CS 4NL3 Final: Never Learning Edition

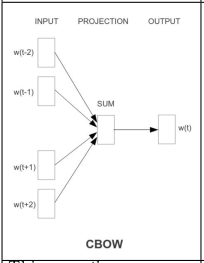
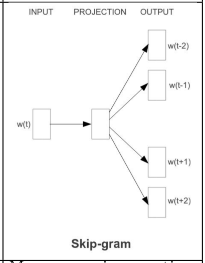
Static Word Embeddings

Distributional Semantics the idea that a word's meaning is given by the words that frequently appear nearby.

Word2Vec

Train a 2 layer neural network on one of the following tasks Given a middle word and contexts from $[w(t-2), w(t+2)]$...

Continuous Bag of Words (CBOW)	**Skip Gram**
Sum the words in the context window ("cumulative bag of words") and predict the missing word in the middle	You have the middle word and you use it to predict the context, individually, one at a time.

 INPUT
PROJECTION
OUTPUT
w(t-2)
w(t-1)
SUM
w(t)
w(t+1)
w(t+2)
CBOW
	 INPUT
	PROJECTION
	OUTPUT
	w(t)
	w(t-2)
	w(t-1)
	w(t+1)
	w(t+2)
	Skip-gram

This *smooths over* context words. Worse representation for rare words. Better representations on small corpora.	More seen in practice, better for large corpora (million iterations). Can be sped up with negative sampling where you predict "are they in the context window or not" (binary classification) as opposed to predicting the word directly.

Once this model is trained you will have two (one from each layer) weight matrices (an encoder $V \times D$ and decoder $D \times V$). The encoder becomes our embedding matrix.

Issues

- Black Sheep Problem (Maxim of Quantity: contribute as much information as required, but no more Paul Grice)
- Type of Similarity (sound, type-of) "flatted"
- Lack of Context - River vs Financial bank
- Human Biases
- Hard to Interpret

PPMI Matrix

Instead of weighing words based on which documents they appear in we can weigh them based on which words they co-occur with. Assuming $w, c \in V$

$$PMI(w,c) = \log_2\left(\frac{P(w,c)}{P(w)P(c)}\right)$$

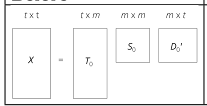
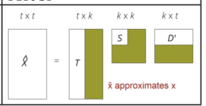
If two events a and b are totally unrelated, then $P(a,b) = P(a)P(b)$

You can represent a word as a vector of their PPMI values with all other words in the vocab!

$$PPMI(w,c) = \max\left(\log_2\left(\frac{P(w,c)}{P(w)P(c)}\right), 0\right)$$

Latent Semantic Indexing (LSI)

Factor a x = term × term matrix into $T_0 S_0 D_0$ using Singular Value Decomposition (SVD), where S_0 tells you how important a concept is, sort it by decreasing value. So set all low values to 0 to get an approximate $\hat{x}$.

Before	**After**
 t x t t x m m x m m x t X = T0 S0 D0'	 t x t t x k k x k k x t X̂ = T S D' x̂ approximates x

Global Vectors (GloVe)

1. Build the co-occurrence matrix
2. **Initialize:** Target word vectors $w_i \in \mathbb{R}^D$, Context word vectors $\hat{w}_j \in \mathbb{R}^D$, Bias terms $b_i, \hat{b}_j \in \mathbb{R}$
3. **Optimize loss function:** $\sum_{i,j=1}^{V} f(X_{ij})\left(w_i^\top \hat{w}_j + b_i + \hat{b}_j - \log X_{ij}\right)^2$ where $f(X_{ij})$ reduces the influence of very frequent or rare word pairs
4. **Final embeddings** $i = w_i + \hat{w}_i$

GloVe vectors use global (whole corpus) co-occurance statistics. This makes them more stable than Word2Vec.

