Attention - I

September 24, 2025 10:09 AM

Previously we had CNNS &RNNs

CNNs

Convolutional NNs
(spatial)
Capture information from neighbors
Work good locally
but struggle with long term dependencies

CNN review:

• conv. layer

$$I_{H \times W \times C} \xrightarrow[\text{with } K_{(f_h, f_w \times C)} \text{ filters}]{\text{Convolution Layer}} S_{H_{out}, W_{out}, C}$$

$$\begin{bmatrix} 10 & 11 & 3 & 8 \\ 2 & 15 & 12 & 19 \\ 7 & 1 & 18 & 11 \\ 13 & 4 & 3 & 14 \end{bmatrix} \longrightarrow \begin{bmatrix} -5 & -1 & -16 \\ 1 & -3 & 1 \\ 3 & -2 & 4 \end{bmatrix}$$

$I_{(4 \times 4 \times 1)}$ $\qquad$ $K(2,2,1), p=0, s=1$ $\qquad$ $S(3,3,1)$

$$S_{H_{out} W_{out} C} \xrightarrow{\text{Activation funct}} A_{H_{out} W_{out} C}$$

$$\begin{bmatrix} -5 & -1 & -16 \\ 1 & -3 & 1 \\ 3 & -2 & 4 \end{bmatrix} \xrightarrow{\text{ReLU}} \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 1 \\ 3 & 0 & 4 \end{bmatrix}$$

max(0,-5)

max(0,3)

$S(3,3,1)$ $\qquad$ $A(3,3,1)$

• Pooling Layer

$$A(H_{out}, W_{out}, C) \xrightarrow{\text{Pooling Layer}} P(H_{out}', W_{out}', C)$$

$$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 1 \\ 3 & 0 & 4 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 1 \\ 3 & 4 \end{bmatrix}$$

$A(3,3,1)$ $\qquad$ $k=2$, $s=1$ $\qquad$ $P(2,2)$

• Flattening Layer

$$P_{(H, W, C)} \xrightarrow{\text{Flattening Layer}} F_{(1, H.W.C)}$$

$$\begin{bmatrix} 1 & 1 \\ 3 & 4 \end{bmatrix} \longrightarrow [1, 1, 3, 4]$$

• Dense output Layer

$$F_{(1, H.W.C)} \xrightarrow[N \text{ neurons}]{\text{Dense output layer}} O_{(N,1)}$$

$$y = \sigma\left(\underbrace{\begin{bmatrix} w_{01} & w_{11} \\ \vdots & \vdots \\ w_{03} & w_{13} \end{bmatrix}}_{2 \times 4} \underbrace{\begin{bmatrix} 1 \\ 1 \\ 3 \\ 4 \end{bmatrix}}_{4 \times 1} + \underbrace{[b_1, b_2]}_{2 \times 1} \right)$$

$$= \sigma\left(\underbrace{\begin{bmatrix} w_1 x_1 + b_1 \\ w_2 x_2 + b_2 \end{bmatrix}}_{2 \times 1} \right) = \begin{bmatrix} p_1 \\ p_2 \end{bmatrix}$$

probability → Neuron1
→ Neuron2

Multi class classification
usually has N neurons
for N classes
where each neuron gives
probability of that class

Binary class. can use only 1 neuron

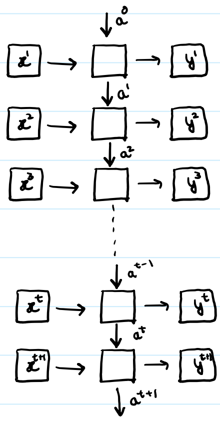
RNNs

Recurrent NNs
(Temporal)
capture information from sequence (Past values, possibly future)
good for sequential dependence
but struggle computationally (no parallelization, finite content)

RNN Review

hidden layer $h^{<t>} = \underset{\text{funct.}}{\text{act.}}\,(W_{hh}\, h^{<t-1>} + W_{xh}\, x^{<t>} + b_h)$

the output $y^{<t>} = \underset{\text{funct.}}{\text{act.}}\,(W_{hy}\, h^{<t>} + b_y)$



Attention

To capture both short term & long term dependencies more accurately Attention mechanism was proposed.

