

EE 628

Deep Learning

Spring 2020

Lecture 11

04/09/2020

Sergul Aydore

Applied Scientist

Amazon Web Services

Announcements

- Deadline for Projects **04/27/2020 Monday at 5pm ET**
 - project presentation on **04/30/2020** (40 %)
 - Details on grading: <https://github.com/sergulaydore/EE-628-Spring-2020>

Overview

- Last lecture we covered
 - GRUs and LSTMs
 - Attention Mechanism
 - Transformers
- Today, we will cover
 - Natural Language Processing: Pretraining & Applications
- Source material:
 - Dive into Deep Learning (<https://d2l.ai/>)
 - <https://github.com/sergulyaydore/EE-628-Spring-2020>

NATURAL LANGUAGE PROCESSING: Pretraining

- To understand text, we can begin with its representation
 - Treating each word as an individual text token
- We will see today that the representation of each token can be pretrained on a large corpus, using
 - Word2vec, GloVe, or subword embedding models
- However, after pretraining, representation of each token remains the same no matter what the context is
 - “bank” is same in both “go to the bank to deposit money” and “go to the bank to sit down”
- Thus, many more recent pretraining models adapt representation of the same token to different contexts.

NATURAL LANGUAGE PROCESSING: Pretraining

- Today, we will focus on how to pretrain such representations for text

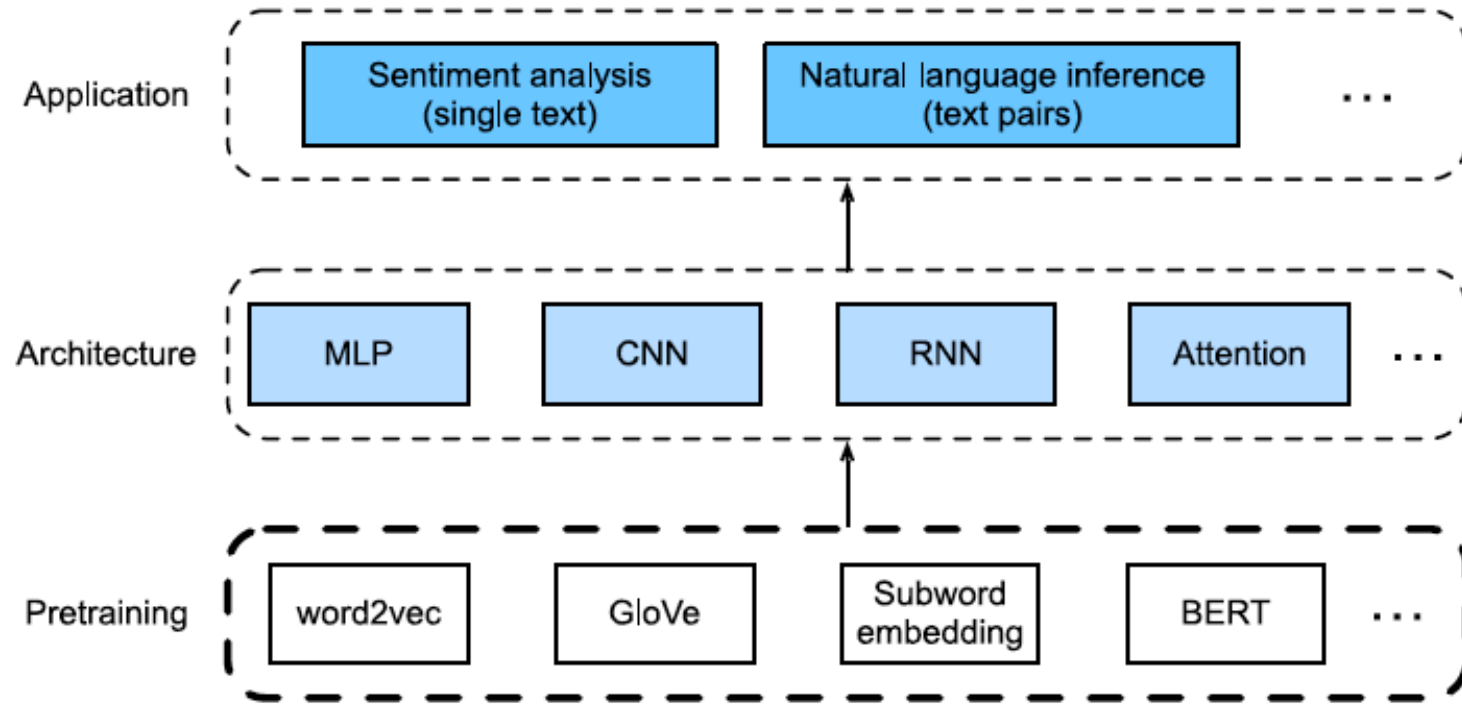


Fig. 14.1: Pretrained text representation can be fed to various deep learning architectures for different downstream natural language processing tasks. This chapter focuses on the upstream text representation pretraining.

Word Embedding (word2vec)

- A natural language is a complex system that we use to communicate.
- Words are commonly used as the unit of analysis in natural language processing.
- A word vector is a vector used to represent a word.
- The technique of mapping words to vectors of real numbers is also known as word embedding.
- Over the last few years, word embedding has gradually become basic knowledge in natural language processing.

Why not use One-hot vectors?

- We used one-hot vectors to represent words in our text processing examples.
- Each word is represented as a vector of length N (dictionary size) that can be used directly by the neural network.
- Although one-hot word vectors are easy to construct, they are usually not a good choice.
- The reason is that the one-hot word vectors cannot accurately express the similarity between different words.

Cosine Similarity

- For the vectors $\mathbf{x}, \mathbf{y} \in R^d$, their cosine similarities are the cosines of the angles between them:

$$\frac{\mathbf{x}^\top \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|} \in [-1, 1].$$

- Since the cosine similarity between the one-hot vectors of any two different words is 0, it is difficult to use the one-hot vector to accurately represent the similarity between multiple different words.

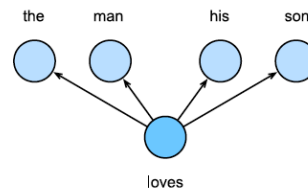
The Skip-Gram Model

- The skip-gram model assumes that a word can be used to generate the words that surround it in a text sequence.
- For example, assume that the text sequence is “the”, “man”, “loves”, “his”, and “son”. We use “loves” as the central target word and set the context window size to 2.
- The skip-gram model is concerned with the conditional probability for generating the context words within a distance of no more than 2 words

$$\mathbb{P}(\text{"the", "man", "his", "son"} \mid \text{"loves"}).$$

- We assume that, the context words are conditionally independent from each other.

$$\mathbb{P}(\text{"the"} \mid \text{"loves"}) \cdot \mathbb{P}(\text{"man"} \mid \text{"loves"}) \cdot \mathbb{P}(\text{"his"} \mid \text{"loves"}) \cdot \mathbb{P}(\text{"son"} \mid \text{"loves"}).$$



The Skip-Gram Model

- In the skip-gram model, each word is represented as **two** d -dimension vectors, which are used to compute the conditional probability.
- We assume that the word is indexed as i in the dictionary, its vector is represented as $\mathbf{v}_i \in R^d$ when it is the central target word, and $\mathbf{u}_i \in R^d$ when it is a context word.
- Let the central target word w_c and context word w_o be indexed as c and o respectively in the dictionary.
- The conditional probability of generating the context word for the given central target word can be obtained by performing a softmax operation on the vector inner product:

$$\mathbb{P}(w_o \mid w_c) = \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)},$$

The Skip-Gram Model

- Assume that a text sequence of length T is given, where the word at time step t is denoted as $w^{(t)}$.
- Assume that context words are independently generated given center words.
- When context window size is m , the likelihood function of the skip-gram model is the joint probability of generating all the context words given any center word

$$\prod_{t=1}^T \prod_{-m \leq j \leq m, j \neq 0} \mathbb{P}(w^{(t+j)} \mid w^{(t)}),$$

Skip-Gram Model Training

- The skip-gram model parameters are the central target word vector and context word vector for each individual word.
- In the training process, we are going to learn the model parameters by maximizing the likelihood function.
- This is equivalent to minimizing the following loss function: $-\sum_{t=1}^T \sum_{-m \leq j \leq m, j \neq 0} \log \mathbb{P}(w^{(t+j)} | w^{(t)})$.
- If we use the SGD, in each iteration we are going to pick a shorter subsequence through random sampling to compute the loss for that subsequence.
- By definition, we have $\log \mathbb{P}(w_o | w_c) = \mathbf{u}_o^\top \mathbf{v}_c - \log \left(\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c) \right)$
- After the training, for any word in the dictionary with index i , we are going to get its two word vector sets \mathbf{v}_i and \mathbf{u}_i .
- In applications of natural language processing (NLP), the central target word vector in the skip-gram model is generally used as the representation vector of a word.

The Continuous Bag of Words (CBOW) Model

- The continuous bag of words (CBOW) model is similar to the skip-gram model.
- The biggest difference is that the CBOW model assumes that the central target word is generated based on the context words before and after it in the text sequence.
- With the same text sequence “the”, “man”, “loves”, “his” and “son”, in which “loves” is the central target word, given a context window size of 2, the CBOW model is concerned with the conditional probability of generating the target word “loves” based on the context words “the”, “man”, “his” and “son”, such as

$$\mathbb{P}(\text{"loves"} \mid \text{"the"}, \text{"man"}, \text{"his"}, \text{"son"}).$$

- Since there are multiple context words in the CBOW model, we will average their word vectors and then use the same method as the skip-gram model to compute the conditional probability.

The Continuous Bag of Words (CBOW) Model

- We assume that $\mathbf{v}_i \in R^d$ and $\mathbf{u}_i \in R^d$ are the context word vector and central target word vector of the word with index i in the dictionary.
- Let central target vector w_c be indexed as c , and context words $w_{o_1}, \dots, w_{o_{2m}}$ be indexes as o_1, \dots, o_{2m} in the dictionary.
- Thus, the conditional probability of generating a central target word from the given context word is

$$\mathbb{P}(w_c \mid w_{o_1}, \dots, w_{o_{2m}}) = \frac{\exp\left(\frac{1}{2m} \mathbf{u}_c^\top (\mathbf{v}_{o_1} + \dots + \mathbf{v}_{o_{2m}})\right)}{\sum_{i \in \mathcal{V}} \exp\left(\frac{1}{2m} \mathbf{u}_i^\top (\mathbf{v}_{o_1} + \dots + \mathbf{v}_{o_{2m}})\right)}.$$

- For brevity, denote $W_o = \{w_{o_1}, \dots, w_{o_{2m}}\}$, and $\bar{\mathbf{v}}_o = (\mathbf{v}_{o_1}, \dots, \mathbf{v}_{o_{2m}})/2m$.
- The conditional probability equation can be simplified as:

$$\mathbb{P}(w_c \mid W_o) = \frac{\exp(\mathbf{u}_c^\top \bar{\mathbf{v}}_o)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \bar{\mathbf{v}}_o)}.$$

CBOW Model Training

- Given a text sequence of length T , we assume that the word at time step t is $w^{(t)}$, and the context window size is m .
- The likelihood function of the CBOW model is the probability of generating any central target word from the context words.

$$\prod_{t=1}^T \mathbb{P}(w^{(t)} \mid w^{(t-m)}, \dots, w^{(t-1)}, w^{(t+1)}, \dots, w^{(t+m)}).$$

- The maximum likelihood estimation of the CBOW model is equivalent to minimizing the loss function.

$$-\sum_{t=1}^T \log \mathbb{P}(w^{(t)} \mid w^{(t-m)}, \dots, w^{(t-1)}, w^{(t+1)}, \dots, w^{(t+m)}).$$

- Notice that $\log \mathbb{P}(w_c \mid \mathcal{W}_o) = \mathbf{u}_c^\top \bar{\mathbf{v}}_o - \log \left(\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \bar{\mathbf{v}}_o) \right)$
- We can compute the gradient of any context word vector v_{o_i} ($i = 1, \dots, 2m$).
- We then use the same method to obtain the gradients for other word vectors. Unlike the skip-gram model, we usually use the context word vector as the representation vector for a word in the CBOW model.

