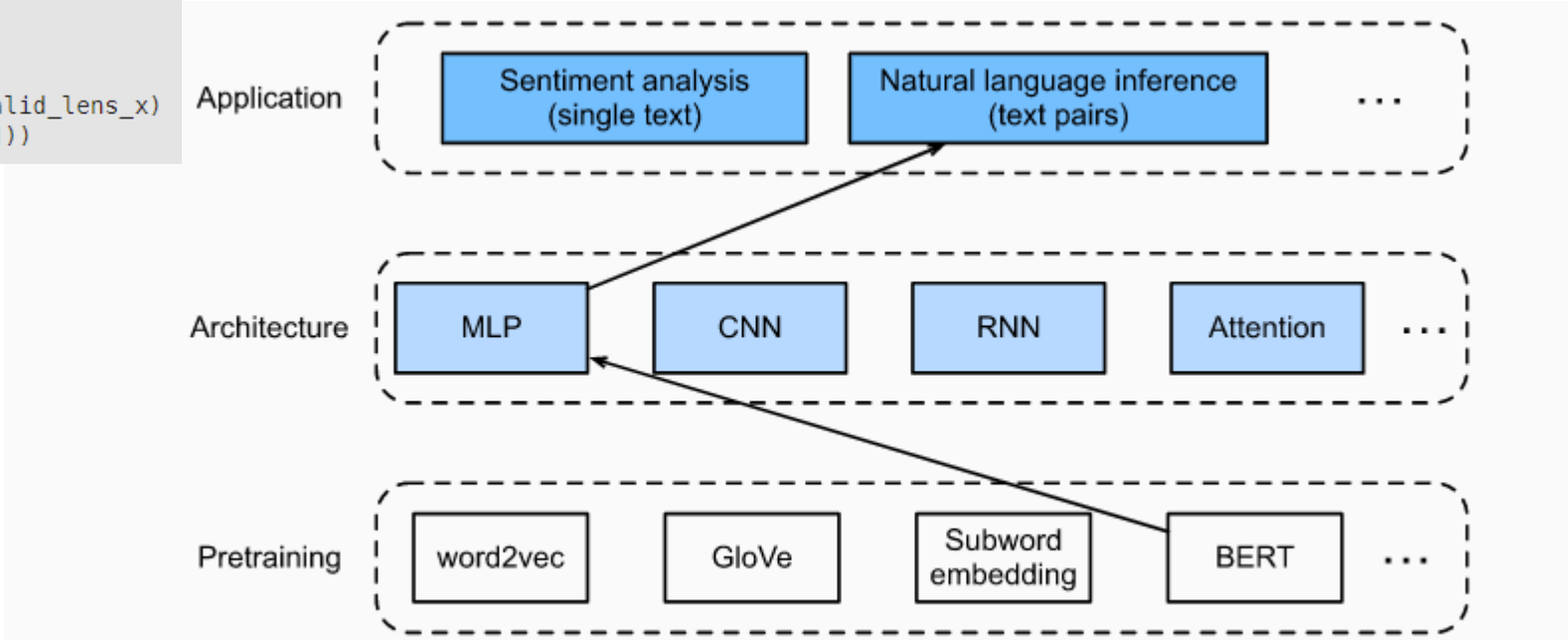


Fig. 15.7.1 This section feeds pretrained BERT to an MLP-based architecture for natural language inference.