instead of us defining what to look at (as we do by architectural choices of CNN & RNN)
the network itself tries to figure out what's important to look at

we'll walk through one of the popular examples where attention mechanism definitely made its mark that is:
Natural Languge Processing

the idea is,

- Words are converted to vectors (in LLMs we usually call 1 entity be 1/2 a word or $1\frac{1}{2}$ word, a token)
- Using the above vectors we create 3 more vectors (linear projections of input using learnt weights)

$Q = I\,W_Q$ - Query

$K = I\,W_K$ - Key

$V = I\,W_V$ - Value

V: Projections (n_{tokens}, d_k)

I: Input Vector (embedded tokens) $(n_{tokens}, d_{embed.\,model})$

W_V: Weight Matrices (learnt) $(d_{embedding\,model}, d_k)$

- Compute relevance
  how much each word should attend to every other word

$$score = Q \cdot K^T$$

We also have to the score in real examples or else they can be too huge normalize

$$score\_normalized = \frac{Q \cdot K^T}{\sqrt{d_{emb.\,model}}}$$

Now we convert it into probabilities

$$\text{attention weights} = \text{prob.} = \text{softmax}(\text{score normalized}) = \underbrace{\text{softmax}\left(\frac{Q \cdot K^T}{\sqrt{d}}\right)}_{\text{This gives us the weight or attention of every word}}$$

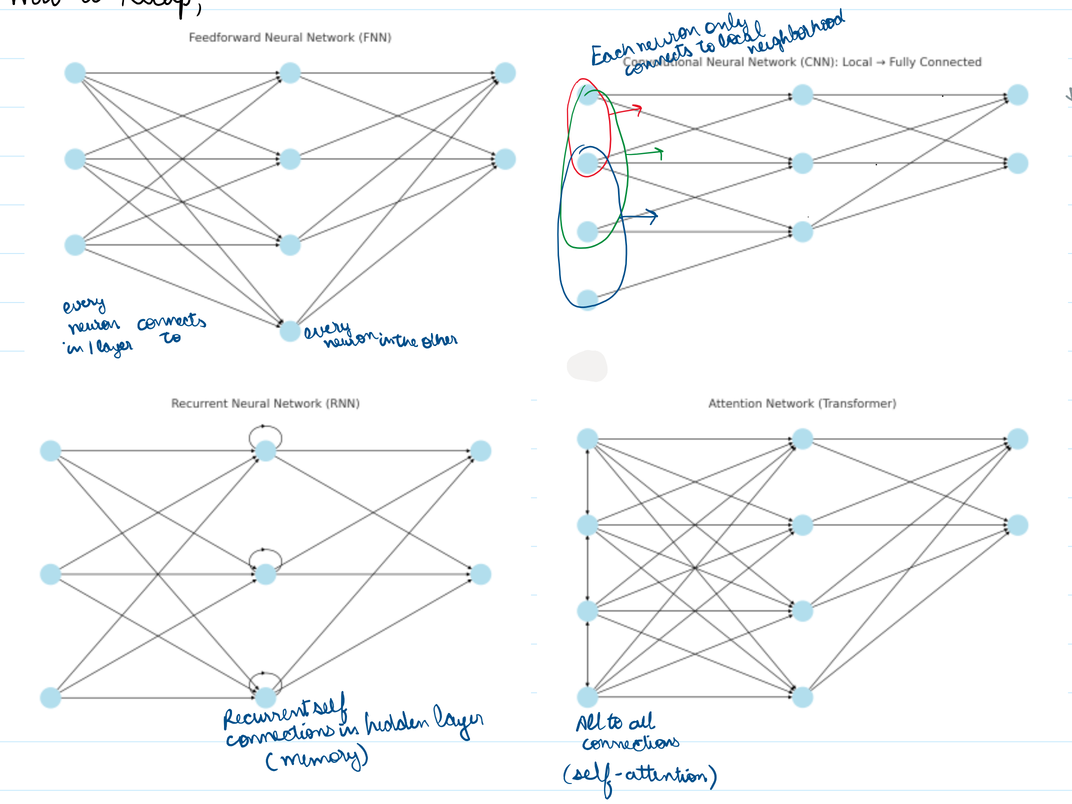
– Now that we bow, which word has how much importance, we incorporate is value accordingly

$$\underbrace{\text{output} = \text{attention weights} \cdot V}$$

Now that we know the weights, we add Value

Output of each neuron, would now represent the info from all other tokens (weighed by the attention)