Approximate Training for Word2vec

- The core feature of the skip-gram model is the use of softmax operations to compute the conditional probability of generating context word w_o based on the given central target word w_c .

$$\mathbb{P}(w_o | w_c) = \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)}.$$

- The logarithmic loss corresponding to the conditional probability is given as

$$-\log \mathbb{P}(w_o | w_c) = -\mathbf{u}_o^\top \mathbf{v}_c + \log \left(\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c) \right)$$

- The loss above includes the sum of the number of items in the dictionary size.
- The gradient computation for each step contains the sum of the number of items in the dictionary size.
- For larger dictionaries with hundreds of thousands or even millions of words, the overhead for computing each gradient may be too high.
- In order to reduce such computational complexity, we will introduce two approximate training methods in this section: **negative sampling** and **hierarchical softmax**.

Negative Sampling

- Negative sampling modifies the original objective function.
- Given a context window for the central target word w_c , we will treat it as an event for context word w_o to appear in the context window and compute the probability of this event from $\mathbb{P}(D = 1 \mid w_c, w_o) = \sigma(\mathbf{u}_o^\top \mathbf{v}_c)$
- We will first consider training the word vector by maximizing the joint probability of all events in the text sequence.
- We consider maximizing the joint probability $\prod_{t=1}^T \prod_{-m \leq j \leq m, j \neq 0} \mathbb{P}(D = 1 \mid w^{(t)}, w^{(t+j)})$
- However, the events included in the model only consider positive examples.
- Negative sampling makes the objective function more meaningful by sampling with an addition of negative examples.

Negative Sampling

- We sample K words that do not appear in the context window to act as noise words.
- By considering negative sampling, we can rewrite the joint probability above, which only considers the positive examples, as

$$\prod_{t=1}^T \prod_{-m \leq j \leq m, j \neq 0} \mathbb{P}(w^{(t+j)} \mid w^{(t)})$$

- Here, the conditional probability is approximated to be

$$\mathbb{P}(w^{(t+j)} \mid w^{(t)}) = \mathbb{P}(D = 1 \mid w^{(t)}, w^{(t+j)}) \prod_{k=1, w_k \sim \mathbb{P}(w)}^K \mathbb{P}(D = 0 \mid w^{(t)}, w_k).$$

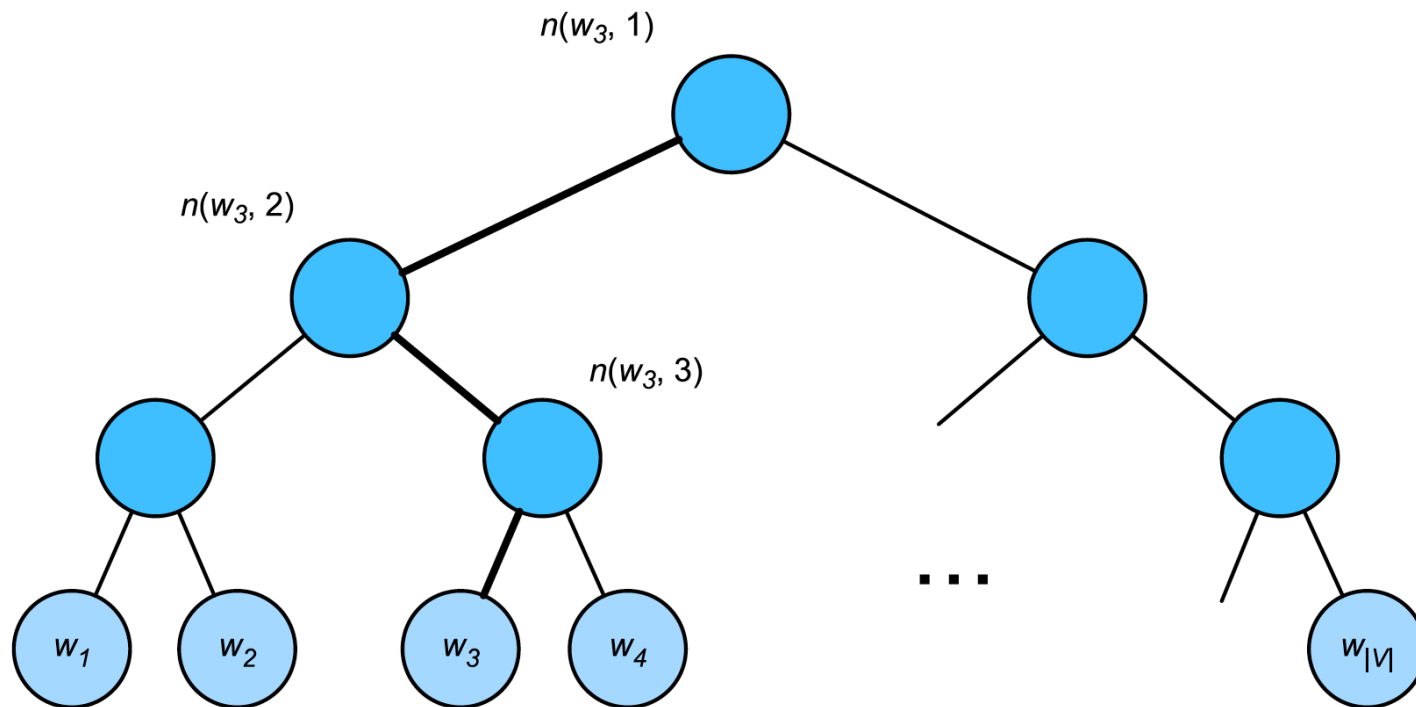
- The logarithmic loss for the conditional probability above is

$$-\log \mathbb{P}(w^{(t+j)} \mid w^{(t)}) = -\log \mathbb{P}(D = 1 \mid w^{(t)}, w^{(t+j)}) - \sum_{k=1, w_k \sim \mathbb{P}(w)}^K \log \mathbb{P}(D = 0 \mid w^{(t)}, w_k)$$

- Here, the gradient computation in each step of the training is no longer related to the dictionary size, but linearly related to K .

Hierarchical Softmax

- Hierarchical softmax is another type of approximate training method.
- It uses a binary tree for data structure, with the leaf nodes of the tree representing every word in the dictionary



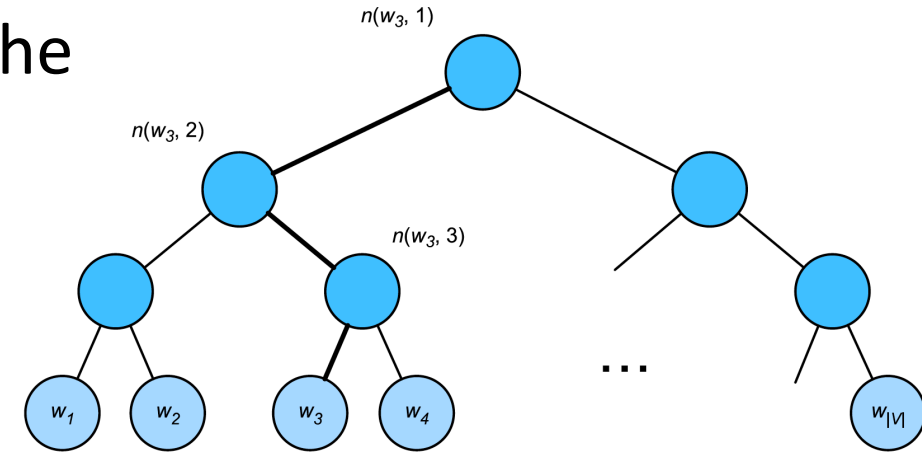
Hierarchical Softmax

- We assume that $L(w)$ is the number of nodes on the path from the root node of the binary tree to the leaf node of word w , e.g. $L(w_3) = 4$ in the figure.
- Let $n(w, j)$ be the j th node on this path, with the context word vector $\mathbf{u}_{n(w, j)}$.
- Hierarchical softmax will approximate the conditional probability in the skip-gram model as

$$\mathbb{P}(w_o \mid w_c) = \prod_{j=1}^{L(w_o)-1} \sigma \left(\mathbb{I}[n(w_o, j+1) = \text{leftChild}(n(w_o, j))] \cdot \mathbf{u}_{n(w_o, j)}^\top \mathbf{v}_c \right)$$

- Now, we can compute the conditional probability:

$$\mathbb{P}(w_3 \mid w_c) = \sigma(\mathbf{u}_{n(w_3, 1)}^\top \mathbf{v}_c) \cdot \sigma(-\mathbf{u}_{n(w_3, 2)}^\top \mathbf{v}_c) \cdot \sigma(\mathbf{u}_{n(w_3, 3)}^\top \mathbf{v}_c).$$



Word Embedding with Global Vectors (GloVe)

- First, we should review the skip-gram model in word2vec.
- The conditional probability $P(w_j|w_i)$ expressed in the skip-gram model using the softmax operation will be recorded as q_{ij} , that is:

$$q_{ij} = \frac{\exp(\mathbf{u}_j^\top \mathbf{v}_i)}{\sum_{k \in \mathcal{V}} \exp(\mathbf{u}_k^\top \mathbf{v}_i)},$$

- where \mathbf{v}_i and \mathbf{u}_i are the vector representations of word w_i of index i as the center word and the context word, respectively.
- For word w_i , it may appear in the data set for multiple times.
- We collect all the context words every time when w_i is a center word and keep duplicates, denoted as multiset \mathcal{C}_i
- The number of an element in a multiset is called the multiplicity of the element.

Word Embedding with Global Vectors (GloVe)

- For instance, suppose that word w_i appears twice in the data set: the context windows when these two w_i become center words in the text sequence contain context word indices $[2, 1, 5, 2]$ and $[2, 3, 2, 1]$.
- Then, multiset $C_i = \{1, 1, 2, 2, 2, 2, 3, 5\}$ where multiplicity of element 1 is 2, multiplicity of element 2 is 4, and multiplicities of elements 3 and 5 are both 1.
- Denote multiplicity of element j in multiset C_i as x_{ij} : it is the number of word w_j in all context windows for center word w_i in the entire dataset.
- As a result, the loss function of the skip-gram model can be expressed in a different way: $-\sum_{i \in \mathcal{V}} \sum_{j \in \mathcal{V}} x_{ij} \log q_{ij}$.
- We add up the number of all the context words for the central target word w_i to get x_i , and record the conditional probability x_{ij}/x_i as p_{ij} .
- We can rewrite the loss function of the skip-gram model as $-\sum_{i \in \mathcal{V}} x_i \sum_{j \in \mathcal{V}} p_{ij} \log q_{ij}$.

