Lecture 4: Sequence Labeling

Tianpei Xie

Jul. 1st., 2022

Contents

$Q = q_1 q_2 \dots q_N$	a set of N **states**
$A = a_{11} \dots a_{ij} \dots a_{NN}$	a **transition probability matrix** A, each a_{ij} representing the probability of moving from state i to state j, s.t. $\sum_{j=1}^{N} a_{ij} = 1 \quad \forall i$
$O = o_1 o_2 \dots o_T$	a sequence of T **observations**, each one drawn from a vocabulary $V = v_1, v_2, ..., v_V$
$B = b_i(o_t)$	a sequence of **observation likelihoods**, also called **emission probabilities**, each expressing the probability of an observation o_t being generated from a state q_i
$\pi = \pi_1, \pi_2, ..., \pi_N$	an **initial probability distribution** over states. π_i is the probability that the Markov chain will start in state i. Some states j may have $\pi_j = 0$, meaning that they cannot be initial states. Also, $\sum_{i=1}^{n} \pi_i = 1$

Figure 1: Hidden Markov Model notations.

1 Hidden Markov Model

A **hidden Markov model** (**HMM**) allows us to talk about both *observed* events (like words that we see in the input) and *hidden* events (like part-of-speech tags) that we think of as causal factors in our probabilistic model. An HMM is specified by the following components [Jurafsky and Martin, 2014]

- **state** at time t, $q_t \in \{1, \dots, N\}$: is not observed. In POS tagging, it is the POS tag.
- **observation** at time t, o_t: is controlled by the state
- **transition probability**, $a_{i,j} = P(q_t = j | q_{t-1} = i)$, transit from state i to state j. $\boldsymbol{A} = [a_{i,j}]_{N \times N}$
- **emission probability**, $b_i(o_t) = P(o_t = i | q_t)$, $\boldsymbol{B} = [b_i(o_t)]$
- **initial probability**, $\pi_i = P(q_0 = i)$

HMM has two assumptions on the distribution

- **Markov assumption**: $P(q_i | q_1, \dots, q_{i-1}) = P(q_i | q_{i-1})$, i.e. given the current state, the future state and the past state are independent.
- **Output Independence**: $P(o_i | q_1, \dots, q_{i-1}, q_i, o_1, \dots, o_{i-1}) = P(o_i | q_i)$, i.e. the current observation is only dependent on the current state.

1.1 HMM for POS tagging

Part-of-speech tagging can be treated as a decoding problem using HMM, i.e. given HMM $(\boldsymbol{A}, \boldsymbol{B})$ and a sequence of observations (words) $(o_1, \dots, o_T)$, find the most probable (maximum a-priori) sequence of states (tags) $(q_1, \dots, q_T)$

$$\begin{aligned}
\hat{q}_{1:T} &= \text{argmax}_{q_{1:T}} P(q_{1:T} | o_{1:T}) \\
&= \text{argmax}_{q_{1:T}} P(o_{1:T} | q_{1:n}) P(q_{1:T}) \\
&= \text{argmax}_{q_{1:T}} \prod_{i=1}^{T} P(o_i | q_i) P(q_i | q_{i-1})
\end{aligned}$$

function VITERBI(*observations* of len T,*state-graph* of len N) **returns** *best-path*, *path-prob*

create a path probability matrix *viterbi[N,T]*
for each state s **from** 1 **to** N **do** ; initialization step
  $viterbi[s,1] \leftarrow \pi_s * b_s(o_1)$
  $backpointer[s,1] \leftarrow 0$
for each time step t **from** 2 **to** T **do** ; recursion step
  for each state s **from** 1 **to** N **do**
    $viterbi[s,t] \leftarrow \max_{s'=1}^{N} viterbi[s',t-1] * a_{s',s} * b_s(o_t)$
    $backpointer[s,t] \leftarrow \operatorname{argmax}_{s'=1}^{N} viterbi[s',t-1] * a_{s',s} * b_s(o_t)$
$bestpathprob \leftarrow \max_{s=1}^{N} viterbi[s,T]$; termination step
$bestpathpointer \leftarrow \operatorname{argmax}_{s=1}^{N} viterbi[s,T]$; termination step
$bestpath \leftarrow$ the path starting at state *bestpathpointer*, that follows backpointer[] to states back in time
return *bestpath*, *bestpathprob*

Figure 8.10 Viterbi algorithm for finding the optimal sequence of tags. Given an observation sequence and an HMM $\lambda = (A,B)$, the algorithm returns the state path through the HMM that assigns maximum likelihood to the observation sequence.

Figure 2: Viterbi algorithm for decoding HMM.

