0.0.0.01em *

0pt3.25ex plus 1ex minus .2ex1.5ex plus .2ex

1 Introduction

We began our journey with word vectors. We looked at vectors on 2D plane in using python to understand analogies and importance of word vectors. Then we moved to RNNS (Recurrent Neural Networks). This architecture had vanishing gradients problem (gradients diminish exponentially as they are propagated back through time) which prevented it from learning long-term dependences in sequences. This issue was partially resolved by LSTMS (Long Short-Term Memory). We used LSTMs for neural machine translation. This was an interesting task. However, it was slow due to its sequential processing of input embeddings. We then moved to transformers to utilize the parallel processing capabilities of GPUs.

1.1 Recurrent Neural Networks(RNNs)

Let's first talk a briefly about RNNs. RNNs are a class of Neural Networks which take inputs sequentially and produce a hidden state.

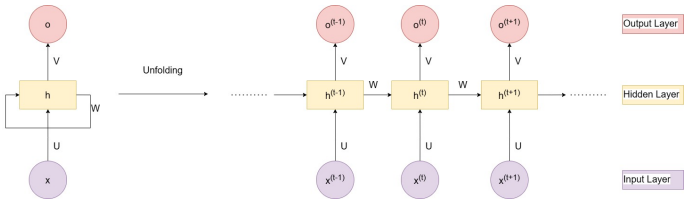

Figure 1: Basic RNN structure. Here x is input sequence, o is output sequence, and h is the hidden state. U, V, and W are training weights

The hidden state is calculated using the following equation:

$$h_t = g(Uh_{t-1} + Wx_{t-1}) \tag{1}$$

Here, h_t is the hidden state at time t, g is the activation function.

Then, using the following equation output is calculated:

$$o_t = f(Vh_t) \tag{2}$$

Here, o_t is the output at time t, f is another activation function.

We studied bidirectional RNNs. **Biderection RNN** combines two independent RNNs, one where the input is processed from the start to the end and the other where the input is processed from the end to the start. We then concatenate the two hidden states into one vector. This vector captures both left and right contexts of an input at each point in time. Vector concatenation is shown like this:

$$h_t = [h_t^f; h_t^b] \tag{3}$$

RNNs are good at learning short term dependencies but fail to learn short term dependencies becauuse of the vanishing graddients problem.

1.2 Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM) networks are an advanced type of recurrent neural network (RNN) designed to handle sequential data while addressing the vanishing gradient problem to some extent. LSTMs achieve this by introducing memory cells and gating mechanisms that regulate the flow of information.

An LSTM cell consists of three primary gates:

- **Forget Gate**: Determines which part of the previous cell state (C_t-1) should be discarded:

$$\mathrm{f}_t = \sigma(\mathrm{W}_f \cdot [\mathrm{h}_{t-1}, \mathrm{x}_t] + \mathrm{b}_f) \tag{4}$$

  where f_t is the forget gate output, W_f and b_f are the weights and biases, h_{t-1} is the hidden state from the previous time step, and xt is the current input.

- **Input Gate**: Decides which new information to store in the cell state:

$$\mathrm{i}_t = \sigma(\mathrm{W}_i \cdot [\mathrm{h}_{t-1}, \mathrm{x}_t] + \mathrm{b}_i) \tag{5}$$

$$\tilde{\mathrm{c}}_t = \tanh(\mathrm{W}_c \cdot [\mathrm{h}_{t-1}, \mathrm{x}_t] + \mathrm{b}_c) \tag{6}$$

  Here, it is the input gate output, and $\tilde{\mathrm{c}}_t$ is the candidate cell state.

- **Cell State Update**: Combines the forget and input gates to update the cell state:

$$\mathrm{i}_t = (\mathrm{f}_t \cdot \mathrm{C}_{t-1} + \mathrm{i}_t \cdot \tilde{\mathrm{C}}_t) \tag{7}$$

- **Output Gate**: Produces the new hidden state (ht):

$$\mathrm{o}_t = \sigma(\mathrm{W}_o \cdot [\mathrm{h}_{t-1}, \mathrm{x}_t] + \mathrm{b}_o) \tag{8}$$

$$\mathrm{h}_t = \mathrm{o}_t \cdot \tanh(\mathrm{C}_t) \tag{9}$$

In the equations above, σ is the sigmoid activation function, and tanh is the hyperbolic tangent function. LSTMs use these mechanisms to store, update, and retrieve information effectively, making them suitable for tasks like time series prediction, natural language processing, and speech recognition.