Word Embedding with Global Vectors (GloVe)

- In this formula, $-\sum_{j \in \mathcal{V}} p_{ij} \log q_{ij}$ is the cross-entropy between the conditional probability distribution p_{ij} for context word generation based on the central target word w_i and the cross-entropy of conditional probability distribution q_{ij} predicted by model.
- The loss function is weighted using the sum of the number of context words with the central target word w_i .
- However, the cross-entropy loss function is sometimes not a good choice.
 - On the one hand the cost of letting the model prediction q_{ij} has the sum of all items in the entire dictionary in its denominator. This can easily lead to excessive computational overhead.
 - On the other hand, there are often a lot of uncommon words in the dictionary, and they appear rarely in the data set. In the cross-entropy loss function, the final prediction of the conditional probability distribution on a large number of uncommon words is likely to be inaccurate.

The GloVe Model

- To address this, GloVe, a word embedding model that came after word2vec, adopts square loss and makes three changes to the skip-gram model based on this loss.
 1. Here, we use the non-probability distribution variables $p'_{ij} = x_{ij}$ and $q'_{ij} = \exp(u_j^T v_i)$ and take their logs. Therefore, we get the square loss $(\log p'_{ij} - \log q'_{ij})^2 = (\mathbf{u}_j^T \mathbf{v}_i - \log x_{ij})^2$
 2. We add two scalar model parameters for each word w_i : the bias terms b_i (for central target words) and c_i (for context words).
 3. Replace the weight of each loss with the function $h(x_{ij})$. The weight function $h(x)$ is a monotone increasing function with the range $[0, 1]$
- Therefore, the goal of GloVe is to minimize the loss function.

$$\sum_{i \in \mathcal{V}} \sum_{j \in \mathcal{V}} h(x_{ij}) (\mathbf{u}_j^T \mathbf{v}_i + b_i + c_j - \log x_{ij})^2$$

The GloVe Model

- Here, we have a suggestion for the choice of weight function $h(x)$: when $x < c$ (e.g. $c = 100$) make $h(x) = \left(\frac{x}{c}\right)^\alpha$ (e.g. $\alpha = 0.75$), otherwise make $h(x) = 1$.
- Because $h(0) = 0$, the squared loss term for $x_{ij} = 0$ can simply be ignored.
- The non-zero x_{ij} are computed in advance based on the entire dataset and they contain global statistics for the data set.
- Therefore, the name GloVe is taken from “Global Vectors”.
- Notice that if word w_i appears in the context window of word w_j , then w_j will also appear in the context window of w_i . Therefore, $x_{ij} = x_{ji}$.
- Unlike word2vec, GloVe fits the symmetric $\log x_{ij}$ while the conditional probabilities p_{ij} are asymmetric.
- Therefore, the central target word vector and context word vector of any word are equivalent in GloVe.
- However, the two sets of word vectors that are learned by the same word may be different in the end due to different initialization values.
- After learning all the word vectors, GloVe will use the sum of the central target word vector and the context word vector as the final word vector for the word.

Understanding GloVe from Conditional Probability Ratios

- From a real example from a large corpus, here we have the following two sets of conditional probabilities with “ice” and “steam” as the central target words and the ratio between them:

$w_k =$	“solid”	“gas”	“water”	“fashion”
$p_1 = \mathbb{P}(w_k \mid \text{“ice”})$	0.00019	0.000066	0.003	0.000017
$p_2 = \mathbb{P}(w_k \mid \text{“steam”})$	0.000022	0.00078	0.0022	0.000018
p_1/p_2	8.9	0.085	1.36	0.96

- We can see that the conditional probability ratio can represent the relationship between different words more intuitively.
- We can construct a word vector function to fit the conditional probability ratio more effectively.
- The conditional probability ratio with w_i as the central target word is p_{ij}/p_{ik} .
- We can find a function that uses word vectors to fit this conditional probability ratio. $f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) \approx \frac{p_{ij}}{p_{ik}}$.

Understanding GloVe from Conditional Probability Ratios

- The possible design of function f here will not be unique.
- We only need to consider a more reasonable possibility.
- Notice that the conditional probability ratio is a scalar, we can limit f to be a scalar function: $f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) = f((\mathbf{u}_j - \mathbf{u}_k)^\top \mathbf{v}_i)$
- One possibility is to use exponential function so that the ratio is 1 when $j = k$.

$$f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) = \frac{\exp(\mathbf{u}_j^\top \mathbf{v}_i)}{\exp(\mathbf{u}_k^\top \mathbf{v}_i)} \approx \frac{p_{ij}}{p_{ik}}.$$

- One possibility that satisfies the right side of the approximation is $\exp(\mathbf{u}_j^\top \mathbf{v}_i) \approx \alpha p_{ij}$ where α is a constant.

- Considering that $p_{ij} = x_{ij}/x_i$, after taking the logarithm we get

$$\mathbf{u}_j^\top \mathbf{v}_i \approx \log \alpha + \log x_{ij} - \log x_i$$

Understanding GloVe from Conditional Probability Ratios

$$\mathbf{u}_j^\top \mathbf{v}_i \approx \log \alpha + \log x_{ij} - \log x_i$$

- We use additional bias terms to fit $\log \alpha - \log x_i$, such as the central target word bias term b_i and context word bias term c_j :

$$\mathbf{u}_j^\top \mathbf{v}_i + b_i + c_j \approx \log(x_{ij})$$

- By using formula below, we can get the loss function of GloVe.

$$\sum_{i \in \mathcal{V}} \sum_{j \in \mathcal{V}} h(x_{ij}) (\mathbf{u}_j^\top \mathbf{v}_i + b_i + c_j - \log x_{ij})^2$$

Subword Embedding

- Words usually have internal structures and formation methods.
- For example, we can deduce the relationship between “dog”, “dogs” and “dogcatcher” by their spelling.
- This association can be extended to other words,
- For example, the relationship between “dog” and “dogs” is just like the relationship between “cat” and “cats”.
- In fact, morphology studies the internal structure and formation of words.

Subword Embedding - fastText

- In word2vec, we did not directly use morphology information.
- For example, “dog” and “dogs” are represented by two different vectors.
- In view of this, fastText (Bojanowski et al., 2017) proposes the method of subword embedding.

Enriching Word Vectors with Subword Information

Piotr Bojanowski* and **Edouard Grave*** and **Armand Joulin** and **Tomas Mikolov**

Facebook AI Research

`{bojanowski, egrave, ajoulin, tmikolov}@fb.com`

Subword Embedding - fastText

- In fastText, each central word is represented as a collection of subwords.
- Below we use the word “where” as an example to understand how subwords are formed.
- First, we add “<”, “>” at the beginning and end of the word to distinguish the subwords used as prefixes and suffices.
- Then, we treat the word as a sequence of characters to extract the n-grams.
- For example, when $n=3$, we can get all subwords with a length of 3:

"<wh", "whe", "her", "ere", "re>"

and the special subword “<where>”.

Subword Embedding - fastText

- In fastText, for a word w , we record the union of all its subwords with length of 3 to 6 and special subwords as G_w .
- Thus, the dictionary is the union of the collection of subwords of all words.
- Assume the vector of the subword g in the dictionary is \mathbf{z}_g
 - Then, the central word vector \mathbf{u}_w for the word w in the skip-gram model can be expressed as

$$\mathbf{u}_w = \sum_{g \in G_w} \mathbf{z}_g.$$

- The rest of the fastText is consistent with the skip-gram model.
- The dictionary in fastText is larger, resulting in more model parameters.
- Also, the vector of one word requires the summation of all subword vectors, which results in higher computation complexity.

Bidirectional Encoder Representations from Transformers (BERT)

- We have introduced several word embedding models.
- These word embedding models are all context independent.
- Context-independent representations have obvious limitations.
 - For instance, the word “crane” in contexts “a crane is flying” and “a crane driver came” has completely different meanings.
 - Thus, the same word may be assigned different representations depending on contexts.
- This motivates the development of context-sensitive word representations, where representations of words depend on their contexts.
- Hence, a context-sensitive representation of token x is a function $f(x, c(x))$ depending on both x and its context $c(x)$.

Bidirectional Encoder Representations from Transformers (BERT)

- Popular context-sensitive representations include TagLM (language-model-augmented sequence tagger), CoVe (Context Vectors), and ELMo (Embeddings from Language Models)
- For example, ELMo combines all the intermediate layer representations from pretrained bidirectional LSTM as the output representation.
- Then the ELMo representation will be added to a downstream task's existing supervised model as additional features, such as by concatenating ELMo representation and the original representations.
- However, each solution still hinges on a task-specific architecture.
- It is practically non-trivial to craft a specific architecture for every natural language processing task.

Bidirectional Encoder Representations from Transformers (BERT)

- The GPT (Generative Pre-training) model represents an effort in designing a general task-agnostic model for context-sensitive representations.
- Built on a Transformer decoder, GPT pretrains a language model that will be used to represent text sequences.
- Contrast to ELMo that freezes parameters of the pretrained model, GPT fine-tunes all the parameters in the pretrained Transformer decoder during supervised learning of the downstream task,
- However, due to the autoregressive nature of language models, GPT only looks forward.
- GPT will return the same representation for “bank” in contexts:
 - “I went to the bank to deposit cash” and “I went to the bank to sit down”

Bidirectional Encoder Representations from Transformers (BERT)

- ELMo encodes context bidirectionally but uses task-specific architectures
- GPT is task-agnostic but encodes context left-to-right
- Combining the best of both worlds, BERT encodes context bidirectionally and requires minimal architecture changes for a wide range of natural language processing tasks.

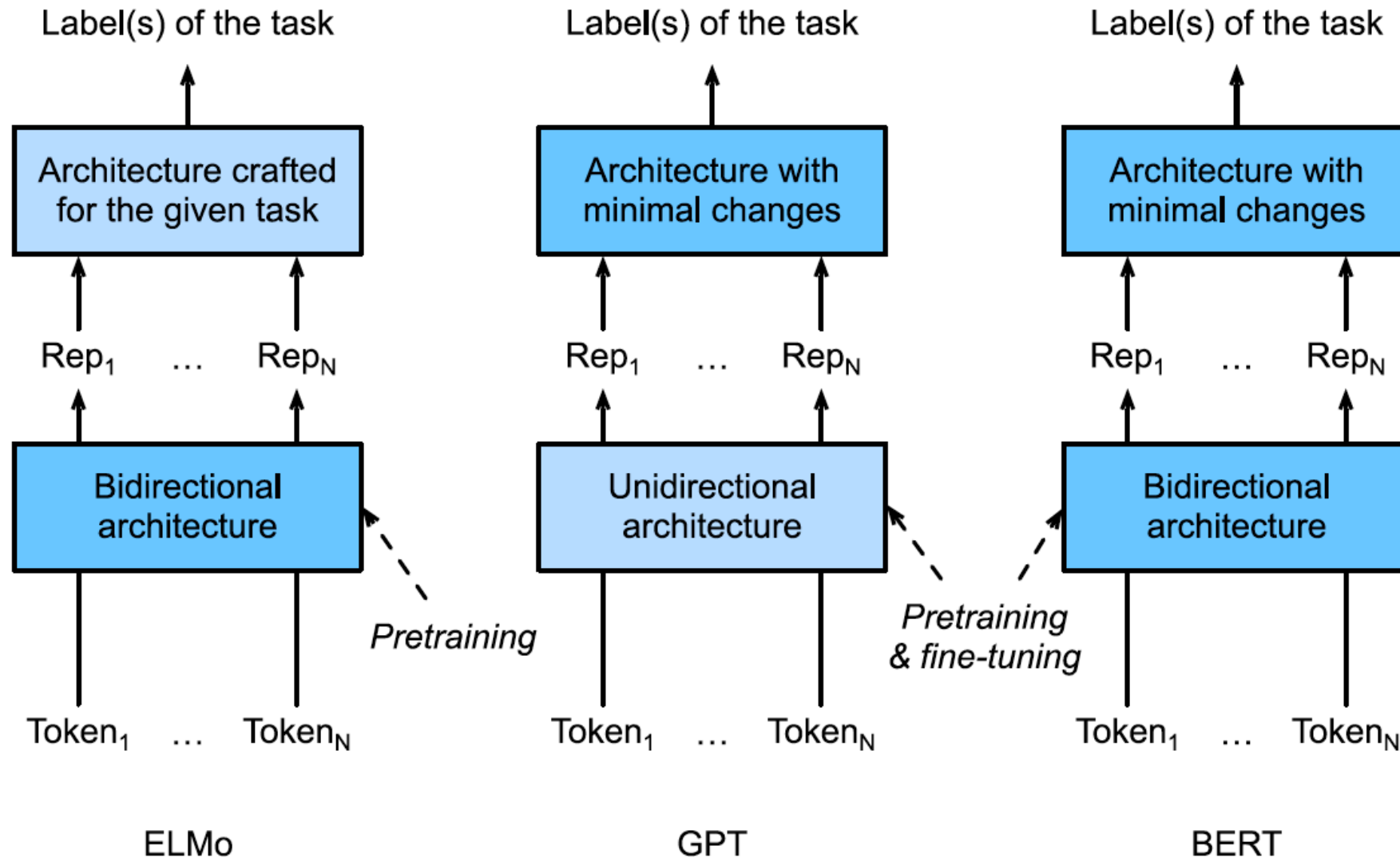
BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding

Jacob Devlin Ming-Wei Chang Kenton Lee Kristina Toutanova

Google AI Language

`{jacobdevlin, mingweichang, kentonl, kristout}@google.com`

Bidirectional Encoder Representations from Transformers (BERT)



BERT – Input representation

- One BERT input sequence may include either one text sequence or two text sequences
 - some tasks (e.g. sentiment analysis) take single text as the input
 - some other tasks (e.g. natural language inference), the input is a pair of text sequences
- To distinguish text pairs, the learner segment embeddings \mathbf{e}_A and \mathbf{e}_B are added to the token embeddings. For single text inputs, only \mathbf{e}_A is used.
- BERT chooses the Transformer encoder as its bidirectional architecture.
- BERT uses learnable positional embeddings unlike the original Transformer encoder.

BERT – Input representation

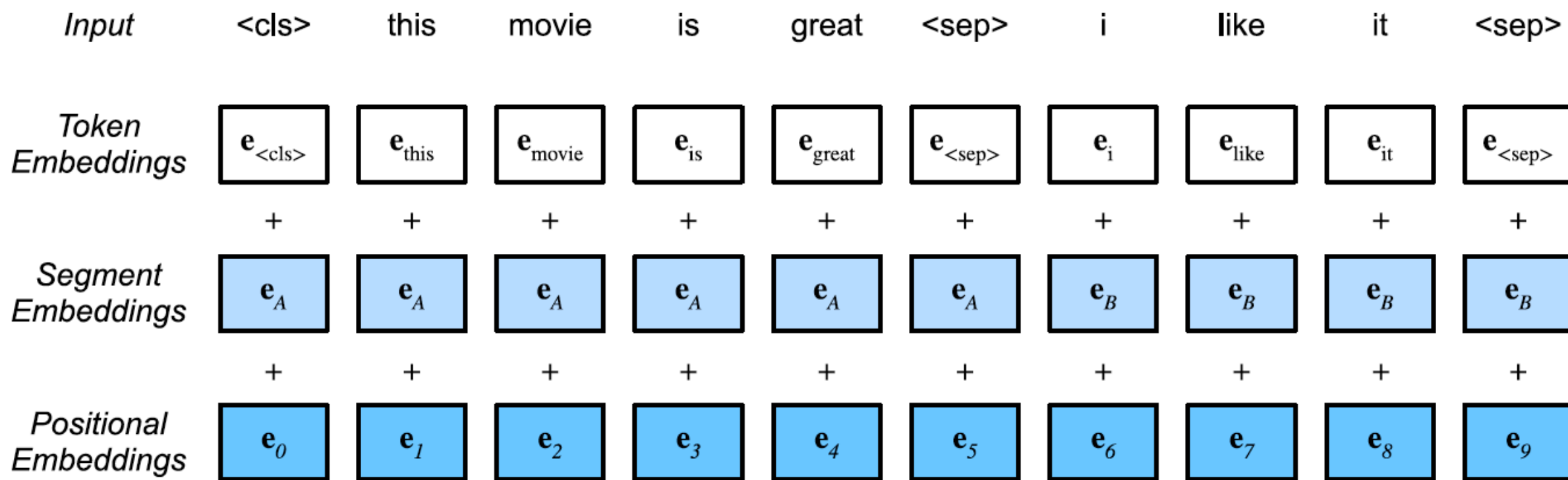


Fig. 14.8.2: The embeddings of the BERT input sequence are the sum of the token embeddings, segment embeddings, and positional embeddings.

BERT – Pretraining Tasks

- The forward propagation of BERTEncoder gives the BERT representation of each token of the input text and the inserted special tokens.
- Next, we will use these representations to compute the loss function for pretraining BERT.
- The pretraining is composed of the following to tasks:
 - Masked language modeling
 - Next sentence prediction

BERT – Pretraining Tasks – Masked Language Modeling

- To encode context bidirectionally for representing each token, BERT randomly masks tokens and uses tokens from the bidirectional context to predict masked tokens.
- In this pretraining task, 15% of tokens will be selected at random as the masked tokens for prediction.
- Assume “great” is selected to be masked and predicted in “this movie is great”. Then, the input will be replaced with
 - a special “<mask>” token for 80% of the time (e.g., “this movie is great” becomes “this movie is <mask>”);
 - a random token for 10% of the time (e.g., “this movie is great” becomes “this movie is drink”);
 - the unchanged label token for 10% of the time (e.g., “this movie is great” becomes “this movie is great”).

BERT – Pretraining Tasks – Next Sentence Prediction

- Although masked language modeling is able to encode bidirectional context for representing words, it does not explicitly model the logical relationship between text pairs.
- To help understand the relationship between two text sequences, BERT considers a binary classification task, next sentence prediction, in its pretraining.
- When generating sentence pairs for pretraining,
 - for half of the time they are indeed consecutive sentences with the label “True”;
 - while for the other half of the time the second sentence is randomly sampled from the corpus with the label “False”.

BERT – Putting All Things Together

- When pretraining BERT, the final loss function is a linear combination of both the loss functions for masked language modeling and next sentence prediction.

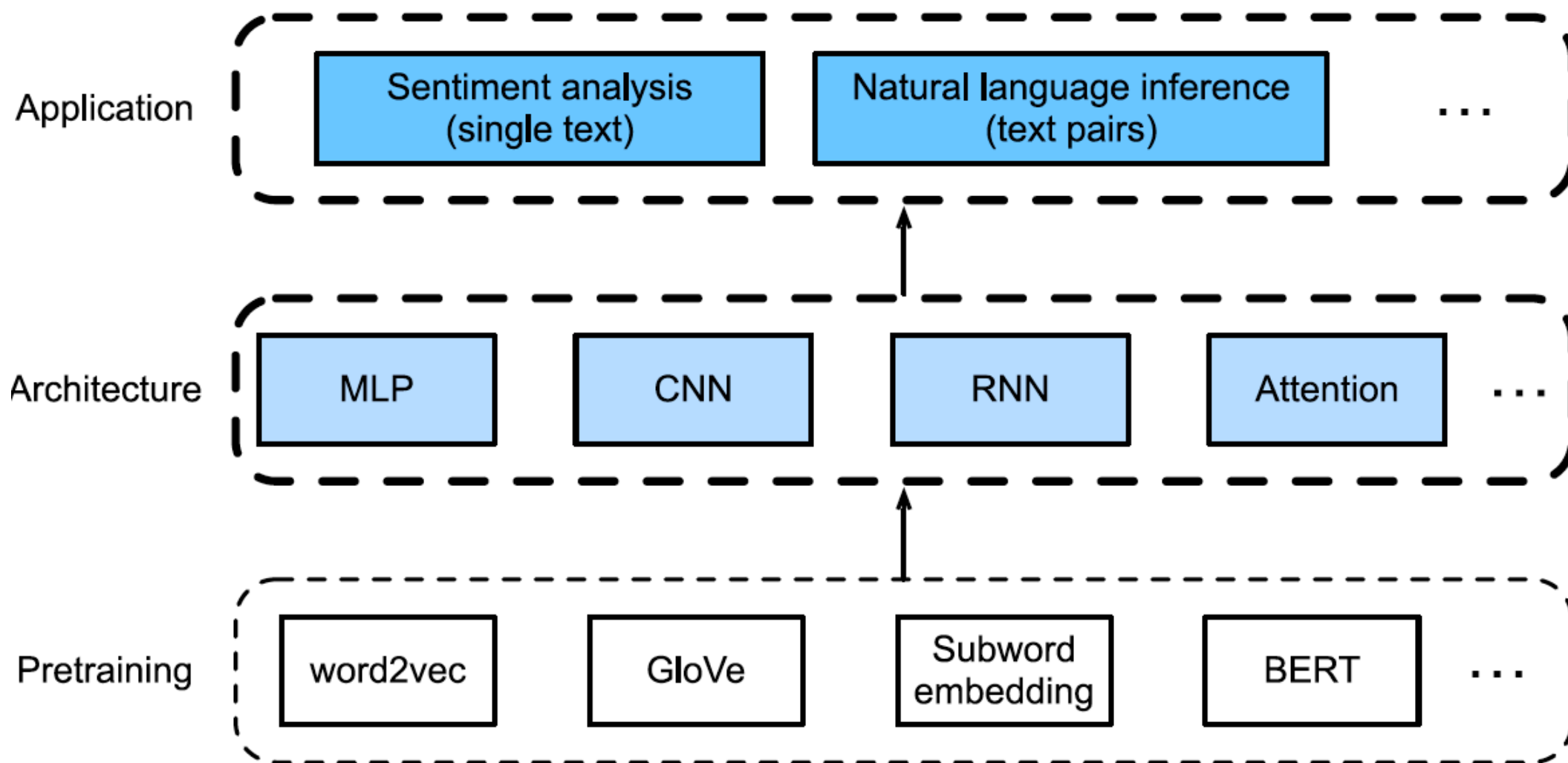
```
# Saved in the d2l package for later use
class BERTModel(nn.Block):
    def __init__(self, vocab_size, num_hiddens, ffn_num_hiddens, num_heads,
                  num_layers, dropout, max_len=1000):
        super(BERTModel, self).__init__()
        self.encoder = BERTEncoder(vocab_size, num_hiddens, ffn_num_hiddens,
                                    num_heads, num_layers, dropout, max_len)
        self.mlm = MaskLM(vocab_size, num_hiddens)
        self.nsp = NextSentencePred(num_hiddens)

    def forward(self, tokens, segments, valid_lens=None, pred_positions=None):
        encoded_X = self.encoder(tokens, segments, valid_lens)
        if pred_positions is not None:
            mlm_Y_hat = self.mlm(encoded_X, pred_positions)
        else:
            mlm_Y_hat = None
        nsp_Y_hat = self.nsp(encoded_X)
        return encoded_X, mlm_Y_hat, nsp_Y_hat
```

Natural Language Processing: Applications

- We have seen how to represent tokens and train their representations.
- Such trained text representation can be fed to various models for different downstream natural language processing tasks.
- We have already discussed several NLP applications without pretraining in earlier classes to discuss RNNs.
- Given pretrained text representation, we will consider two more downstream NLP tasks:
 - Sentiment analysis: analyzes single text
 - Natural language inference: analyzes relationship of text pairs

Natural Language Processing: Applications



Semtiment Analysis: Text Classification

- Text classification is a common task in natural language processing, which transforms a sequence of text of indefinite length into a category of text.
- By using text sentiment classification we can analyze the emotions of the text's author.
- This problem is also called sentiment analysis and has a wide range of applications.
- For example, we can analyze user reviews of products to obtain user satisfaction statistics, or analyze user sentiments about market conditions and use it to predict future trends.