Solving the optimization problem above can be done via **Viterbi algorithm**, an instance of **dynamic programming**. Figure 2 describes the Viterbi algorithm. Denote $v_t(i)$ being the optimal HMM probability in state i at time t together with stats and observations before t,

$$v_t(i) = \max_{q_{1:t-1}} P(q_{1:t-1}, o_{1:t-1}, q_t = i|\lambda)$$

The dynamic programming utilize the recursion formula (Bellman equation):

$$v_t(i) = \max_{j \in \{1,\dots,N\}} v_{t-1}(j) \;\; a_{j,i} \; b_i(o_t) \tag{1}$$

This choose the most probable **path probability** based on optimal path from previous stage $t-1$.

We can compare (1) to a linear recursive network. In linear recursive network, the states are updated based on $\boldsymbol{v}_t = (\boldsymbol{A}\boldsymbol{v}_{t-1}) \odot \boldsymbol{B}$. In this equation, we marginalized over all **path probabilities** (i.e. **path integral**) to obtain the *marginalized distribution* instead of *posterior* distribution from Viterbi algorithm. In (1), we use max operation $(\max_j v_{t-1}(j)a_{j,i})b_i$ for each path instead of summation $(\sum_j v_{t-1}(j)a_{j,i})b_i$ over all paths.

2 Conditional Random Fields (CRFs)

HMM is a generative model. It is hard for HMM to add arbitrary features directly into the model. A discriminative model can be better. The **conditional random field (CRF)** can be used to solve this issue. The most commonly used CRF is linear chain CRF, which is close to HMM. Similar to HMM, CRF optimize the posterior probability

$$\hat{q}_{1:T} = \operatorname{argmax}_{q_{1:T}} P(q_{1:T}|o_{1:T}).$$

Unlike HMM, CRF define the conditional probability directly as a **log-linear** model:

$$P(q_{1:T}|o_{1:T}) = \frac{1}{Z(o_{1:T})} \exp\left(\sum_{k=1}^{K} w_k F_k(q, o)\right)$$
$$= \frac{1}{Z(o_{1:T})} \exp\left(\sum_{k=1}^{K}\sum_{t}^{T} w_k f_k(q_t, q_{t-1}, o_{1:T}, t)\right)$$

Each f_k is called a **local feature**, which is accumulated over time to form a **global feature** F_k. Each of local feature is a linear-chain CRF which makes use of the current state/tag q_t and previous state/tag q_{t-1} and the entire observations $o_{1:T}$ and the current position t. It is linear chain CRG since f_k only depends on q_t and q_{t-1}. An example pos feature $f_k := \mathbb{1}\{o_t = \text{"the"}, q_t = \text{DET}\}$ and $f_s := \mathbb{1}\{o_{t+1} = \text{"Street"}, q_t = \text{PROPN}, q_{t-1} = \text{NUM}\}$. For NER feature, could be "identity of o_i, identity of neighboring words".

The linear structure of CRF makes it easy to build **binary feature** and extend to different special cases. These feature can be manually designed or automatically generated via **feature templates**: $< q_t, o_t >, < q_t, q_{t-1} >, < q_t, o_{t-1}, o_{t+1} >$. These features can be generated from training data. For unknown words, we can define the **word shape** features, which represent the ***abstract letter pattern*** of the word by mapping lower-case letters to x, upper-case to X, numbers to d, and retaining punctuation, e.g. $I.M.F := X.X.X$. Prefix and suffix features are also useful. For example some feature templates with unknown words such as "o_i contains a particular prefix/suffix" or "o_i's word shape" etc.

The *known-word templates* are computed for every word seen in the training set; the *unknown word features* can also be computed for all words in training, or only on training words whose frequency is below some threshold. The result of the known-word templates and word-signature features is *a very large set of features.*

In NER, one feature that is especially useful for **locations** is a **gazetteer**, a list of place names, often providing millions of entries for locations with detailed geographical and political information. This can be implemented as a **binary feature** indicating a phrase appears in the list. Other related resources like name-lists can be used, as can other entity dictionaries like lists of corporations or products, although they may not be as helpful as a gazetteer

2.1 Inference on CRF

$$\hat{q}_{1:T} = \text{argmax}_{q_{1:T}} P(q_{1:T}|o_{1:T})$$
$$= \text{argmax}_{q_{1:T}} \frac{1}{Z(o_{1:T})} \exp\left(\sum_{k=1}^{K} w_k F_k(q_{1:T}, o_{1:T})\right)$$
$$= \text{argmax}_{q_{1:T}} \exp\left(\sum_{k=1}^{K} w_k F_k(q_{1:T}, o_{1:T})\right)$$
$$= \text{argmax}_{q_{1:T}} \sum_{k=1}^{K} w_k F_k(q_{1:T}, o_{1:T})$$