Recurrent Neural Networks (RNN)

Forward pass

```
$h_0 \leftarrow 0$
    for $i \leftarrow 1$ to length($x$) do
    $h_i \leftarrow g(Uh_{i-1} + Wx_i)$
    $y_i \leftarrow f(Vh_i)$
    return $y$
```

Backpropagation you unroll the network

Objective Function Negative log likelihood $L_{CE} = -\sum_{w \in V} y_t[w] \log \hat{y}_t[w]$

With Embedding Layer

Before the forward pass, embed $e_t = Ex_t$

Weight Tying

Before	**After**
$x_t \in \lvert V\rvert \times 1$ $h_t \in d_2 \times 1$ $y_t \in \lvert V\rvert \times 1$ $E \in \lvert V\rvert \times d_1$ $W \in d_1 \times d_2$ $U \in d_2 \times d_2$ $V \in d_2 \times \lvert V\rvert$	$x_t \in \lvert V\rvert \times 1$ $h_t \in d \times 1$ $y_t \in \lvert V\rvert \times 1$ $E \in \lvert V\rvert \times d$ $W \in d \times d$ $U \in d \times d$ $V \in d \times \lvert V\rvert$ $E = V^\top$

Options for Initialization

	Yes	**No**
Weight Tying	Input and outputs matrices are the same	Input and output are separate layers
Pretraining	Embeddings are initialized randomly	Embeddings initialized from pretraining
Weight Freezing	Embeddings update while training	Input embeddings frozen

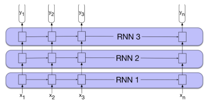
Low compute? Initialize pretrained embeddings.

Worried about rare words? Untie weights, freeze input.

RNN for Sequence Labeling

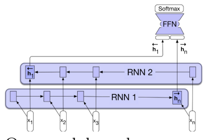
Before NNs, there was (Generative) Hidden Markov Model and (Discriminative) Conditional Random Field. Now, you can take either the last time step of an RNN (assuming all time steps are "remembered"), or weight all the hidden states equally and plug it into an NN classifier.

RNN Stacking



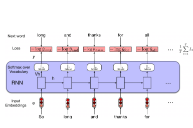
Different layers learn different levels of "abstraction"

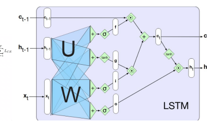
Bidirectional RNN



One model reads forward, the other backwards, then concatenate the two hidden states.

Long Short-Term Memory Network (LSTM)





Gates (2 columns)

Forget Gate Decide what old information to erase from memory.

$$f_t = \sigma(U_f h_{t-1} + W_f x_t)$$

$$k_t = c_{t-1} \cdot f_t$$

Mask Content Figure out what new information might be useful.
$g_t = \tanh(U_g h_{t-1} + W_g x_t)$

Add Gate Decide how much of that new information to actually keep.

$$i_t = \sigma(U_i h_{t-1} + W_i x_t)$$

$$j_t = g_t \cdot i_t$$

New Context Vector Combine the remembered past with the newly approved content.

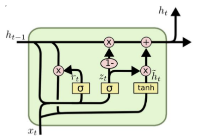
$$c_t = j_t + k_t$$

Output Gate Decide what part of the memory we show to the world.

$$o_t = \sigma(U_o h_{t-1} + W_o x_t)$$

$$h_t = o_t \cdot \tanh(c_t)$$

Gated Recurrent Unit (GRU)



$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$$
$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$$
$$\tilde{h}_t = \tanh(W \cdot [r_t * h_{t-1}, x_t])$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

1. **Interpolation Factor** z_tThis is like our "slider" between old and new information.
2. **Gate** r_t It is like a forget and add vector in one! (Linear Interpolation)
3. **Combine "New" and "Previous" Hidden States** h_t We use z_t to take from both accordingly

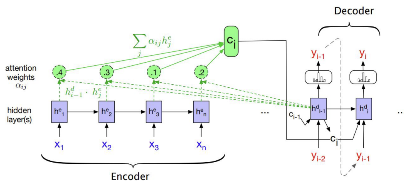
Encoder-Decoder RNN

1. Encoder takes input sequence and generates a sequence of states $h_n^{\oplus} = c = h_0^d$
2. Compute a context representation c
3. Decoder takes context and generates output sequence $h_t^d = g(\hat{y}_{t-1}, h_{t-1}^d, c)$

If we use an Encoder-Decoder RNN, the last hidden state is biased towards information at the end of the sentence which may not contain Information from earlier in the sequence.
Some ideas:

- use c at each step in decoding
- average all encountered hidden states to compute c
- weigh them appropriately with attention

Encoder-Decoder RNN with Attention



We compute

$$\text{score}(h_{i-1}^d, h_j^e) = h_{i-1}^d \cdot h_j^e$$

(represents how similar these two states are)
Then you use soft-max to normalize that into a probability distribution

$$\alpha_{ij} = \text{softmax}(\text{score}(h_{i-1}^d, h_j^e))$$

This givens you a distribution of those hidden states and how much they are relevant to the current state.
Now we can take a weighted sum of the encoder states at each decoder timestep, this contains the information that is relevant for that step of decoding and changes at each timestep.