Text Sentiment Classification Data

- We use Stanford's Large Movie Review Dataset as the data set for text sentiment classification
- This dataset is divided into two data sets for training and testing purposes, each containing 25,000 movie reviews downloaded from IMDb.
- In each data set, the number of comments labeled as “positive” and “negative” is equal.

Reading Data

- Data is downloaded from Stanford's server

```
# Save to the d2l package.
def download_imdb(data_dir='../data'):
    url = 'http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz'
    fname = gluon.utils.download(url, data_dir)
    with tarfile.open(fname, 'r') as f:
        f.extractall(data_dir)

download_imdb()
```


Reading Data

- Next, read the training and test data sets. Each example is a review and its corresponding label: 1 indicates “positive” and 0 indicates “negative”.

```
# Save to the d2l package.
def read_imdb(folder='train', data_dir='../data'):
    data, labels = [], []
    for label in ['pos', 'neg']:
        folder_name = os.path.join(data_dir, 'aclImdb', folder, label)
        for file in os.listdir(folder_name):
            with open(os.path.join(folder_name, file), 'rb') as f:
                review = f.read().decode('utf-8').replace('\n', '')
                data.append(review)
                labels.append(1 if label == 'pos' else 0)
    return data, labels

train_data = read_imdb('train')
print('# trainings:', len(train_data[0]))
for x, y in zip(train_data[0][:3], train_data[1][:3]):
    print('label:', y, 'review:', x[0:60])
```

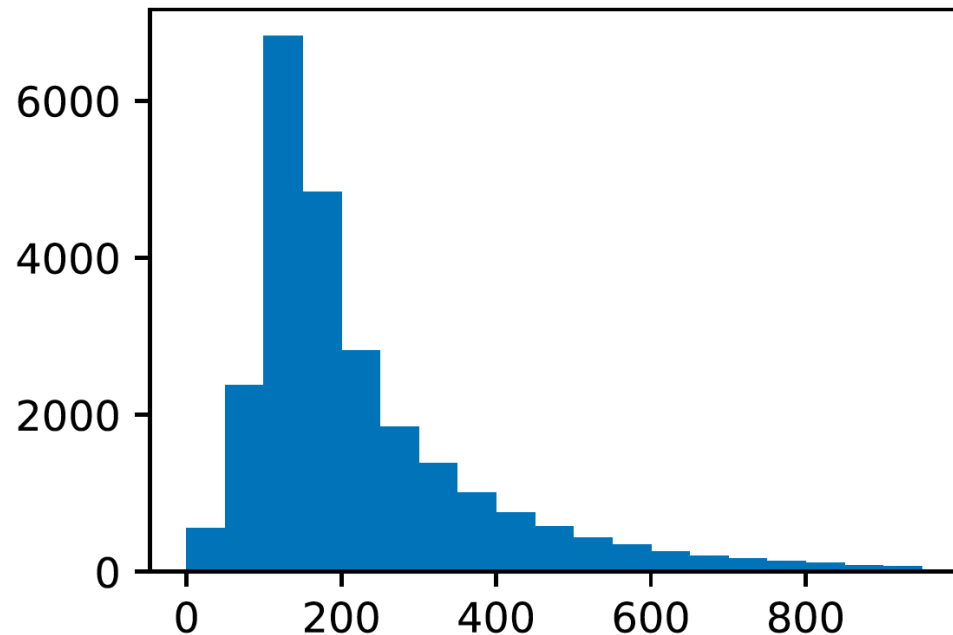
```
# trainings: 25000
label: 1 review: Normally the best way to annoy
label: 1 review: The Bible teaches us that the
label: 1 review: Being someone who lists Night
```

Tokenization and Vocabulary

- We use a word as a token, and then create a dictionary based on the training data set.

```
train_tokens = d2l.tokenize(train_data[0], token='word')
vocab = d2l.Vocab(train_tokens, min_freq=5)

d2l.set_figsize((3.5, 2.5))
d2l.plt.hist([len(line) for line in train_tokens], bins=range(0,1000,50));
```



Padding to the Same Length

- Because the reviews have different lengths, so they cannot be directly combined into mini-batches. Here we fix the length of each comment to 500 by truncating or adding “<unk>” indices.

```
num_steps = 500  # sequence length
train_features = nd.array([d2l.trim_pad(vocab[line], num_steps, vocab.unk)
                           for line in train_tokens])
train_features.shape
```

```
(25000, 500)
```

Create Data Iterator

- Now, we will create a data iterator. Each iteration will return a mini-batch of data.

```
train_iter = d2l.load_array((train_features, train_data[1]), 64)

for X, y in train_iter:
    print('X', X.shape, 'y', y.shape)
    break
'# batches:', len(train_iter)
```

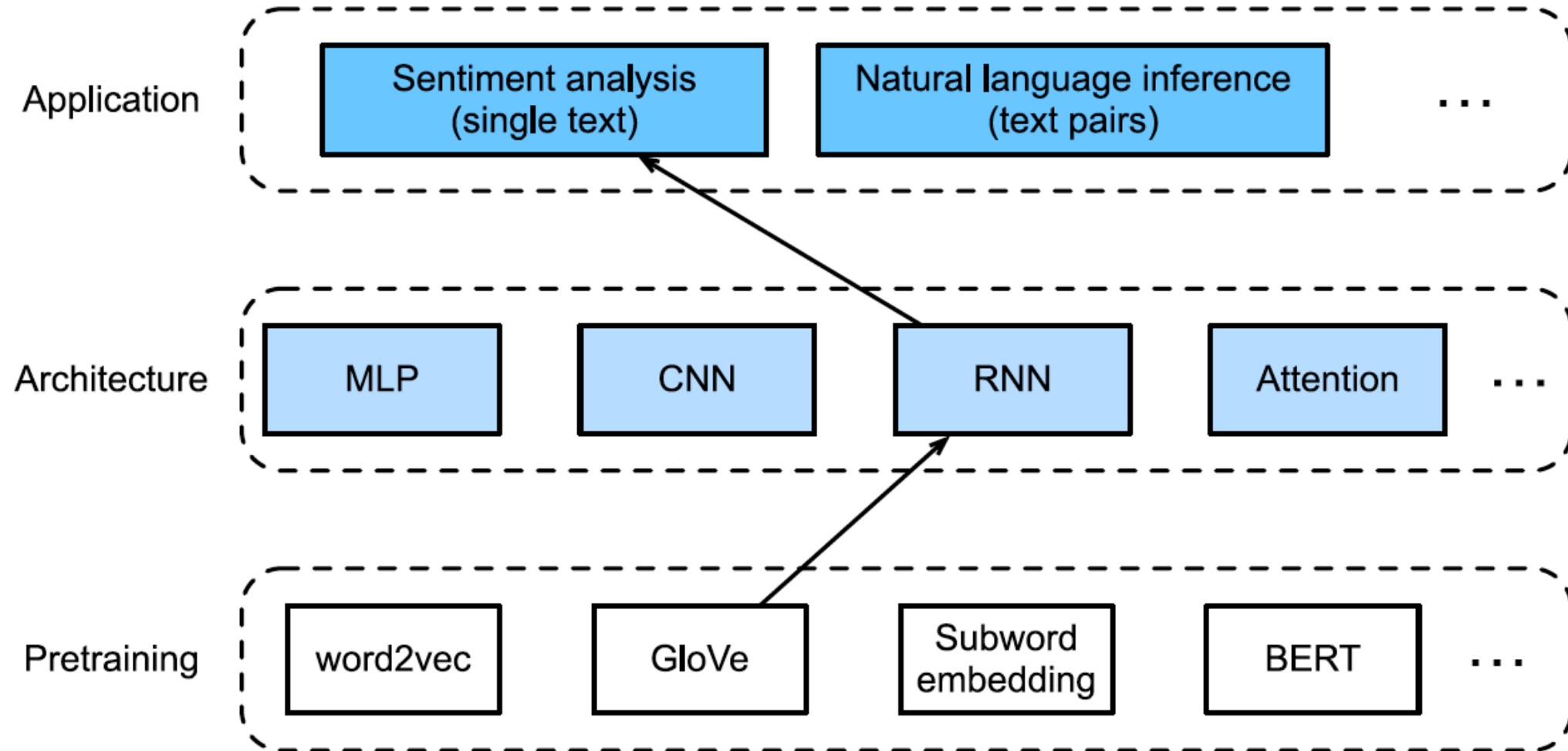
```
X (64, 500) y (64,)
```

```
('# batches:', 391)
```

Put All Things Together

```
# Save to the d2l package.
def load_data_imdb(batch_size, num_steps=500):
    download_imdb()
    train_data, test_data = read_imdb('train'), read_imdb('test')
    train_tokens = d2l.tokenize(train_data[0], token='word')
    test_tokens = d2l.tokenize(test_data[0], token='word')
    vocab = d2l.Vocab(train_tokens, min_freq=5)
    train_features = nd.array([d2l.trim_pad(vocab[line], num_steps, vocab.unk)
                               for line in train_tokens])
    test_features = nd.array([d2l.trim_pad(vocab[line], num_steps, vocab.unk)
                              for line in test_tokens])
    train_iter = d2l.load_array((train_features, train_data[1]), batch_size)
    test_iter = d2l.load_array((test_features, test_data[1]), batch_size,
                               is_train=False)
    return train_iter, test_iter, vocab
```

Text Sentiment Classification: Using Recurrent Neural Networks



Text Sentiment Classification: Using Recurrent Neural Networks

- Text classification is also a downstream application of word embedding.
- Here, we will apply pre-trained word vectors (GloVe) and bidirectional recurrent neural networks with multiple hidden layers.
- We will use them to determine whether a text sequence of indefinite length contains positive or negative emotion.

```
import d2l
from mxnet import gluon, init, nd
from mxnet.gluon import nn, rnn
from mxnet.contrib import text

batch_size = 64
train_iter, test_iter, vocab = d2l.load_data_imdb(batch_size)
```

Use a Recurrent Neural Network Model

- Each word first obtains a feature vector from the embedding layer
- Then, we further encode the feature sequence using a bidirectional recurrent neural network to obtain sequence information.
- Finally, we transform the encoded sequence information to output through the fully connected layer.

```
class BiRNN(nn.Block):  
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers, **kwargs):  
        super(BiRNN, self).__init__(**kwargs)  
        self.embedding = nn.Embedding(vocab_size, embed_size)  
        # Set Bidirectional to True to get a bidirectional recurrent neural  
        # network  
        self.encoder = rnn.LSTM(num_hiddens, num_layers=num_layers,  
                                bidirectional=True, input_size=embed_size)  
        self.decoder = nn.Dense(2)
```