Now to Recap,



This is for vis purposes convolution windows are of same size -first one would be 3 as well

Step 1 - Tokenize

Now lets encode the text (we just tokenize each word for simplicity)

```
I = np.array([
    [1,0,0,0,0,0,0,0,0,0],  # The
    [0,2,0,0,0,0,0,0,0,0],  # hungry
    [0,1,3,0,0,0,0,0,0,0],  # lion
    [0,0,1,3,0,0,1,0,0,0],  # attacked
    [1,0,0,0,0,0,0,0,0,0],  # the
    [0,2,0,0,0,0,0,0,0,0],  # slow
    [0,0,0,0,0,1,3,0,0,0],  # zebra
    [0,0,0,0,2,0,0,0,0,1],  # in
    [1,0,0,0,0,0,0,0,0,0],  # the
    [0,2,0,0,0,0,0,0,0,0],  # tall
    [0,0,1,0,0,2,0,1,0,0],  # grass
])
```

Let's not focus on the embedding model here, its just meant to assign a unique combination to each word

Let the above be our input matrix of 10×10

Step 2 - Initialize

Let's initialize 3 matrices W_Q, W_K, W_V
Query Key Value

In our example, we're randomly initializing these and the model will learn them over time

Let these be 2D matrices for simplicity, i.e. $W_Q, W_K = [10,3]$ $W_V = [10,2]$

$W_Q =$

```python
W_q = np.array([
    [0,0,0],   # DET
    [1,0,0],   # ADJ queries NOUN
    [0,2,0],   # NOUN queries VERB
    [1,1,1],   # VERB queries NOUN, OBJ, CONTEXT
    [0,0,0],   # DET
    [1,0,0],   # ADJ queries NOUN
    [0,0,2],   # OBJ queries VERB
    [0,1,1],   # PREP queries CONTEXT
    [0,0,0],   # DET
    [1,0,0],   # ADJ
])  # Shape: (10, 3)
```

$W_K =$

```python
W_k = np.array([
    [0,0,0],   # DET
    [2,0,0],   # ADJ offers descriptive info
    [2,0,0],   # NOUN identity
    [0,2,2],   # VERB offers strong verb/action
    [0,0,0],   # DET
    [2,0,0],   # ADJ
    [0,0,3],   # OBJ identity
    [0,1,1],   # PREP
    [0,0,0],   # DET
    [2,0,0],   # ADJ
])  # Shape: (10, 3)
```

$W_V =$

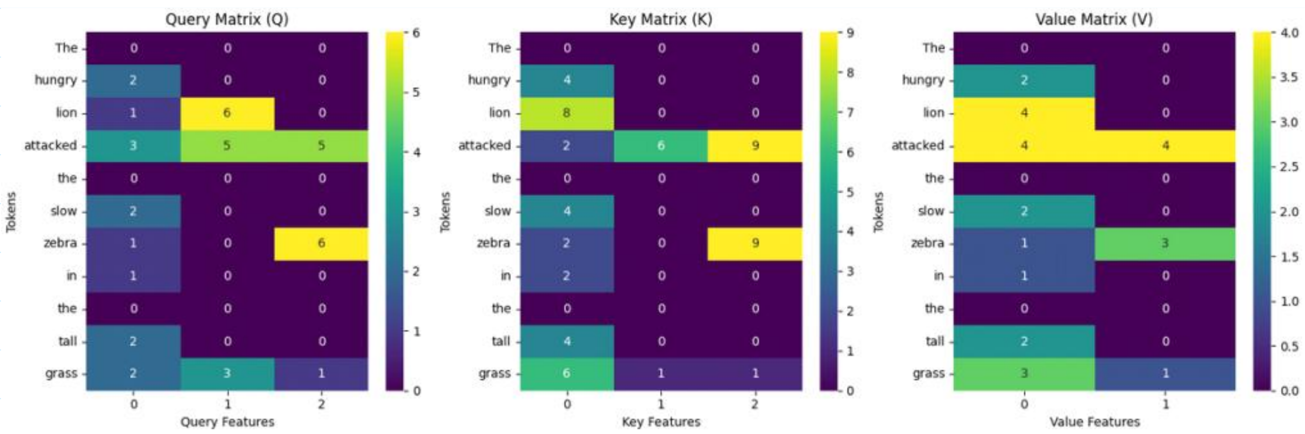
```python
W_v = np.array([
    [0,0],  # DET
    [1,0],  # ADJ info
    [1,0],  # NOUN info
    [1,1],  # VERB info
    [0,0],  # DET
    [1,0],  # ADJ
    [0,1],  # OBJ info
    [0,1],  # PREP/CONTEXT
    [0,0],  # DET
    [1,0],  # ADJ
])  # Shape: (10, 2)
```