Contextualized Word Vectors

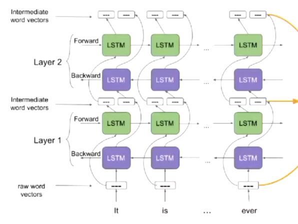
Contextualized Word Vector (CoVe)

Train an Encoder-Decoder LSTM to translate sentences (use GloVe to initialize embedding layer).
The hidden states of this encoder are the CoVe vectors (contextualized embeddings)

$$\text{CoVe}(w) = \text{MT-LSTM}(\text{GloVe}(w))$$

These are then concatenated with the GloVe vectors and passed to your downstream model.

Embeddings from Language Models (ELMo)



1. raw static word embeddings at the first layer
2. 2-layer BiLSTM, trained to predict the next word (language modeling), the lower layer will capture syntax, the upper layer will capture semantics
3. learned weights to combine them into your final embedding

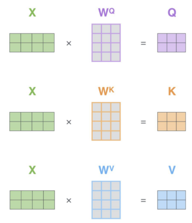
This one can distinguish between "play" in sports vs acting. Better performance than CoVe.

Muli-task Modeling

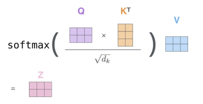
Train multiple tasks at the same time. The network may learn useful information for both tasks e.g. Name-Entity Recognition (NER) + Part-of-Speech (POS) Tagging. Earlier layers share parameters, learning representations useful for both tasks.

Self-Attention (Transformers)

How similar each word is to all of the words (or previous words) in a sentence, **including itself**.



- **Query** current element (being compared to preceding inputs)
- **Key** preceding input to be compared to current element (other vectors, including itself)
- **Value** value of preceding element to get weighted and summed

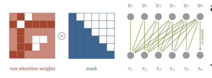


- $x_t \times W_Q$, $x_t \times W_k$, $x_t \times W_v$ to get queries, keys, values, respectively
- $QK^\top$ similarity between the queries and keys by taking their dot product

- $\frac{1}{\sqrt{d_k}}$ for numerical stability
- softmax amongst all the keys → element i of this output vector will represent what percentage of this query will be represented by key i
- $\times V$ sums i^{th} element from our SoftMax output with Value i for all i words in our sentence.

Similar idea to what we did before with our CoVe except before our K and V were the same (Q was the decoder state and K was the encoder state).

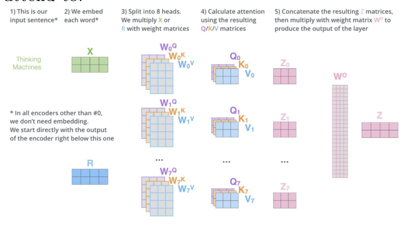
Masked Attention often, we don't want to earlier words to attend to the later words



Note we can compute the Attention for each word in parallel. This is super efficient and awesome. Transformers handle long-range dependencies effectively — they can capture the context of a word at the end of a sentence even if the relevant information appears at the beginning. This is something RNNs often struggle with.
Note we do not necessarily have to use Dot Product, we can use any cosign similarity or a normalized dot product i.e.
$= \frac{\text{Dot Product}}{\text{\# of Embedding Values}}$

Multihead Attention

We can create multiple heads that learn different functions of what linguistic (or other) features to attend to.


1) This is our input sentence*
2) We embed each word*
3) Split into 8 heads. We multiply X or R with weight matrices
4) Calculate attention using the resulting Q/K/V matrices
5) Concatenate the resulting Z matrices, then multiply with weight matrix WO to produce the output of the layer
Thinking Machines
X
* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one
R
W0Q
W0K
W0V
W1Q
W1K
W1V
W7Q
W7K
W7V
Q0
K0
V0
Q1
K1
V1
Q7
K7
V7
Z0
Z1
Z7
WO
Z

Tokenization

Byte-Pair Encoding (BPE)

Start with single characters in the vocabulary, take the most frequent pair of tokens and concatenate them. Merge greedily in order. This means that token frequencies in test set (after tokenizer is learned) do not affect how the text will be tokenized.
Add _symbol to denote the end of a word
We end up with tokens for frequent words and subwords (morphemes).

