HW4 Part 1

11-785 Fall 2023

Danso, Josh, Harshit, Harini

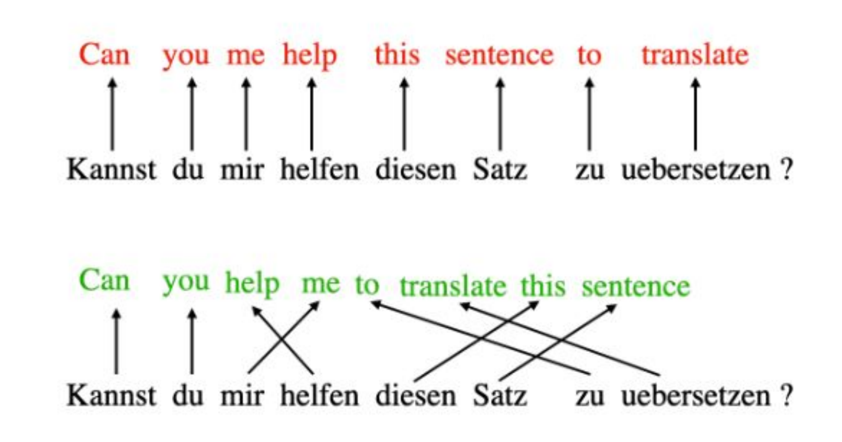
Overview

- This homework focuses on
  - Self - Attention (20 points)
  - Language Modelling (80 points) - estimating the probability distribution of token (in our case word) sequences in a language
  - Two main tasks in Language Modelling:
    - Prediction (40 points)
    - Generation (40 points)

Self-Attention

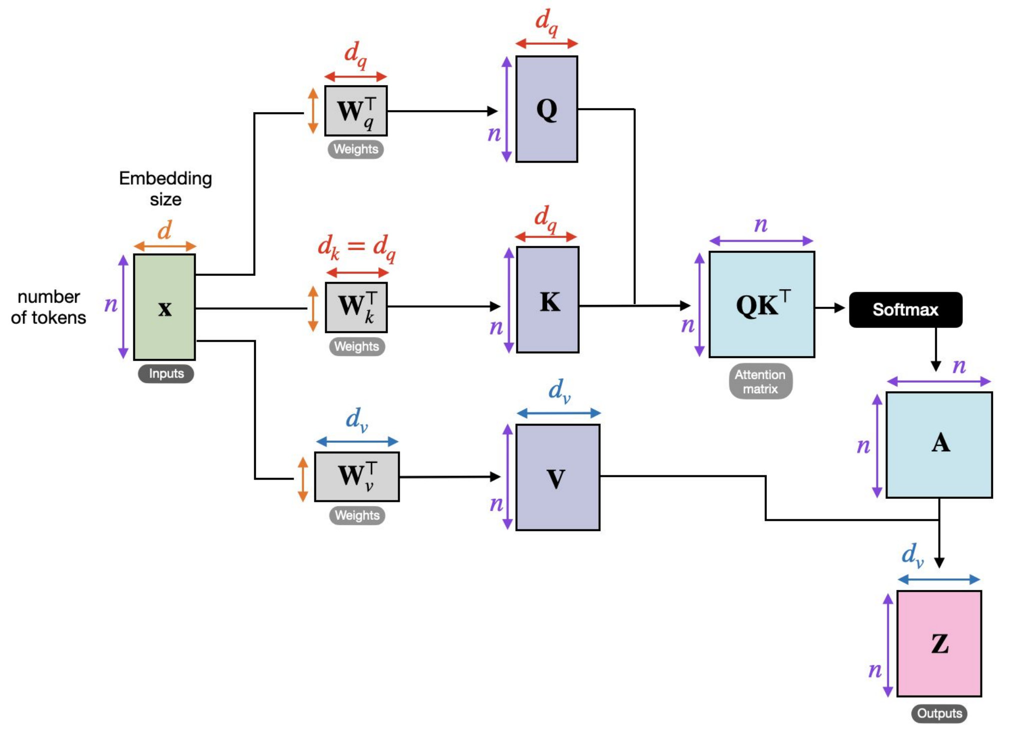
Self Attention

- In simple terms, self-attention allows the inputs to interact with each other ("self") and find out where they should pay more attention to ("attention").
- It gives access to all sequence elements at each time step.
    - The goal is to be selective and determine which words are most important in a specific context.



[Understanding and Coding the Self-Attention Mechanism of Large Language Models From Scratch (sebastianraschka.com)]

How does Self Attention work?