Use a Recurrent Neural Network Model

- Specifically, we can concatenate hidden states of bidirectional long-short term memory in the initial timestep and final timestep and pass it to the output layer classification.

```
def forward(self, inputs):
    # The shape of inputs is (batch size, number of words). Because LSTM
    # needs to use sequence as the first dimension, the input is
    # transformed and the word feature is then extracted. The output shape
    # is (number of words, batch size, word vector dimension).
    embeddings = self.embedding(inputs.T)
    # Since the input (embeddings) is the only argument passed into
    # rnn.LSTM, it only returns the hidden states of the last hidden layer
    # at different timestep (outputs). The shape of outputs is
    # (number of words, batch size, 2 * number of hidden units).
    outputs = self.encoder(embeddings)
    # Concatenate the hidden states of the initial timestep and final
    # timestep to use as the input of the fully connected layer. Its
    # shape is (batch size, 4 * number of hidden units)
    encoding = np.concatenate((outputs[0], outputs[-1]), axis=1)
    outs = self.decoder(encoding)
    return outs
```

Load Pre-trained Word Vectors

- We will directly use word vectors pre-trained on a larger corpus as the feature vectors of all words.
- Here, we load a 100-dimensional GloVe word vector for each word in the dictionary vocab.

```
glove_embedding = text.embedding.create(  
    'glove', pretrained_file_name='glove.6B.100d.txt')
```

Query the word vectors that in our vocabulary.

```
embeds = glove_embedding.get_vecs_by_tokens(vocab.idx_to_token)  
embeds.shape
```

```
(49339, 100)
```

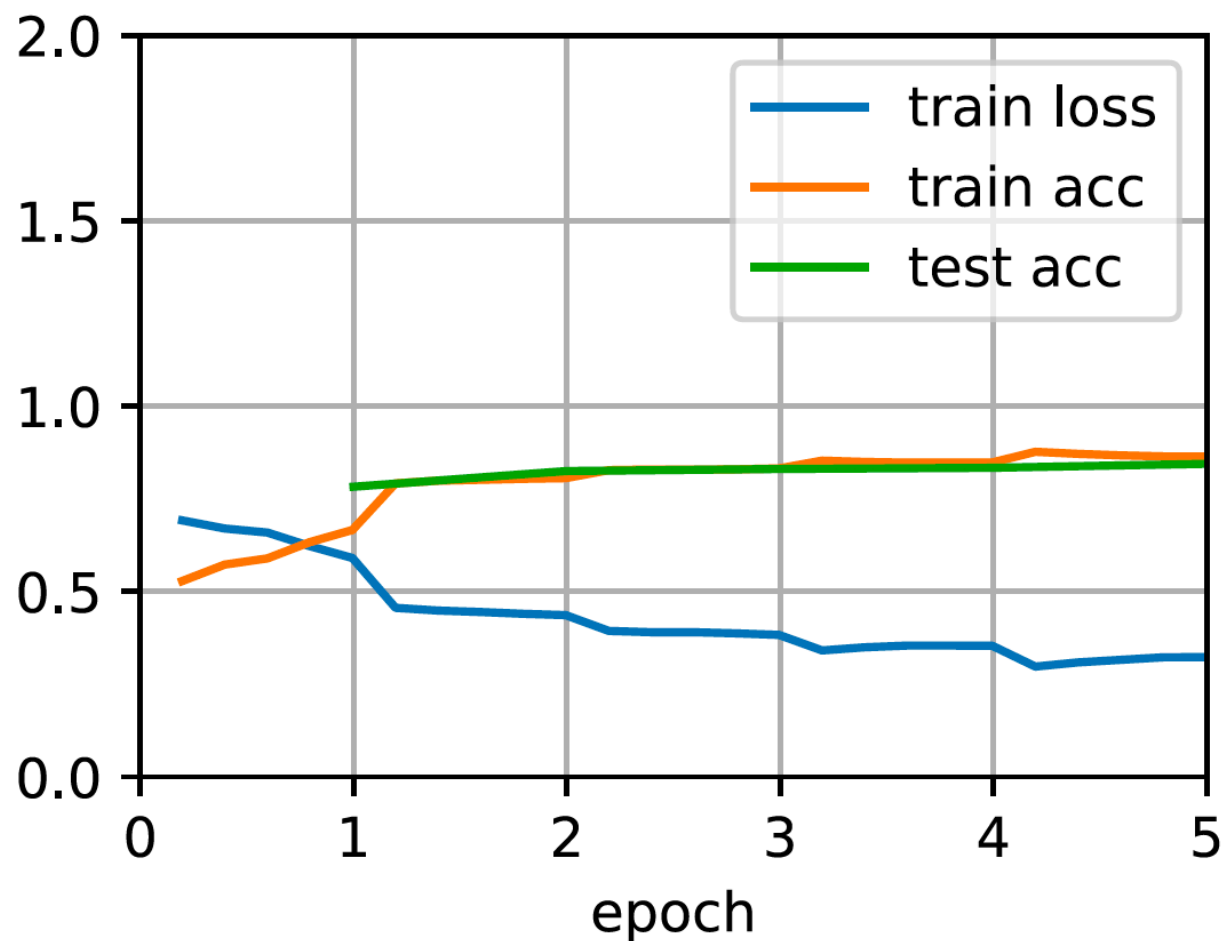
- Then, we will use these word vectors as feature vectors for each word in the reviews.

```
net.embedding.weight.set_data(embeds)  
net.embedding.collect_params().setattr('grad_req', 'null')
```

Training

```
lr, num_epochs = 0.01, 5
trainer = gluon.Trainer(net.collect_params(), 'adam', {'learning_rate': lr})
loss = gluon.loss.SoftmaxCrossEntropyLoss()
d2l.train_ch12(net, train_iter, test_iter, loss, trainer, num_epochs, ctx)
```

```
loss 0.323, train acc 0.864, test acc 0.845
887.7 exampes/sec on [gpu(0), gpu(1)]
```



Prediction

```
# Save to the d2l package.
def predict_sentiment(net, vocab, sentence):
    sentence = nd.array(vocab[sentence.split()], ctx=d2l.try_gpu())
    label = nd.argmax(net(sentence.reshape((1, -1))), axis=1)
    return 'positive' if label.asscalar() == 1 else 'negative'
```

Then, use the trained model to classify the sentiments of two simple sentences.

```
predict_sentiment(net, vocab, 'this movie is so great')
```

```
'positive'
```

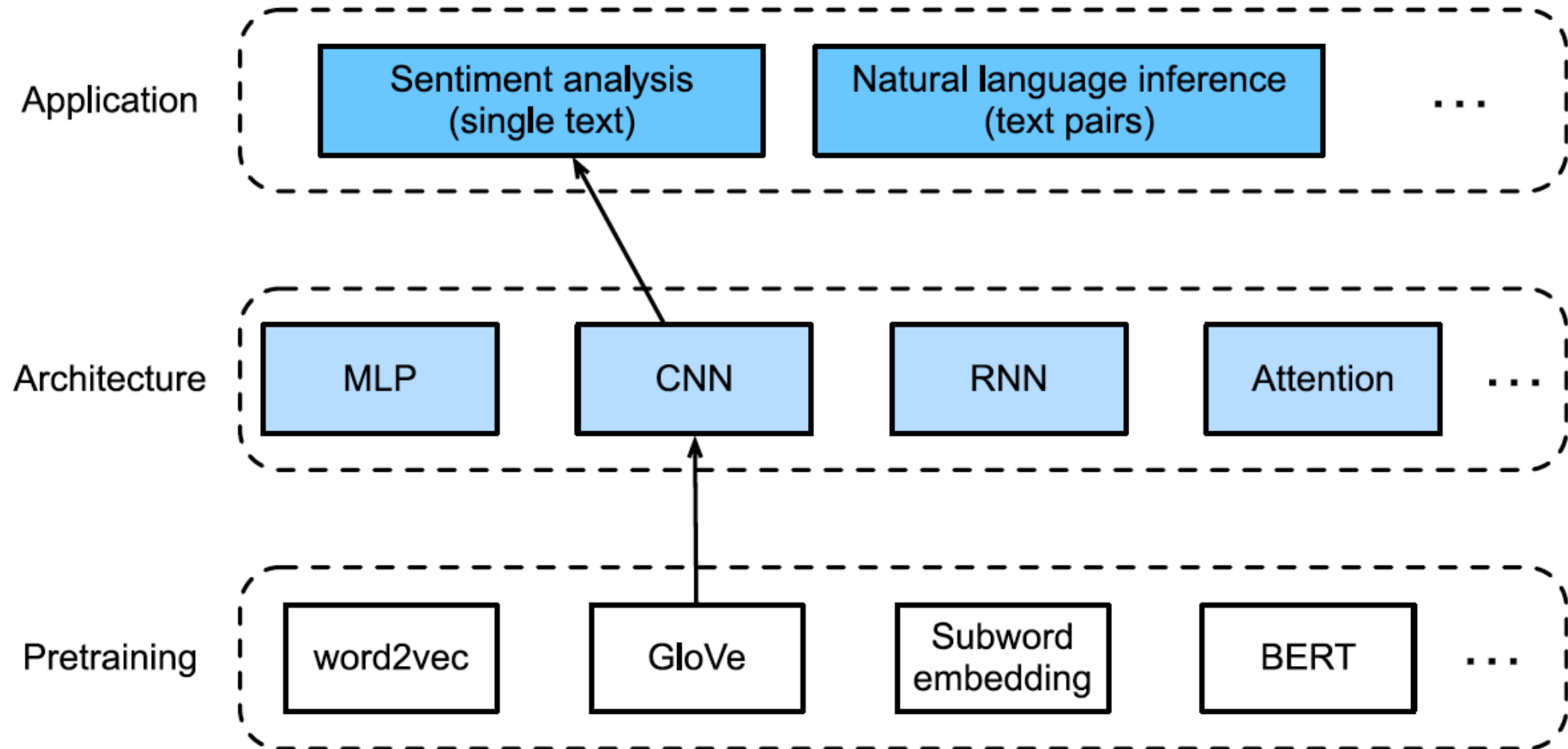
```
predict_sentiment(net, vocab, 'this movie is so bad')
```

```
'negative'
```

Text Sentiment Classification: Using Convolutional Neural Networks (textCNN)

- We explored how to process two-dimensional image data with two-dimensional convolutional neural networks.
- In the previous language models and text classification tasks, we treated text data as a time series with only one dimension, and naturally, we used recurrent neural networks to process such data.
- In fact, we can also treat text as a one-dimensional image, so that we can use one-dimensional convolutional neural networks to capture associations between adjacent words.
- Here we describe a groundbreaking approach to applying convolutional neural networks to text analysis: textCNN (Kim, 2014)

Text Sentiment Classification: Using Convolutional Neural Networks (textCNN)



One-dimensional Convolutional Layer

- Like a two-dimensional convolutional layer, a one-dimensional convolutional layer uses a one-dimensional cross-correlation operation.
- In the one-dimensional cross-correlation operation, the convolution window starts from the leftmost side of the input array and slides on the input array from left to right successively.