Word Piece

Initial vocab is each unit in the vocabulary.
Compute pair score score = $\frac{\text{freq of pair}}{\text{freq of first element} \times \text{freq of second element}}$ this looks similar to PMI! Merge until we have a new vocabulary
We denote each section with two hashtags in front ##

Sampling

Top-k

- Assign probabilities to all words in a vocabulary, model the softmax output
- Sort words by probability and take only the top k words
- Renormalize and sample the next word from these top k

Here, we are limiting the distribution by a fixed number of words.

Top-p

- Assign probabilities to all words in a vocabulary, model the softmax output
- Sort words by probability and take only the top p probability mass
- Renormalize and sample the next word from these top p

Temperature Sampling

Reshape the distribution instead of truncating

$$y = \text{softmax}\left(\frac{u}{\tau}\right)$$

With temperature sampling $\tau \le 1$, probability of probable words increase and probability of rare words decrease.
So larger τ, more diversity, and as $\tau \to 0$, less diversity (we approach Top-1 sampling).

Fine-tuning

Bottleneck Adapter

add new layers between transformer layers by "projecting" (linear layer) the hidden states down and then back up (plus a residual connection, r)

$$h \leftarrow W_{\text{up}} \cdot f(W_{\text{down}} \cdot h) + r$$

We freeze all other parameters in the model and only update the new layers (FF Up and FF Down)

Low-Rank Adaptation (LoRa)

learns low-rank matrices in place of the transformer layer weights for keys, values, queries, and output (W^Q, W^K, W^V, W^O)
Freeze all original parameters and only update the low-rank adaptations
E.g., you can factor $W \in Nxd$ matrix into $A \in Nxr$ and $B \in rxd$ Then in the forward pass we compute $h = xW + xAB$
You can use a scaling parameter a to weight the impact of new parameters. For faster inference, merge $AB + W \to W'$ (after training).

Evaluation of Language Models

Intrinsic

Projection

Visualize word vectors in 2D to see if similar words are near each other.

Perplexity

The inverse probability assigned to test set normalized by length. Lower Perplexity is better, best Perplexity is 1.
So for example, if Model A has less words in its Vocabulary as opposed to Model B, with no other information, Model A is more likely to have a lower perplexity! Model A has fewer options even if you were randomly guessing the next word, it is more likely you will get the answer right with model A.
It's the inverse of the Probability that the model assigns to the held-out data (lower is better). Instead of simply saying "it's not in the right bin" (like Accuracy), we say, what is the probability that it will be in the right bin.

$$\text{Perplexity}(W)$$

$$= P(w_1 w_2 \ldots w_N)^{-\frac{1}{N}} = \sqrt[N]{\frac{1}{P(w_1 w_2 \ldots w_N)}}$$

For an N-gram Language Model

$$= e^{\left(-\frac{1}{N}\sum_{i=1}^{N}\log(P(w_i|w_1 \ldots w_{i-1}))\right)}$$

Extrinsic Evaluation

See how well the embedding performs for other downstream tasks.

Blue Score

How much overlap between generated model and a doc from training data, using different N-grams.

Few-Shot Learning

Original Definition

You create a feature representation of each class, and then you want to find which class the instance is closest to.

Prompting Definition

You prompt the model to perform classification.
One-Shot: One example **Zero-Shot:** No examples, only task description

Large Language Models

GPT-1 (2018)

Uni-directional.

1. Pretrain 12x Decoder-Only transformers for language modelling
2. Fine-tunefor Classification (Sentiment, Grammaticality), Question answering, Sentence similarity, Natural langage inference
3. They also had an auxillary objective function but it barely made a difference

Bidirectional Encoder Representations from Transformers BERT (2018)

Pretraining

Used

- **Masked Language Modeling:** Instead of predicting w_n given $w_1, \ldots, w_{n-1}$ (i.e. from left to right one by one), remove n random tokens and predict those instead. say, 50% of tokens.
- **Next Sentence Prediction** The task is to predict whether or not the given sentence follows the previous sentence.
- **CLS Token** Before, people were using "start tokens" for the beginning of a sequence. Alternatively [CLS] tokens (which stands for "classification") is the only token that will be used for classification tasks/objectives. By doing that, it encourages the model to encode the *entire meaning of the sentence* (in a way) in this [CLS] token because it is all that is being used to perform the classification.