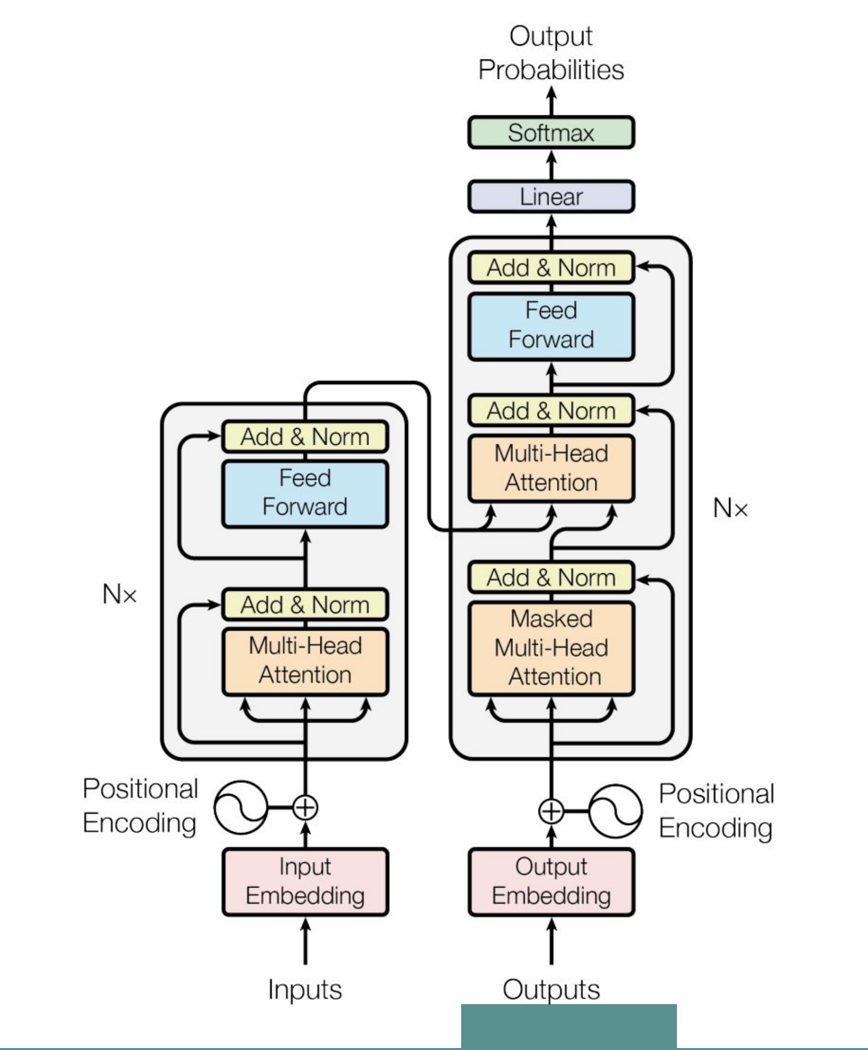


This shows the forward pass. Traverse back these same paths to perform the backward pass!

[Understanding and Coding the Self-Attention Mechanism of Large Language Models From Scratch (sebastianraschka.com)]

Attention in the broader context

A very important component in Transformers (HW4P2)!



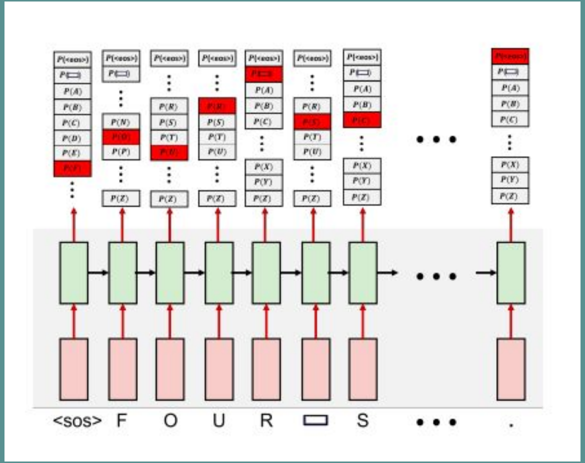
Language Modelling

Computing Probabilities of token sequences

- Problem: How do we know that our sequence is a *complete* sentence?
    - Sequence markers! <sos> is prepended and <eos> is appended to sequences to indicate completeness
- Problem: How do we represent tokens?
    - One-hot vector? Wasteful due to high dimensionality
    - Solution: Projections! Project the vectors to a lower-dimensions, with linear transformation
- How to compute the probability of a token sequence?
    - Bayes Rule! See Figure below
    - Probability of a token sequence is product of probabilities of each nth token given all previous n-1 tokens

$$
\begin{aligned}
P(<\text{sos}>, \mathrm{t}_1, \mathrm{t}_2, \cdots, \mathrm{t}_N <\text{eos}>; \theta) \;&=\; P(t_1 | <\text{sos}>; \theta) \mathrm{P}(\mathrm{t}_2 | <\text{sos}> \mathrm{t}_1; \theta) \mathrm{P}(\mathrm{t}_3 | <\text{sos}> \mathrm{t}_1 \mathrm{t}_2; \theta) \\
&\quad \cdots P(<\text{eos}> \mid <\text{sos}> \mathrm{t}_1 \cdots \mathrm{t}_N; \theta)
\end{aligned}
$$