1.3 Transformers

Transformer, as described in the paper "Attention is all you need" and introduced in 2017 is an architecture that transforms one sequence into another sequence by using two blocks: Encoder and Decoder. Unlike LSTMs, RNNs etc. which used sequence-to-sequence techniques to process the data, transformer processes the data in parallel and relies on attention mechanisms, thus improving efficiency and performance. The advantage of attention mechanisms is that it speeds up the translation processed by the model. Some of the key components of the Transformer are as follows:

1.3.1

1.4 Bidirectional Encoder Representations from Transformers: BERT

1.4.1 Introduction

Prior to BERT word embedding models like Word2Vec1 and GloVe2 provide static representations for words where the same word like "bank" would always map on same vector. It doesn't depends on the context in which it is used. They were effective for some task but struggled with contextual meaning of sentence. The advent of transformer architecture with it's self-attention mechanism firstly enabled parallel processing solving a major problem of not utilising complete power of GPUs but also enabled long range dependency modeling. BERT was introduced in 2018 which leveraged these advancements to create deeply bidirectional contextual embedding revolutionizing Natural Language Processing (NLP).

1.4.2 Architecture

BERT's architecture is based on transformer encode from Vasani et al. 3, which uses stacked slef attention layers to process input sequence in parallel. Unlike decoder based models BERT uses the encoder to generate representation bidirectionally. Key architectural features includes Multi head self-attention, Positional embedding, Layer normalization and Residual connections that helps captures contextual relationship with words injects position of each word into the input tokens and stabilizes the training in deep networks respectively. BERT has two primary configurations: BERT-base (12 layers, 786 hidden dimensions, 12 attention heads) BERT-large(24 layers, 1024 hidden dimensions, 16 attention heads)

Tokenization

2 Tasks

2.1 Week 1

2.1.1 Rooshan Khan: Attention Method in bert.py

In this method we had to implement Attention method from class BertSelfAttention. The attention mechanism is given by:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{10}$$

where Q, K, and V represent the query, key, and value matrices, respectively, and d_k is the dimension of the key vectors.

I used method *torch.matmul* to multiply Q and transpose of K. I multiplied the result with *attention_mask* to apply the mask. The dimensions of attention_mask are $[bs, 1, 1, seq\_len]$. The attention mask distinguishes between non-padding tokens and padding tokens. The non-padding tokens have a value of 0 while padding tokens have a value of a large negative number. The dimensions of key_layer,query_layer and value_layer are $[bs, num\_attention\_heads, seq\_len, attention\_head\_size]$

Now I will tell how I concatenated all heads. When we transpose the tensor we change the shape of the tensor from $[bs, num\_attention\_heads, seq\_len, attention\_head\_size]$ to $[bs, seq\_len, num\_attention\_heads, attention\_head\_size]$. This enables us to reshape the tensor to $[bs, seq\_len, num\_attention\_heads * attention\_head\_size]$. After applying the transpose method the data sequence does not follow a contiguous order so we need to use **contiguous** method before using the **view** method.

2.2 Hussnain Amjad: add_norm Method in bert.py

The add_norm function is a critical component of the Transformer architecture, ensuring stable and effective training by combining residual connections, dropout, and layer normalization. It transforms the output of a sub-layer using a dense layer, applies dropout for regularization, and then adds it to the sub-layer's input through a residual connection. This output is further normalized using a layer norm to maintain consistent feature scaling across the model. This combination not only stabilizes gradient flow but also mitigates issues like vanishing gradients and overfitting, enabling the model to learn deeper representations effectively.

2.3 Areesha Noor: forward Method in bert.py

In the Bertlayer class, we have a forward function following add-norm function. The forward function is responsible for the overall flow from multi-head attention, followed by add-norm, a feed-forward layer and then another add-norm operation in the BERT architecture.So, the forward function transforms the input (hidden states) through series of sequential operations which represents the contextual relationships between tokens of sentences.

This was implemented by first computing attention scores by using "forward function" in BertSelfAttention class. By using this class, the hidden_states are processed through attention mechanism. The output of the attention is of the same size as of input i.e. $\mathbf{R^{N \times D}}$. Then, add-norm operation is applied to the attention scores using add_norm function in BertLayer class. Here, the residual connection is applied to attention layer. Next, for implementation of feed-forward layer, the normalized output of attention layer (attention_out_norm) is first passed through dense linear transformation (interm_dense). Mathematically,

$$\texttt{feedforward\_out} = \texttt{W}_1\texttt{.attention\_out\_norm} + \texttt{b}_1 \tag{11}$$

where W_1 is weight matrix and b_1 is bias vector.

Then, activation function GELU (Gaussian Error Linear Unit) is applied to introduce non-linearity in the model. Mathematically,

$$\texttt{GELU} = \frac{1}{2}.(1 + tanh(\frac{\pi}{2})).(x + 0.044715x^3) \tag{12}$$

Finally, add-norm is applied by passing the feed-forward network through add-norm function for residual connection.

2.4 Week 2

2.4.1 Rooshan Khan and Abdul Samad: step method in optimizer.py

The pseudocode for the Adam Optimizer was given in the default project document file. We followed it. We had to implement only one step. Hence, the loop is not needed. We have written the code for both cases: More efficient and Less efficient. `self.param_groups` is a list with only one element that is a dictionary. The first for loop in the step() method acceses the only element of the list(i.e. dictionary). The state dictionary contains current values of time(t), momentum(mt), velocity(vt).