Properties

- uses word piece tokenization
- encode-only
- 12 transformer layers stacked

Fine-Tuned

- Sentence Pair Classification Tasks
- Single Sentence Classification Tasks
- Question Answering Tasks
- Sentence Tagging Tasks

GPT-2 (2018)

Higher quality training data. 40GB of test.

RoBERTa (2019)

BERT improved

- Dynamic masks: different masks for each sentence
- No next-sentence prediction, turns out you can get a lot more out of increasing the context size and doing more masked prediction
- expanded masked language model context size
- larger mini-batches
- bye-pair tokenization
- more data
- longer training

GPT-3 (2020)

More data. 570GB of test.
Strong improvement on the LAMBADA (common sense) dataset.

Sentence-BERT (SBERT) (2018)

Used to create sentence embeddings.
[CLS] token embeddings or averaged hidden states perform poorly, so... use BERT base model to create high-quality sentence embeddings. How?

- Pool encoding for each sentence
- Concatenate vectors with element-wise difference
- Linear for classification (entailed, contradict, or neutral)

Often competitive with LLM encodings of sentences and generally much quicker.

Llama Models (2023-2025)

Models are very similar to other transformer LLMs (esp. GPT-3)

- Train on publicly available data (Llama 1) and scraped data (Llama 2 & 3)
- From 7B to 70B model sizes, context length 4K
- Llama 3 uses Direct Preference Optimization to learn rank w/o Reward Model

Properties of MLM Pretraining

- Contextual word representations
- improved generalization
- seeing wide range texts, fill in missing words
- fine tunable
- Robustness to missing data

Extra Last Minute Topics

Stochastic Parrot

LLMs cannot understand language, they are just convincing (bullshit)

Octopus

Models cannot learn meaning (not grounded in reality)

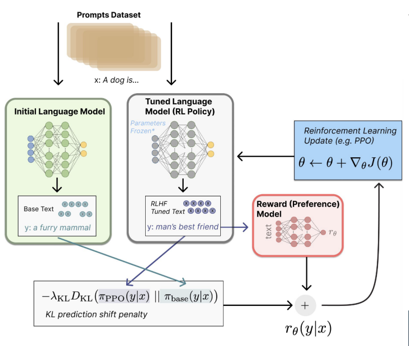
BBQ

The idea is that if enough Information is not provided, the model should say the data is Unknown, it is biased if it chooses a group.

Reinforcement Learning (RL)

The concept that you have some context (state) and need to choose a class (action), but you don't know if the output is good right away. We need to trace it back to determine which actions were helpful. A reward signal comes from actions taken by an agent and is used to update a policy that guides the agent's actions.

Reinforcement with Human Feedback (RLHF)


Prompts Dataset
x: A dog is...
Initial Language Model
Base Text
y: a furry mammal
Tuned Language Model (RL Policy)
Parameters Frozen*
RLHF Tuned Text
y: man's best friend
Reinforcement Learning Update (e.g. PPO)
$\theta \leftarrow \theta + \nabla_\theta J(\theta)$
Reward (Preference) Model
text
r_θ
$-\lambda_{\text{KL}} D_{\text{KL}}(\pi_{\text{PPO}}(y|x) \,||\, \pi_{\text{base}}(y|x))$
KL prediction shift penalty
$r_\theta(y|x)$

Instruction Tuning

Fine-tuning on data that contains instructions and answers.
How to get data:

1. Have people write examples
2. find examples from existing data e.g. QA, summarization
3. use the instructions given to the annotators
4. generate examples with LLMs and manually check instances before adding to the training data

(FLAN and T5)

Mixture of Experts (MoE)

1. different "expert" models can be created for different tasks/domain
2. let an NN decide which expert to use
3. you can set unused experts activations to 0 (more efficient)