Computing Probabilities of token sequences



- How do we compute these probabilities?
- These probabilities are computed by our RNN: can compute the conditional probability of nth token by a given previous n-1 tokens as inputs
- Steps (outlined in writeup):
  1. The entire sequences of characters is fed into the network.
  2. At each step, the network outputs the probability distribution for the next character in the sequence.
  3. At each time we select the probability assigned to the next character in the sequence
  4. The product of all of the selected probabilities is the total probability of the sentence

Pseudocode for computing probabilities

Pseudocode 1: Computing the probability of a word sequence

```
# Takes in token sequence T and returns its log probability
function LogProb = ComputeTokenSequenceLogProbability(T)
    LogProb = 0
    h = initial_h # Assuming that h[-1] = hinit is part of the model
    i = 0 # Assuming T[0] = <sos>
    do
        E = P*onehot(T[i]) # P is the projection matrix from one-hot vectors to embeddings
        [Y, h] = RNNstep(h, E)
        LogProb = LogProb + log(Y[T[i+1]])
        i = i + 1
    until (T[i] = <eos>) # Stop when we hit the end of the sequence
    return LogProb
end
```

Training a Language Model

$$\log P(S_i\theta) = \sum_{t=1}^{l_i+1} \log P(T_{i,t}| < sos >, T_{i,1}, \cdots, T_{i,t-1}; \theta)$$

- Standard approach for training parametric probability-distribution models – maximum likelihood estimation.
- The log probability of any sequence S_i is given by the Figure above
- The loss is equivalent to the negative total log probability of the training data
- MLE computes the theta that minimizes this loss

$$Loss(\theta) = -\sum_i \log P(S_i; \theta)$$

Dataset

- WikiText-2 Language Modeling Dataset
- Train dataset contain an array of articles - 579 articles.
- Each article is an array of integers, corresponding to words in the vocabulary.
- May concatenate those articles to perform batching, should shuffle articles between epochs
- Vocabulary file with an array of strings - 33,278 vocabulary items.

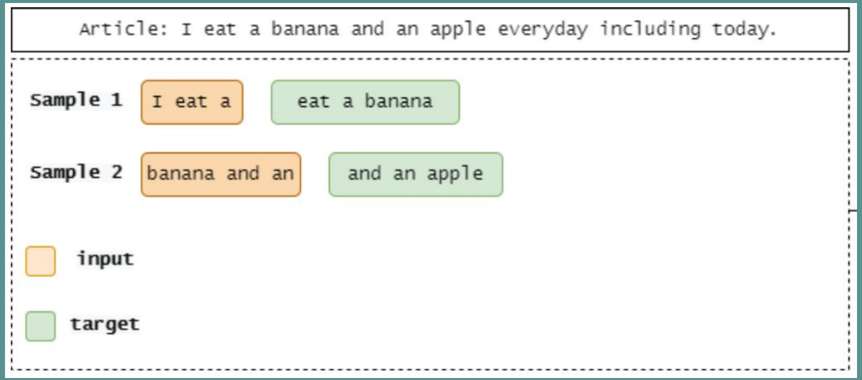
Dataloader

- Randomly shuffle all the articles from the WikiText-2 dataset.
- Concatenate all articles in the dataset.

- Divide the complete corpus into inputs and targets, where targets are shifted by a window of 1 word.

- Resize your inputs and targets into batches based on batch size and sequence length.

- Run a loop that yields a tuple of (input, target) in every iteration based on sequence length, with both these components being in shape (batchsize, seqlen).



Functions to implement

- Forward
- rnn_step
- predict
- generate

Forward

```
def forward(self, x, hidden_states=None):
    batch_size, timesteps   = x.shape
    embeddings              = self.embedding(x)
    token_probabilities  = []
    for t in range(timesteps):
        token_embedding_t           = NotImplemented
         # Get the appropriate embedding for the current timestep
        rnn_out, hidden_states      = NotImplemented
         # Feed the embedding through all LSTM Cells and retrieve the final hidden state output
        token_prob_dist             = NotImplemented
         # Map the hidden state output to a probability distribution
        token_probabilities.append(token_prob)
     # Stack the token probabilities along the time dimension
    return token_probabilities, hidden_states
```

rnn_step

```
def rnn_step(self, embedding, hidden_states_list):
    for i in range(len(self.lstm_cells)):
        # Forward pass through each lstm cell
        # Obtain the embedding and hidden_state

        hidden_states_list[i] = NotImplemented
        embedding = NotImplemented


    return embedding, hidden_states_list
```