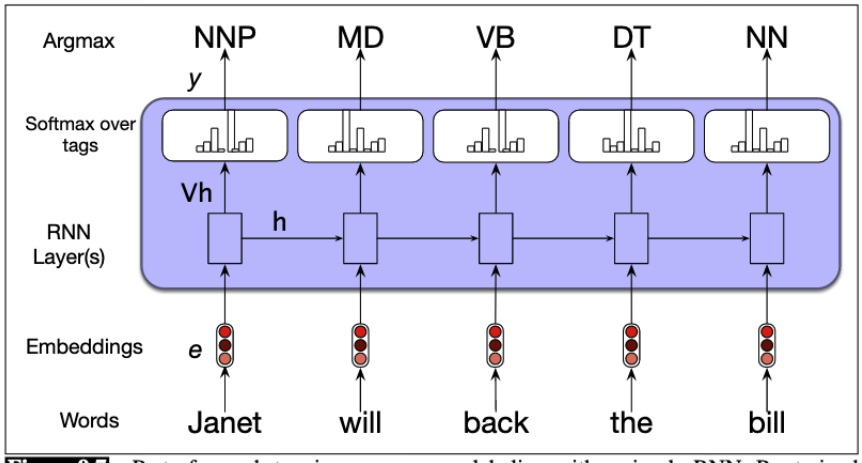

Figure 9.7 Part-of-speech tagging as sequence labeling with a simple RNN. Pre-trained word embeddings serve as inputs and a softmax layer provides a probability distribution over the part-of-speech tags as output at each time step.

Figure 3: The inference process for RNN on sequence labeling task.

$$= \text{argmax}_{q_{1:T}} \sum_{t}^{T} \sum_{k=1}^{K} w_k f_k(q_t, q_{t-1}, o_{1:T}, t)$$

Similar to HMM, we can solve the problem using Viterbi algorithm

$$v_t(i) = \max_{j=1,\ldots,N} v_{t-1}(j) \sum_{k=1}^{K} w_k f_k(q_t, q_{t-1}, o_{1:T}, t) \tag{2}$$

Compared to (1), we can see that the matrix multiplication $\boldsymbol{AB}$ is replaced by linear combinations of binary features.

Learning CRF i.e. the weights for global features $\{w_k\}$, is a standard supervised learning problem. Given a sequence of observations, feature functions, and corresponding outputs, we use stochastic gradient descent to train the weights to maximize the log-likelihood of the training corpus.

3 Recurrent Neural Networks

We can use simple feed-forward neural network to model the sequence labeling: use a fixed window of size d, each time, providing all context words in within the window (centered around a word w_i) to the feed-forward neural network to predict the label of the word and then slides the window to the right. This method does not handle the sequential nature of the natural languages, thus it fails to capture and exploit the temporal relationships.

The **recurrent network (RNN)** offers a new way to represent the prior context, allowing the models decision to depend on information from hundreds of words in the past. The transformer offers new mechanisms (self-attention and positional encodings) that help represent time and help focus on how words relate to each other over long distances.

The inference process of RNN for sequence labeling task is shown below:

$$\boldsymbol{B}_t = \boldsymbol{E}\boldsymbol{o}_t$$

$$\boldsymbol{v}_t = \sigma\left(\boldsymbol{A}\boldsymbol{v}_{t-1} + \boldsymbol{W}\boldsymbol{B}_t\right)$$
$$\boldsymbol{g}_t = \text{softmax}\left(\boldsymbol{U}\boldsymbol{v}_t\right)$$

Here, $\boldsymbol{g}_t$ is the distribution over tags, and $\boldsymbol{o}_t$ is a word embedding at position t. The state vector $\boldsymbol{v}_t$. Figure 3 illustrate the inference process of RNN on sequence labeling task.

3.1 LSTM

In practice, it is quite difficult to train RNNs for tasks that require a network to make use of information distant from the current point of processing. Yet distant information is critical to many language applications. One reason for the inability of RNNs to carry forward critical information is that the hidden layers, and, by extension, the weights that determine the values in the hidden layer, are being asked to perform two tasks simultaneously: provide information useful for the current decision, and updating and carrying forward information required for future decisions.

The most commonly used extension to RNNs is the **Long short-term memory (LSTM)** network. LSTMs divide the context management problem into two sub-problems: *removing information no longer needed from the context*, and *adding information likely to be needed for later decision making*. The key to solving both problems is to learn how to manage this context rather than hard-coding a strategy into the architecture.

LSTMs accomplish this by first adding an explicit *context layer* to the architecture (in addition to the usual *recurrent hidden layer*), and through the use of specialized neural units that make use of **gates** to control the flow of information into and out of the units that comprise the network layers. These gates are implemented through the use of additional weights that operate sequentially on the input, and *previous hidden layer*, and *previous context layers*.

LSTM structure can be divided into several parts (see Figure 4):

- **forget gate**: The purpose of this gate to **delete information from the context** that is no longer needed.

  $$\boldsymbol{f}_t = \sigma\left(\boldsymbol{U}_f\boldsymbol{h}_{t-1} + \boldsymbol{W}_f\boldsymbol{x}_t\right)$$

  where $\boldsymbol{x}_t$ is the input observation, $\boldsymbol{h}_{t-1}$ is the recurrent hidden layer (short memory) at $t-1$, and $\boldsymbol{c}_t$ is the context layer (long memory). $\sigma(\cdot)$ is the sigmoid function output $[0, 1]$.