Calcs

Sigmoid

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Theoretical Temperature controlled sigmoid

$$\sigma(x, T) = \frac{1}{1 + e^{-x/T}}$$

Log Rules

$\ln(a^x) = x\ln(a)$; $\ln(ab) = \ln(a) + \ln(b)$;
$\ln(\frac{a}{b}) = \ln(a) - \ln(b)$

Element wise multiplication

$a \odot b = [1, 2] \odot [4, 3] = [4, 6]$

Cross Entropy Loss

Given the true token t, and predicted probability for token $x = p_x$

$$L_{CE} = -\log(p_t)$$

Softmax

Given a vector X, we want to create a probability distribution of the elements in the vector.

$$softmax(X) = \frac{e^{x_i}}{\sum_j^{|X|} e^{x_j}}$$

Softmax with Temperature

$$softmax_\tau(X) = \frac{e^{(x_i/\tau)}}{\sum_j^{|X|} e^{(x_j/\tau)}}$$

The vector components should be *logits*, so X should not already be a probability distribution. Make sure you log The vector before you apply softmax.

Matrix Multipication

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \quad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$C = AB = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

Matrix Transpose

$$A = \begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} \qquad A^\top = \begin{bmatrix} a & c & e \\ b & d & f \end{bmatrix}$$

Gradient Descent Rule

$$f = wx + b, \quad L = f^2$$

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial w} = 2f \cdot x = 2(wx + b) \cdot x$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial b} = 2f \cdot 1 = 2(wx + b)$$

Review Questions

PPMI

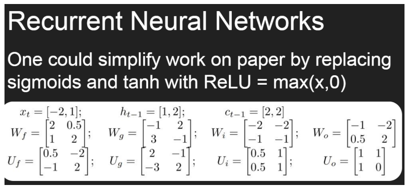
You will be given a term-doc matrix. Your job is to calculate the term-term matrix via a self dot-product – resulting in the following. Then sum each column/row, and the sum of that row - as seen below.

	spring	flowers	bounce	ball	game	like	person	sum
spring	2	2	1	1	0	6	6	**18**
flowers	2	4	0	0	0	6	6	**18**
bounce	1	0	1	1	0	3	3	**9**
ball	1	0	1	5	4	7	9	**27**
game	0	0	0	4	4	4	6	**18**
like	6	6	3	7	4	22	24	**72**
person	6	6	3	9	6	24	27	**81**
sum	**18**	**18**	**9**	**27**	**18**	**72**	**81**	**243**

$$PPMI(like, person) = \log_2\left(\frac{P(like, person)}{P(like)P(person)}\right)$$

$$= \log_2\left(\frac{24/243}{(72/243)(81/243)}\right) = \log_2(2) = 1$$

RNN Passthrough



Recurrent Neural Networks

One could simplify work on paper by replacing sigmoids and tanh with ReLU = max(x,0)

$x_t = [-2, 1]$; $h_{t-1} = [1, 2]$; $c_{t-1} = [2, 2]$

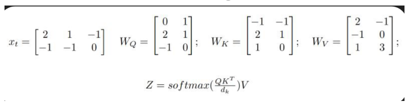
$W_f = \begin{bmatrix} 2 & 0.5 \\ 1 & 2 \end{bmatrix}$; $W_g = \begin{bmatrix} -1 & 2 \\ 3 & -1 \end{bmatrix}$; $W_i = \begin{bmatrix} -2 & -2 \\ -1 & -1 \end{bmatrix}$; $W_o = \begin{bmatrix} -1 & -2 \\ 0.5 & 2 \end{bmatrix}$

$U_f = \begin{bmatrix} 0.5 & -2 \\ -1 & 2 \end{bmatrix}$; $U_g = \begin{bmatrix} 2 & -1 \\ -3 & 2 \end{bmatrix}$; $U_i = \begin{bmatrix} 0.5 & 1 \\ 0.5 & 1 \end{bmatrix}$; $U_o = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$

Given the outlined in **Recursive Neural Network**, find $f_t, k_t, c_t, i_t, j_t, g_t, o_t, h_t$.

Answer: to Solve, just perform matrix multiplication (dot product) or pairwise multiplication where necessary.