Evaluating a Language Model - Prediction

- Your task: Complete the function *predict* - predicting the next token
- For a large test set compute the probability distribution for the (k + 1)th token - if average predicted log probability for the last token (in the sequence) is large enough, LM may have been well estimated.
- Takes as input a batch of sequences
- Return your model's guess of what the next token might be.
- Returned array will assign a probability score to every token in the vocabulary
- Model will be evaluated based on the score it assigns to the actual next word in the test data. CUTOFF: 5.2

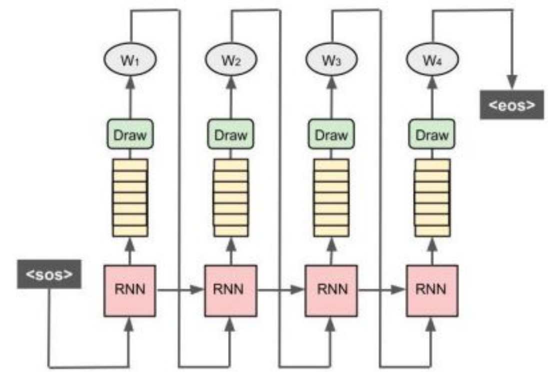
Pseudocode for Prediction

```
model_out   = NotImplemented
        # Pass the input sequence through the model

final_out = NotImplemented
      # Get the probability distribution of the last timestep

return final_out
```

Evaluating a Language Model - Generation



- We now know our model can compute probability distribution for the next character in the sequence.
- How do we generate tokens from these distributions?
- Based on the Bayes rule, a token sequence is drawn from the distribution shown in figure below below
- Each token is drawn from the corresponding term in the product
- Each nth token in the sequence is drawn from the distribution output by the network at the nth timestep, then fed back as input, affecting outputs for subsequent tokens.

$$\begin{aligned}\langle\text{sos}\rangle, \mathrm{t}_1, \mathrm{t}_2, \cdots, \mathrm{t}_N, \langle\text{eos}\rangle \quad \sim \quad & P(t_1 | \langle\text{sos}\rangle; \theta)\mathrm{P}(\mathrm{t}_2 | \langle\text{sos}\rangle\ \mathrm{t}_1; \theta)\cdots \\ & P(t_N | \langle\text{sos}\rangle\ \mathrm{t}_1, \cdots, \mathrm{t}_{N-1}; \theta)\mathrm{P}(\langle\text{eos}\rangle \mid \langle\text{sos}\rangle\ \mathrm{t}_1 \cdots \mathrm{t}_N; \theta)\end{aligned}$$

Evaluating a Language Model - Generation

- Sequence Completion
- How well does model extend (or completes) partial token sequences.
- Given partial word sequence, use LM to extend N words in the sequence.
- Initial sequence of words is input, begin drawing words from the output at the last word in the sequence.
- Goal: find an extension that is highly probable according to our LM. If LM is well trained, high-probability extensions are likely to be closer to distribution of training data, and are more plausible.

Evaluating a Language Model - Generation

- Your task: Complete the function *generate*
- As before, this function takes as input a batch of sequences
- Should generate an entire sequence of words.
- Requires sampling the output at one time-step and incorporate that in the input at the next time-step.
- YOU will evaluate the mean perplexity (e^NLL) of your generations using an LLM accessed through OpenAI API and report this number. CUTOFF: 1400

Pseudocode for Generation

```
token_prob_dist, hidden_states_list     = self.forward(x)

# Process the input to get both the predicted distribution for the next token and the hidden states for all cells

next_token                              = NotImplemented

# Draw the next predicted token from the probability distribution

generated_sequence = [next_token]

for t in range(timesteps):

        #   Process the next token and hidden_states_list through the model

        #   Get the most probable token for the next timestep

        generated_sequence.append(next_token)

                generated_sequence = torch.stack(generated_sequence, dim= 1) # keep last timesteps generated words

return generated_sequence
```

Training and Regularization Techniques

- Locked Dropout
- Embedding Dropout
- Weight Tying
- Activation Regularization
- Temporal Activation Regularization

General tips

- Start early!!!
- Read the write up carefully
- Remember to do the HW4 quiz on Canvas! (Deadline: Nov 16 11:59pm)
- Make sure the validation prediction NLL is sufficiently low before submitting (0.5-0.7 margin)
- ONLY 5 SUBMISSIONS ALLOWED!

Thank you! Any questions?