- **input gate**: The purpose of this gate is to select the information to **add to the current context**.

  $$\boldsymbol{i}_t = \sigma\left(\boldsymbol{U}_i\boldsymbol{h}_{t-1} + \boldsymbol{W}_h\boldsymbol{x}_t\right)$$

- **candidate for context update**:

  $$\boldsymbol{g}_t = \tanh\left(\boldsymbol{U}_g\boldsymbol{h}_{t-1} + \boldsymbol{W}_g\boldsymbol{x}_t\right)$$

- **context update**: The context (long-term memory) is determined by combination of its previous context masked by forget gate and the current hidden state candidate masked by input gate

  $$\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \boldsymbol{g}_t$$

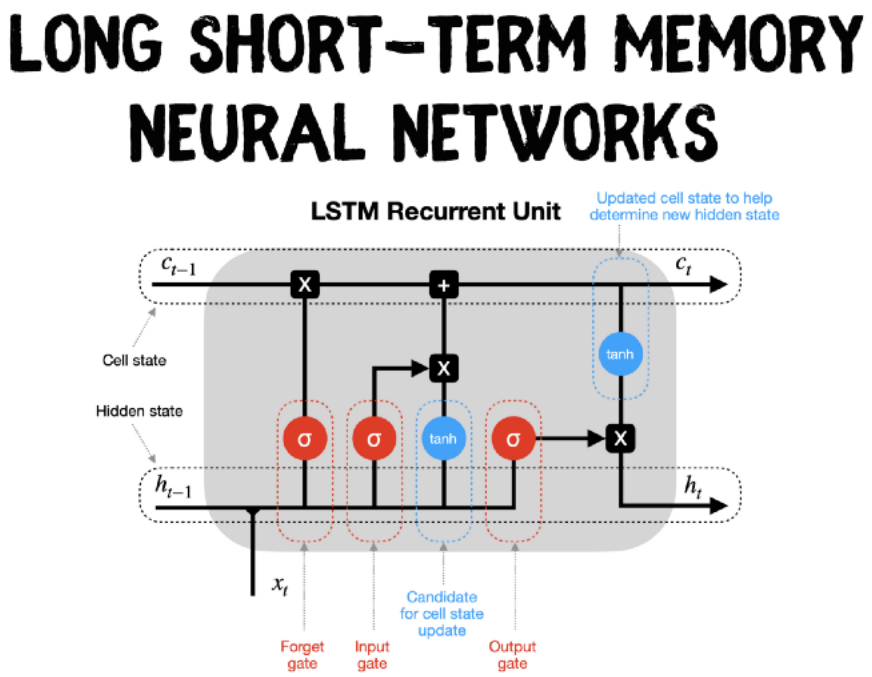

Figure 4: The Long Short Term Memory.

- **output gate**: This gate is used to decide what information is required for the **current hidden state** (as opposed to what information needs to be *preserved for future decisions*).

$$\boldsymbol{o}_t = \sigma\left(\boldsymbol{U}_o \boldsymbol{h}_{t-1} + \boldsymbol{W}_o \boldsymbol{x}_t\right)$$

- **hidden (recurrent) state update**: the updated hidden (recurrent) state is determined by output gate and context vector (long-term memory)

$$\boldsymbol{h}_t = \boldsymbol{o}_t \odot \tanh\left(\boldsymbol{c}_t\right)$$

As seen in Figure 4, the hidden recurrent state (short-term memory) at previous stage $\boldsymbol{h}_{t-1}$ and input $\boldsymbol{x}_t$ at current stage jointly determined the *forget gate*, the *input gate*, the *output gate* and the *candidate context*. Thus $\boldsymbol{h}_{t-1}$, $\boldsymbol{x}_t$ and the previous stage $\boldsymbol{c}_{t-1}$ jointly determine the long-term memory $\boldsymbol{c}_t$ at current stage. The long-term memory $\boldsymbol{c}_t$ and the output gate (thus $\boldsymbol{h}_{t-1}$ and $\boldsymbol{x}_t$) determine the short-term memory $\boldsymbol{h}_t$ at current stage.

4 Transformer

LSTM still not fully resolve the challenges of long dependencies and vanishing gradient problem. Passing information through an extended series of recurrent connections leads to information loss and difficulties in training. Moreover, the inherently sequential nature of recurrent networks makes it hard to do computation in *parallel*. In [Vaswani et al., 2017], it introduces an encoder-decoder

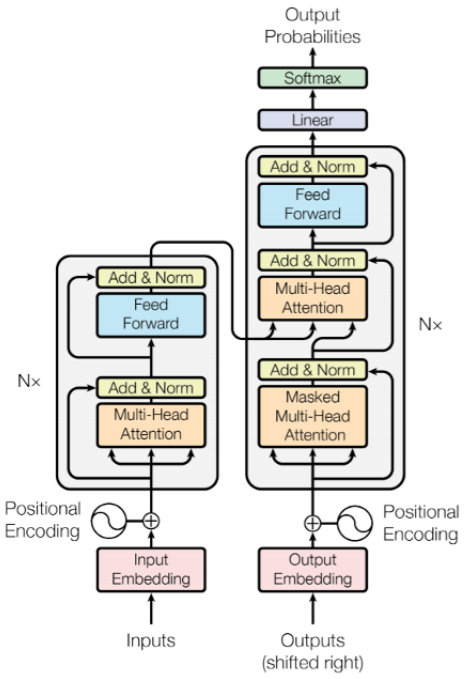

Figure 1: The Transformer - model architecture.