Self-Attention Passthrough



$x_t = \begin{bmatrix} 2 & 1 & -1 \\ -1 & -1 & 0 \end{bmatrix}$ $W_Q = \begin{bmatrix} 0 & 1 \\ 2 & 1 \\ -1 & 0 \end{bmatrix}$; $W_K = \begin{bmatrix} -1 & -1 \\ 2 & 1 \\ 1 & 0 \end{bmatrix}$; $W_V = \begin{bmatrix} 2 & -1 \\ -1 & 0 \\ 1 & 3 \end{bmatrix}$;

$Z = softmax(\frac{QK^T}{d_k})V$

$$Q = x_t W_Q = \begin{bmatrix} 3 & 3 \\ -2 & -2 \end{bmatrix} \quad K = x_t W_K = \begin{bmatrix} -1 & -1 \\ -1 & 0 \end{bmatrix}$$

$$V = x_t W_V = \begin{bmatrix} 2 & -5 \\ -1 & 1 \end{bmatrix}$$

Question gives us d_K, the number of columns in $K = 2$.

$$QK^T = \begin{bmatrix} -6 & -3 \\ 4 & 2 \end{bmatrix} \quad A = d_K^{-1} QK^T = \begin{bmatrix} -3 & -1.5 \\ 2 & 1 \end{bmatrix} \tag{1}$$

Typically we solve for $(d_K^{-1/2})QK^T$, but here the question asks for $d_K^{-1}(QK^T)$. Note $K = K^T$ here. Shortform this to A.

$$softmax(A) = \begin{bmatrix} softmax([-3 & -1.5]) \\ softmax([2 & 1]) \end{bmatrix}$$

To solve softmax of a matrix, you have to solve it row by row or column by column. Since we did QK^t, we do row by row.

$$\text{softmax}(A)V = \begin{bmatrix} \frac{e^{-3}}{e^{-3}+e^{-1.5}} & \frac{e^{-1.5}}{e^{-3}+e^{-1.5}} \\ \frac{e^2}{e^2+e} & \frac{e}{e^2+e} \end{bmatrix} \begin{bmatrix} 2 & -5 \\ -1 & 1 \end{bmatrix}$$

WordPiece Tokenization

Given the following corpus, compute the first 2 merges using WordPiece:
`raw, how, how, wow, wow, who`

Merge Step 1:

Reconstruct the corpus using token pairs, then calculate scores via $S(h,t) = \frac{C(h,t)}{C(h)C(t)}$. Merge highest score greedily. Hack: Keep track of what was updated – rows with non-updated tokens don't need to be touched.

```
C: r=1, h=2, w=3, #a=1, #o=5, #h=1, #w=5

r #a:  1  -- S() = 1/(1*1) = 1 <<<
#a #w: 1 -- s() = 1/(1*5) = 1/5
h #o:  2  -- S() = 2/(2*5) = 1/5
#o #w: 4 -- S() = 4/(5*5) = 4/25
w #o:  2  -- S() = 2/(3*5) = 2/15
w #h:  1  -- S() = 1/(3*1) = 1/3
```

Merge Step 2:

When we reconstruct the token pairings, use the longest substring match. Thus, we use `ra` instead of `r`.

```
C: h=2, w=3, #o=5, #h=1, #w=5, >updated> ra=1, r=0, #a=0
ra #w: 1  -- S() = 1/(1*5) = 1/5
h #o:  2  -- S() = 2/(2*5) = 1/5
#o #w: 4  -- S() = 4/(5*5) = 4/25
w #o:  2  -- S() = 2/(3*5) = 2/15
w #h:  1  -- S() = 1/(3*1) = 1/3 <<
```

Tokenization Result

`1x(ra #w), 2x(h #o #w), 2x(w #o #w), 1x(wh #o)`

BPE Tokenization

Given the same corpus as above, compute the first 2 merges using BPE

Merge Step 1

Find the counts of all present token pairs

```
C: r, a, w, h, o, _

r a  -- 1
a w  -- 1
w _  -- 5 <<
h o  -- 3
o w  -- 4
w o  -- 2
w h  -- 1
o _  -- 1
```

Merge `w_`.

Merge Step 2

```
C: r a w h o _ w_

r a   -- 1
a w_  -- 1
h o   -- 3
o w_  -- 4 <<
w o   -- 2
w h   -- 1
o _   -- 1
```

Merge `o w_`.
Final Vocab:

```
Vocab: r a w h o _ w_ ow_

1x  r a w_
2x  h ow_
2x  w ow_
1x  w h o _
```