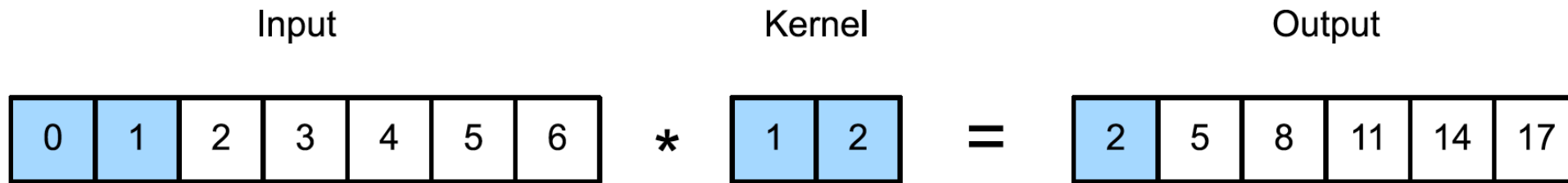
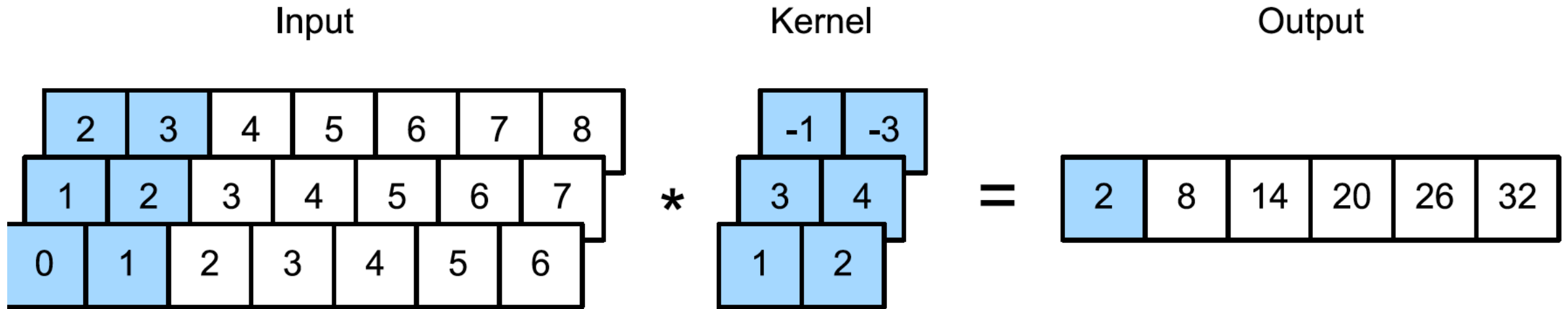


Fig. 15.10.1: One-dimensional cross-correlation operation. The shaded parts are the first output element as well as the input and kernel array elements used in its calculation: $0 \times 1 + 1 \times 2 = 2$.

One-dimensional Convolutional Layer

- The one-dimensional cross-correlation operation for multiple input channels is also similar to the two-dimensional cross-correlation operation for multiple input channels.
- On each channel, it performs the one-dimensional cross-correlation operation on the kernel and its corresponding input and adds the results of the channels to get the output.



One-dimensional Convolutional Layer

- The definition of a two-dimensional cross-correlation operation tells us that a one-dimensional cross-correlation operation with multiple input channels can be regarded as a two-dimensional cross-correlation operation with a single input channel.

Input Kernel Output

2	3	4	5	6	7	8
1	2	3	4	5	6	7
0	1	2	3	4	5	6

*

-1	-3
3	4
1	2

=

2	8	14	20	26	32
---	---	----	----	----	----

The diagram illustrates a 1D convolution operation. It shows an input vector of size 7, a kernel of size 2, and an output vector of size 6. The input vector is [2, 3, 4, 5, 6, 7, 8], the kernel is [-1, -3], and the output is [2, 8, 14, 20, 26, 32]. The first two elements of the input (2 and 3) and the entire kernel are highlighted in blue. The first element of the output (2) is also highlighted in blue. The operation is represented by a multiplication symbol (*) and an equals sign (=).

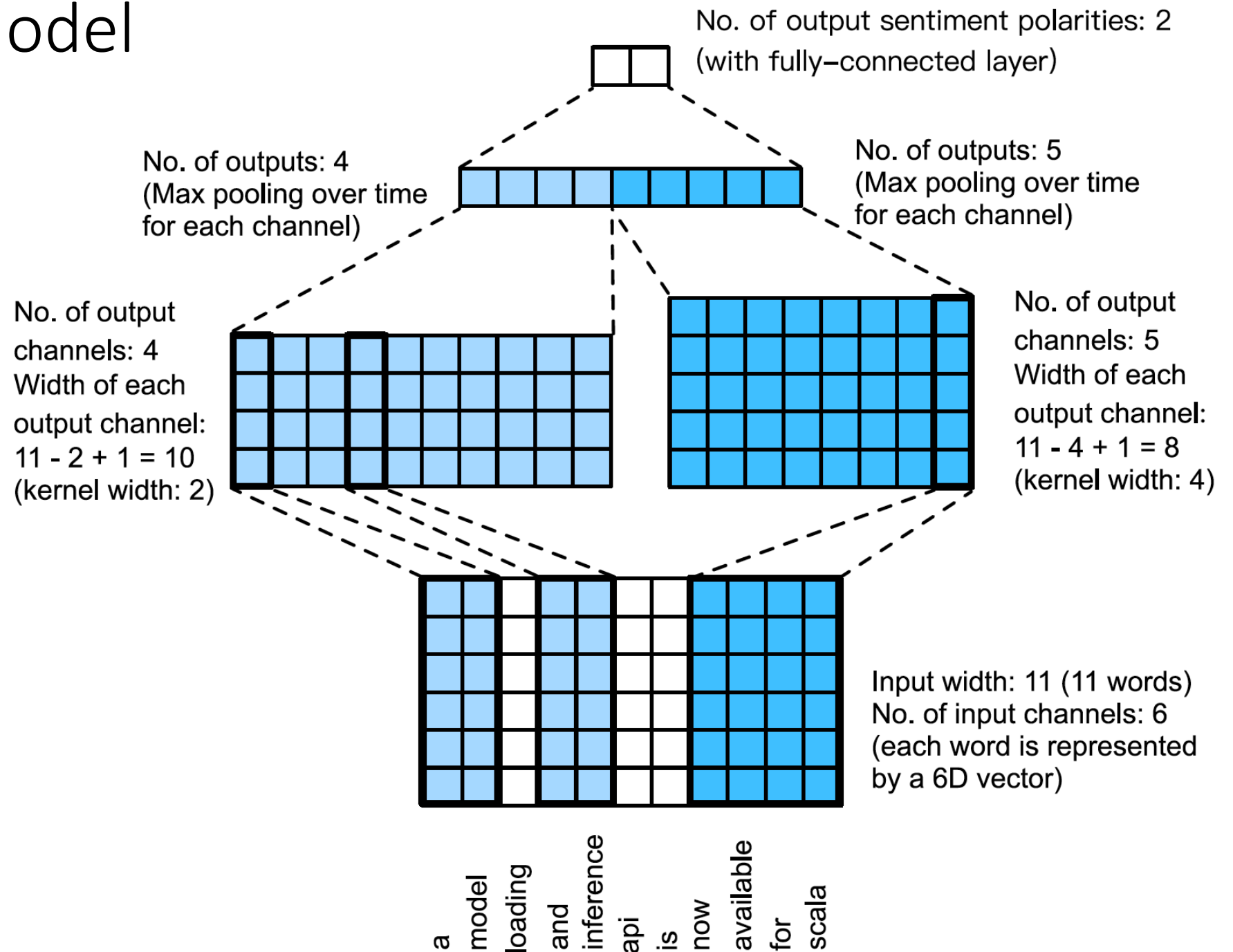
Max-Over-Time Pooling Layer

- Similarly, we have a one-dimensional pooling layer.
- The max-over-time pooling layer used in TextCNN actually corresponds to a one-dimensional global maximum pooling layer.
- Assuming that the input contains multiple channels, and each channel consists of values on different time steps, the output of each channel will be the largest value of all time steps in the channel

The TextCNN Model

- TextCNN mainly uses a one-dimensional convolutional layer and max-over-time pooling layer.
- Suppose the input text sequence consists of n words, and each word is represented by a d -dimension word vector.
- Then the input example has a width of n , a height of 1, and d input channels.
- The calculation of textCNN can be mainly divided into the following steps:
 1. Define multiple one-dimensional convolution kernels and use them to perform convolution calculations on the inputs.
 2. Perform max-over-time pooling on all output channels, and then concatenate the pooling output values of these channels in a vector.
 3. The concatenated vector is transformed into the output for each category through the fully connected layer. A dropout layer can be used in this step to deal with overfitting.

The TextCNN Model



Natural Language Inference

- Natural language determines the logical relationship between a pair of text sequences.
- Such relationships usually fall into three types:
 - *Entailment*: the hypothesis can be inferred from the premise.
 - *Contradiction*: the negation of the hypothesis can be inferred from the premise.
 - *Neutral*: all the other cases.

Premise: Two women are hugging each other.

Hypothesis: Two women are showing affection.

Premise: A man is running the coding example from Dive into Deep Learning.

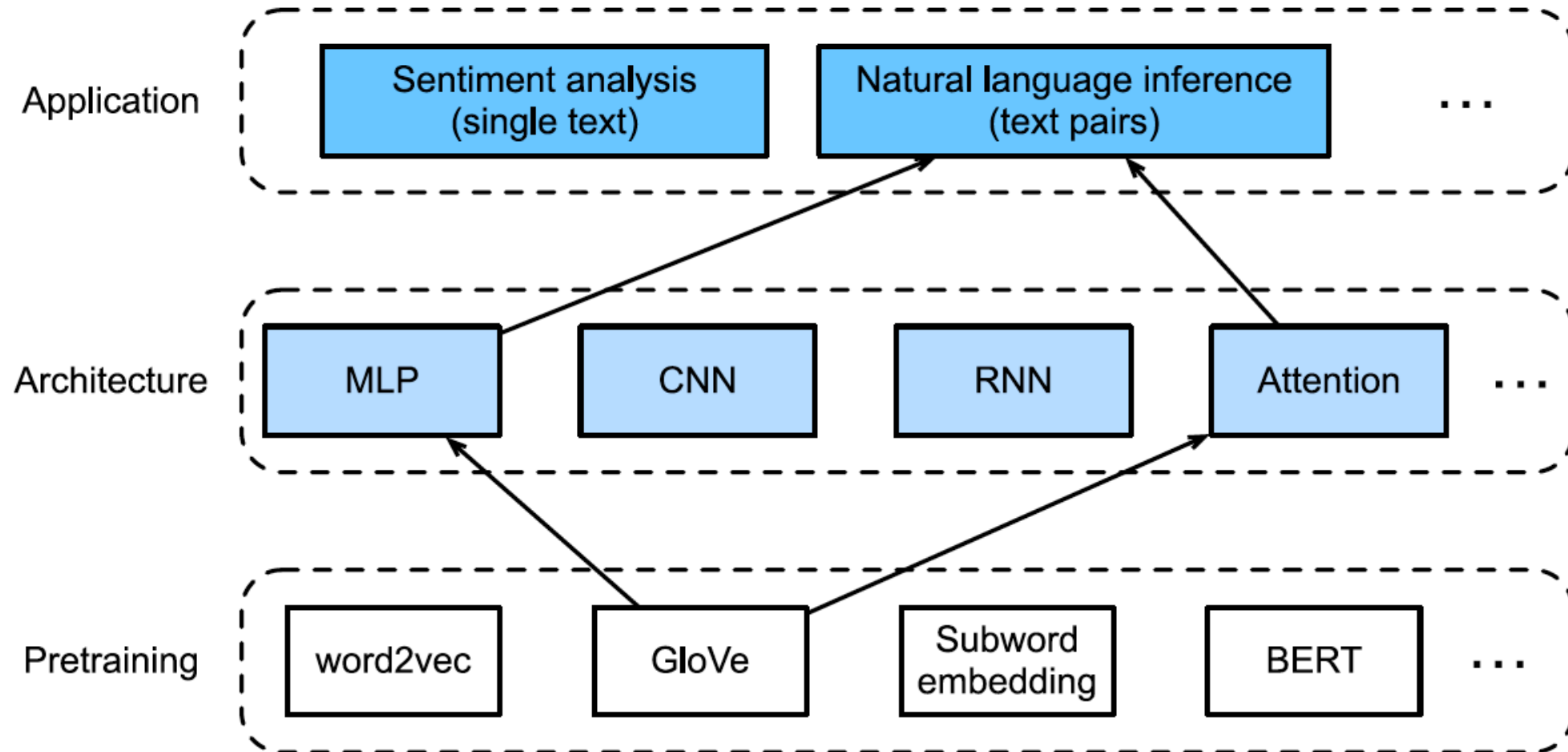
Hypothesis: The man is sleeping.