Figure 5: The encoder-decoder structure of transformer.

architecture based on *attention layers*, termed as the **transformer**. One main difference is that the input sequence can be passed parallelly so that GPU can be utilized effectively, and the speed of training can also be increased. And it is based on the multi-headed attention layer, vanishing gradient issue is also overcome by a large margin.

Transformers map sequences of input vectors $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ to sequences of output vectors $(y_1, \ldots y_n)$ of the same length. Transformers are made up of stacks of transformer blocks, which are multi-layer networks made by combining simple linear layers, feedforward networks, and **self-attention layers**, the key innovation of transformers. Self-attention allows a network to directly extract and use information from arbitrarily large contexts without the need to pass it through intermediate recurrent connections as in RNNs.

At the core of an attention-based approach is the ability to ***compare*** an item of interest to a collection of other items in a way that reveals their relevance in the current context. In the case of self-attention, the set of comparisons are to other elements within a given sequence. The result of these comparisons is then used to compute an output for the current input. The attention score is computed using softmax of inner product between two inputs:

$$\begin{aligned} \alpha_{i,j} &= \text{softmax}(\boldsymbol{x}_i^T \boldsymbol{x}_j) \\ &= \frac{\exp(\boldsymbol{x}_i^T \boldsymbol{x}_j)}{\sum_k \exp(\boldsymbol{x}_i^T \boldsymbol{x}_k)} \end{aligned}$$

$\alpha_{i,j}$ indicates the proportional relevance of each input to the input element i that is the current focus of attention. And the output can be seen as linear combination of inputs weighted by attention

score

$$\boldsymbol{y}_i = \sum_j \alpha_{i,j} \boldsymbol{x}_j.$$

The result of a dot product can be an arbitrarily large (positive or negative) value. Exponentiating such large values can lead to numerical issues and to an effective loss of gradients during training. To avoid this, the dot product needs to be *scaled* in a suitable fashion. A scaled dot-product approach divides the result of the dot product by a factor related to the size of the embeddings before passing them through the softmax. A typical approach is to divide the dot product by the square root of the dimensionality of the query and key vectors d_k.

$$\alpha_{i,j} = \text{softmax}\left(\frac{\boldsymbol{x}_i^T \boldsymbol{x}_j}{\sqrt{d_k}}\right)$$

The core of an attention-based approach includes:

- a set of comparisons to relevant items in some context,
- a normalization of those scores to provide a probability distribution
- a weighted sum using this distribution. The output $\boldsymbol{y}$ is the result of this straightforward computation over the inputs.

Transformers allow us to create a more sophisticated way of representing how words can contribute to the representation of longer inputs. Consider the three different ***roles*** that each input embedding plays during the course of the attention process.

- As the *current focus* of attention when being compared to all of the other preceding inputs. This role is refer to as a **query**.
- In its role as a preceding input *being compared to* the current focus of attention. This role is refer to as a **key**.
- And finally, as a **value** used to compute the output for the current focus of attention.

To capture these three different roles, transformers introduce weight matrices $\boldsymbol{W}_Q \in \mathbb{R}^{d_k \times d}$, $\boldsymbol{W}_K \in \mathbb{R}^{d_k \times d}$, and $\boldsymbol{W}_V \in \mathbb{R}^{d_v \times d}$. These weights will be used to project each input vector $\boldsymbol{x}_i \in \mathbb{R}^d$ into a representation of its role as a key, query, or value.

$$\boldsymbol{q}_i = \boldsymbol{W}_Q \boldsymbol{x}_i, \quad \boldsymbol{k}_i = \boldsymbol{W}_K \boldsymbol{x}_i, \quad \boldsymbol{v}_i = \boldsymbol{W}_V \boldsymbol{x}_i$$

The attention score is computed between key and query

$$\alpha_{i,j} = \text{softmax}\left(\frac{\boldsymbol{q}_i^T \boldsymbol{k}_j}{\sqrt{d_k}}\right) \tag{3}$$

Each output $\boldsymbol{y}_i$ can be computed in parallel by taking advantage of efficient matrix multiplication routines. Let $\boldsymbol{X} \in \mathbb{R}^{N \times d}$ be the input matrix with each row being the embedding of word i. Then we can compute the query $\boldsymbol{Q} \in \mathbb{R}^{N \times d_k}$, key $\boldsymbol{K} \in \mathbb{R}^{N \times d_k}$ and value matrix $\boldsymbol{V} \in \mathbb{R}^{N \times d_v}$ via

$$\boldsymbol{Q} = \boldsymbol{X}\boldsymbol{W}_Q^T, \quad \boldsymbol{K} = \boldsymbol{X}\boldsymbol{W}_K^T, \quad \boldsymbol{V} = \boldsymbol{X}\boldsymbol{W}_V^T$$

Given these matrices we can compute all the requisite query-key comparisons **simultaneously** by multiplying $\boldsymbol{Q}$ and $\boldsymbol{K}^T$ in a single matrix multiplication. Taking this one step further, we can scale

these scores, take the softmax, and then multiply the result by $\boldsymbol{V}$ resulting in a matrix of shape $N \times d_v$: a *vector embedding representation* for each token in the input.

$$\text{self-attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{d_k}}\right)\boldsymbol{V} \qquad \in \mathbb{R}^{N \times d_v} \tag{4}$$

The $N \times N$ self-attention comparison matrix can be **masked**, e.g. to cover all entries in upper triangle with $-\infty$ to avoid knowing words ahead of current word. Computation of this $N \times N$ takes quardratic time which is very expensive for long input sentences.

4.1 Transformer blocks

The self-attention calculation lies at the core of whats called a *transformer block*, which, in addition to the self-attention layer, includes additional feedforward layers, residual connections, and normalizing layers. The input and output dimensions of these blocks are matched so they can be stacked just as was the case for stacked RNNs.

A transformer block consists of following components:

- a self-attention layer with residual connections followed by layer normalization:

$$\boldsymbol{z} = \text{LayerNorm}\left(\boldsymbol{x} + \text{SelfAttn}(\boldsymbol{x})\right)$$

- a feed-forward layer with residual connections followed by layer normalization:

$$\boldsymbol{y} = \text{LayerNorm}\left(\boldsymbol{z} + \text{FFNN}(\boldsymbol{z})\right)$$

In deep networks, **residual connections** [He et al., 2016] are connections that pass information from a lower layer to a higher layer without going through the intermediate layer. Allowing information from the activation going forward and the gradient going backwards to skip a layer improves learning and gives higher level layers direct access to information from lower layers.

Layer normalization (or **layer norm**) is one of many forms of normalization that can be used to improve training performance in deep neural networks by keeping the values of a hidden layer in a range that facilitates gradient-based training. Layer norm is a variation of the standard score, or **z-score**, from statistics applied to a single hidden layer.

$$\begin{aligned}
\hat{x}_i &= \frac{(x_i - \mu)}{\sigma} \\
\text{where } \mu &= \frac{1}{d_h}\sum_i x_i \\
\sigma &= \sqrt{\frac{1}{d_h}\sum_i (x_i - \mu)^2}. \\
\text{LayerNorm}(\boldsymbol{x}) &= \gamma\hat{\boldsymbol{x}} + \beta,
\end{aligned}$$

where γ and β are two learnable parameters.

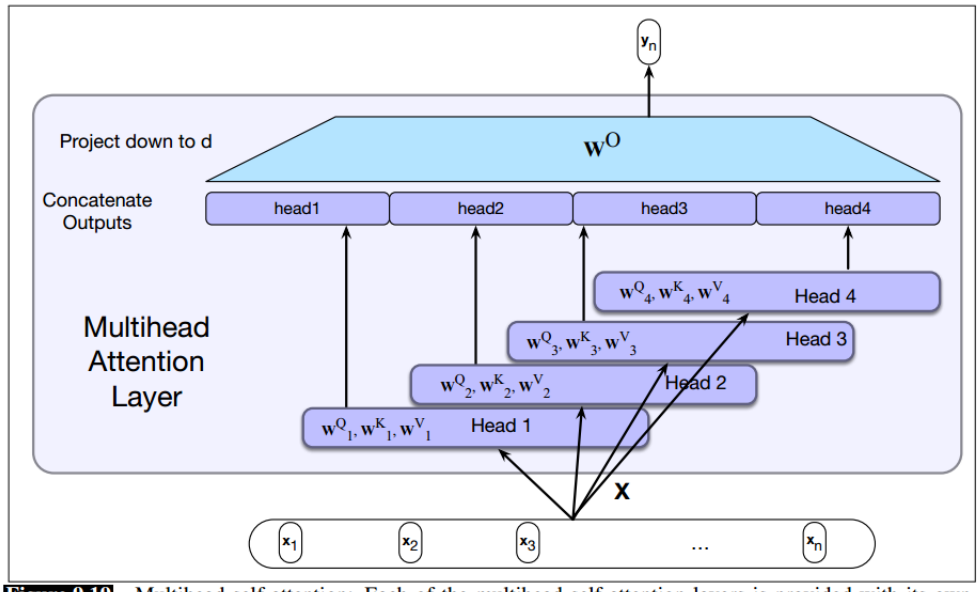

Figure 9.19 Multihead self-attention: Each of the multihead self-attention layers is provided with its own set of key, query and value weight matrices. The outputs from each of the layers are concatenated and then projected down to d, thus producing an output of the same size as the input so layers can be stacked.