Premise: The musicians are performing for us.

Hypothesis: The musicians are famous.

Natural Language Inference: Using Attention

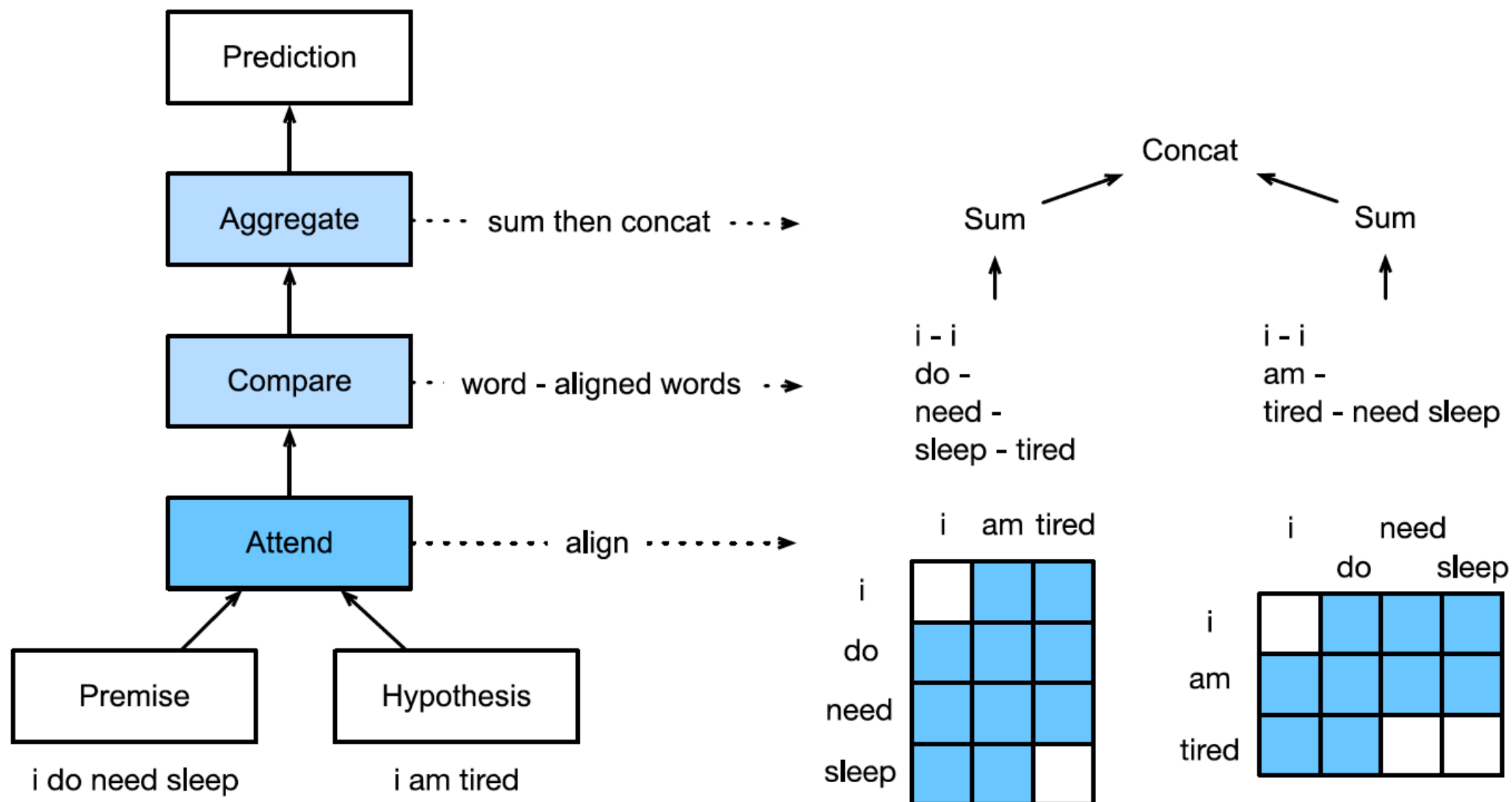
- Parikh et al. 2016, proposed "decomposable attention model" without recurrent or convolutional layers.



Natural Language Inference: Using Attention

- Instead of preserving the order of words in premises and hypotheses, we can just align words in one text sequence to every word in the other, and vice versa, then compare and aggregate such information.
- The alignment of words can be neatly accomplished by attention mechanisms.

Natural Language Inference: Using Attention



Natural Language Inference: Using Attention - attending

- Assume \mathbf{a}_i is d-dimensional embedding for word i in premise
- And \mathbf{b}_i is d-dimensional embedding for word i in hypothesis
- For soft alignment, we compute attention weights as

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

Where f is a MLP.


- We compute the weighted average of all the word embeddings

$$\beta_i = \sum_{j=1}^n \frac{\exp(e_{ij})}{\sum_{k=1}^n \exp(e_{ik})} \mathbf{b}_j, \quad \alpha_j = \sum_{i=1}^m \frac{\exp(e_{ij})}{\sum_{k=1}^m \exp(e_{kj})} \mathbf{a}_i.$$

Natural Language Inference: Using Attention - comparing

- Next, we compare a word in one sequence with the other sequence that is softly aligned with that word.
- In the comparing step, we feed the concatenation of words from one sequence and aligned words from other sequence into a function g .

$$\mathbf{v}_{A,i} = g([\mathbf{a}_i, \beta_i]), i = 1, \dots, m$$
$$\mathbf{v}_{B,j} = g([\mathbf{b}_j, \alpha_j]), j = 1, \dots, n.$$



The comparison between word i in the premise and all the hypothesis words that are softly aligned with word i .

Natural Language Inference: Using Attention - aggregating

- In the last step, we will aggregate comparison vectors to infer logical relationship.
- We begin by summing both sets:

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

- Next, we feed the concatenation of both summarization results into function h to obtain the classification result of the logical relationship:

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

Natural Language Inference: Using Attention – Fine-Tuning BERT

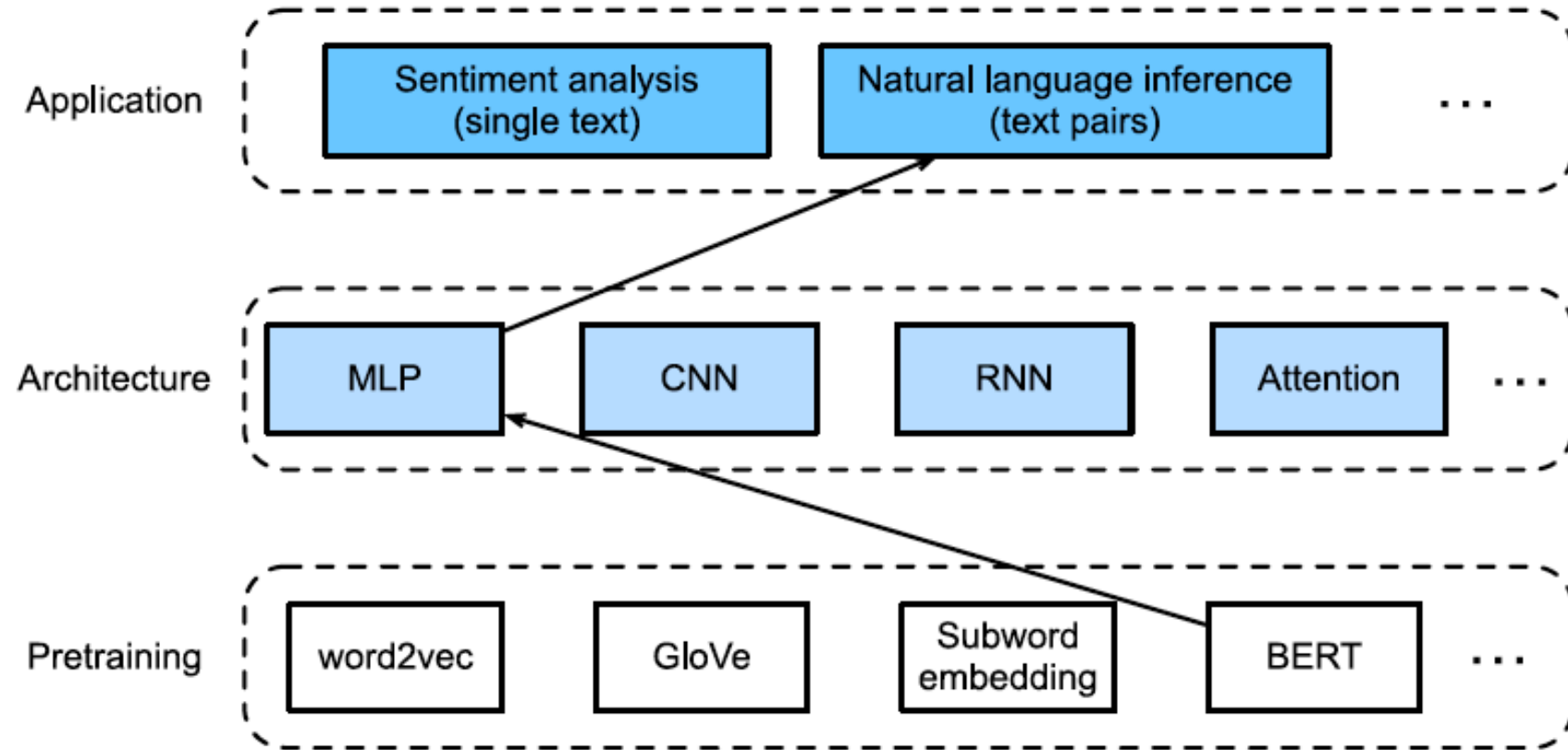


Fig. 15.7.1: This section feeds pretrained BERT to an MLP-based architecture for natural language inference.

Natural Language Inference: Using Attention – Fine-Tuning BERT

- Fine-tuning BERT for language inference requires only an extra MLP layer.

```
class BERTClassifier(nn.Block):  
    def __init__(self, bert):  
        super(BERTClassifier, self).__init__()  
        self.bert = bert  
        self.classifier = nn.Sequential()  
        self.classifier.add(nn.Dense(256, activation='relu'))  
        self.classifier.add(nn.Dense(3))  
  
    def forward(self, inputs):  
        tokens_X, segments_X, valid_lens_x = inputs  
        encoded_X, _, _ = self.bert(tokens_X, segments_X, valid_lens_x)  
        return self.classifier(encoded_X[:, 0, :])
```