Figure 6: The multi-head attention.

4.2 Multi-head attention

The different words in a sentence can relate to each other in many different ways *simultaneously*. For example, distinct syntactic, semantic, and discourse relationships can hold between verbs and their arguments in a sentence. It would be difficult for a single transformer block to learn to capture all of the different kinds of parallel relations among its inputs. Transformers address this issue with **multihead self-attention layers**. These are sets of self-attention layers, called **heads**, that reside in parallel layers at the same depth in a model, each with its own set of parameters. Given these distinct sets of parameters, each head can learn *different aspects of the relationships* that exist among inputs at the same level of abstraction. Figure 6 illustrate the approach.

To implement this notion, each head, i, in a self-attention layer is provided with its own set of key, query and value matrices $\boldsymbol{W}_Q^i \in \mathbb{R}^{d_k \times d}$, $\boldsymbol{W}_K^i \in \mathbb{R}^{d_k \times d}$, and $\boldsymbol{W}_V^i \in \mathbb{R}^{d_v \times d}$. These are used to project the inputs into separate key, value, and query embeddings separately for each head, with the rest of the self-attention computation remaining unchanged. The output of each of the h heads is of shape $N \times d_v$, and so the output of the multi-head layer with h heads consists of h vectors of shape $N \times d_v$. To make use of these vectors in further processing, they are combined and then reduced down to the original input dimension d. This is accomplished by **concatenating** the outputs from each *head* and then using yet another linear projection, $\boldsymbol{W}_O \in \mathbb{R}^{h\,dv \times d}$, to reduce it to the original output dimension for each token, or a total $N \times d$ output

$$\begin{aligned}
\text{MultiHeadAttn}(\boldsymbol{X}) &= (\boldsymbol{head}_1 \oplus \ldots \oplus \boldsymbol{head}_h)\,\boldsymbol{W}_O \quad \in \mathbb{R}^{N \times d} \\
\boldsymbol{head}_i(\boldsymbol{Q}_i, \boldsymbol{K}_i, \boldsymbol{V}_i) &= \text{softmax}\left(\frac{\boldsymbol{Q}_i \boldsymbol{K}_i^T}{\sqrt{d_k}}\right)\boldsymbol{V}_i \quad \in \mathbb{R}^{N \times d_v} \\
&\text{where } \boldsymbol{Q}_i = \boldsymbol{X}(\boldsymbol{W}_Q^i)^T, \;\; \boldsymbol{K}_i = \boldsymbol{X}(\boldsymbol{W}_K^i)^T, \;\; \boldsymbol{V}_i = \boldsymbol{X}(\boldsymbol{W}_V^i)^T
\end{aligned}$$

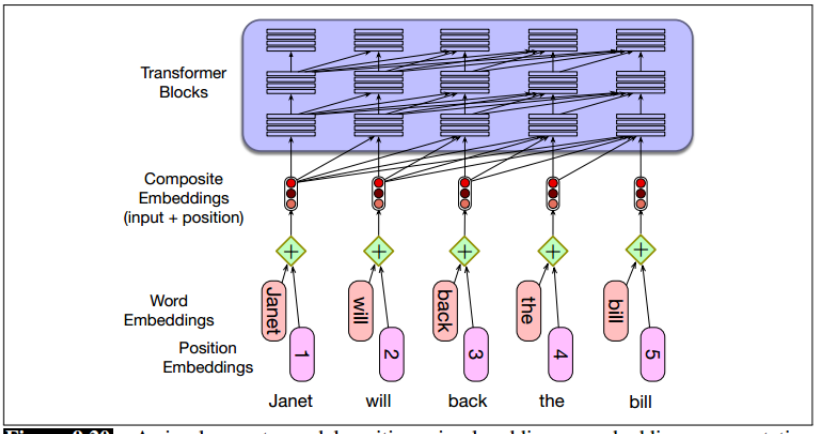

Figure 9.20 A simple way to model position: simply adding an embedding representation of the absolute position to the input word embedding.

Figure 7: The simple way to implement positional embedding.

4.3 Modeling word order: positional embeddings

The attention mechanism itself does not contain information on the relative, or absolute positions of the tokens in the input. One simple solution is to modify the input embeddings by combining them with **positional embeddings** specific to each position in an input sequence.

A simple implementation of postitional embedding is to add word embedding with the embedding of the position index, as shown in Figure 7. A potential problem with the simple ***absolute position*** *embedding approach* is that there will be plenty of training examples for the initial positions in our inputs and correspondingly **fewer at the outer length limits**. An alternative approach to positional embeddings is to choose a *static function* that maps integer inputs to real-valued vectors in a way that **captures the inherent relationships among the positions**. That is, it captures the fact that position 4 in an input is more closely related to position 5 than it is to position 17. A combination of *sine* and *cosine functions* with differing frequencies (**Fourier transform**) was used in the original transformer work. Developing better position representations is an ongoing research topic.

4.4 Encoder-Decoder structure

Figure 5 shows the architecture of transformer used for sequence-to-sequence task such as machine translation, text summarization, question answering.

4.5 Transformer for Contextual Generation and Summarization

A simple variation on autoregressive generation that underlies a number of practical applications uses a prior context to prime the autoregressive generation process. Figure 8 describes an auto-regressive process to generate text based on context. Here a standard language model is given the **prefix** to some text and is asked to generate a possible completion to it. Note that as the generation process proceeds, the model has direct access to the *priming context* as well as to all of *its own subsequently generated outputs*. This ability to incorporate the entirety of the *earlier*

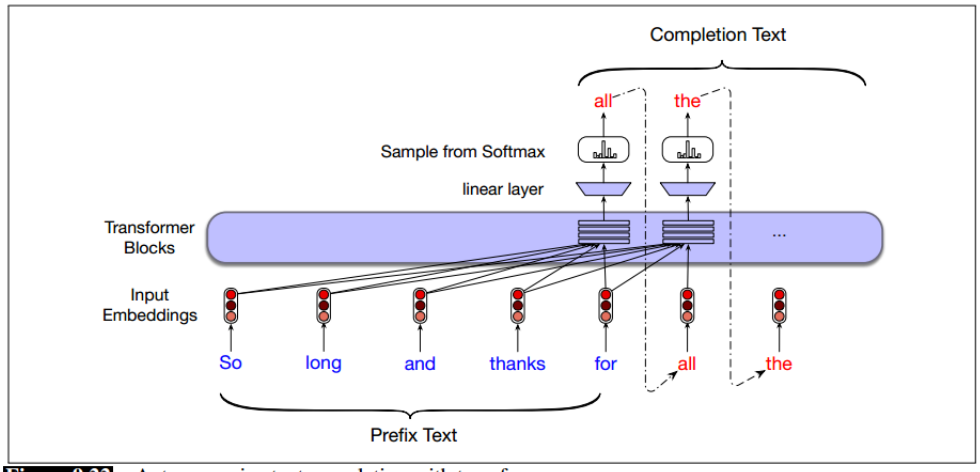

Figure 9.22 Autoregressive text completion with transformers.

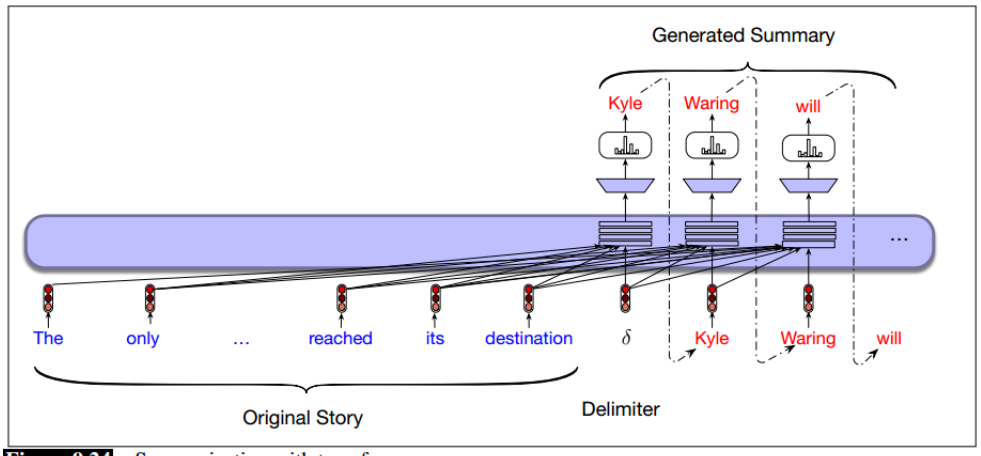

Figure 9.24 Summarization with transformers.

Figure 8: The autoregressive text generation using transformer.

context and generated outputs at each time step is the **key** to the power of these models.

Text summarization is a practical application of context-based autoregressive generation. The task is to take a full-length article and produce an effective summary of it. A simple but surprisingly effective approach to applying transformers to summarization is to **append** a *summary* to each full-length article in a *corpus*, with a unique ***marker*** separating the two. More formally, each article-summary pair $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$, $(y_1, \ldots y_n)$ in a training corpus is converted into a single training instance $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n, \delta, y_1, \ldots y_n)$ with an overall length of $n + m + 1$. These training instances are treated as long sentences and then used to train an autoregressive language model using **teacher forcing**, exactly as we did earlier.

Once trained, full articles ending with the special marker are used as the context to prime the generation process to produce a summary as illustrated in Figure 8.

References

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

Dan Jurafsky and James H Martin. Speech and language processing. vol. 3. *US: Prentice Hall*, 2014.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.