Step 3 - Projections

$$\underset{10\times3}{Q} = I_{10\times10}\, W_{Q\,10\times3}$$

$$\underset{10\times3}{K} = I_{10\times10}\, W_{K\,10\times3}$$

$$\underset{10\times2}{V} = I_{10\times10}\, W_{V\,10\times2}$$

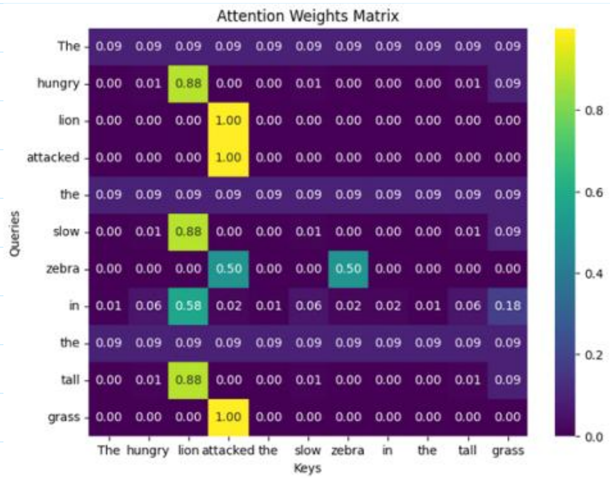


Step 4 - Scores

$$\text{score} = \frac{Q \times K^T}{\sqrt{\text{len}(Q)}}$$

```python
scores = Q @ K.T / np.sqrt(Q.shape[1])
exp_scores = np.exp(scores - scores.max(axis=1, keepdims=True))
weights = exp_scores / exp_scores.sum(axis=1, keepdims=True)
```

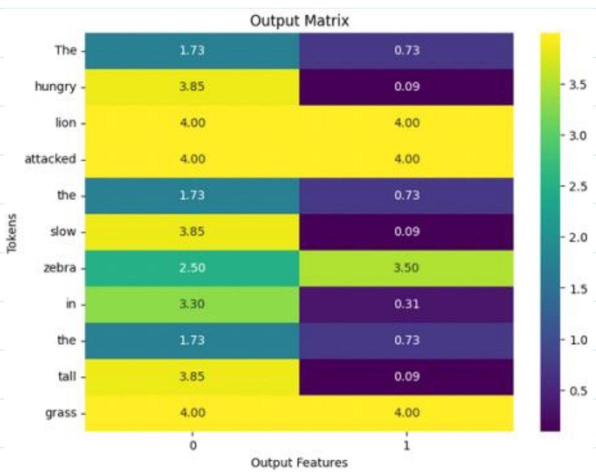
weight = softmax (score)



Attention for 'hungry':
The : 0.00
hungry : 0.01
lion : 0.88
attacked : 0.00
the : 0.00
slow : 0.01
zebra : 0.00
in : 0.00
the : 0.00
tall : 0.01
grass : 0.09

Query Token	Top Attention	Interpretation
hungry	lion (0.88)	Modifier attends to the noun it qualifies
lion	attacked (1.00)	Subject attends to its verb
attacked	zebra (1.00)	Verb focuses on its object
slow	zebra (0.88)	Adjective modifies noun
zebra	lion, zebra (0.50)	Balanced attention between actor and self
in	attacked, zebra	Preposition attends to verb and object
tall	grass (0.88)	Modifier attends to noun
grass	attacked (1.00)	Context attending back to action

Step 5 - Output = $\underset{\text{weights}}{\text{attention}}$ x Value




```
output = weights @ V
```

Token	Output Vector	Interpretation
lion	[4.00, 4.00]	Attends strongly to attacked -> inherits its [1,1] features completely (from V) - full action context
attacked	[4.00, 4.00]	Self-attended -> full value retained
zebra	[2.5, 3.5]	Mixed focus between lion ([1,0]) and zebra itself ([0,1]) makes sense as it's being acted upon
in	[3.3, 0.31]	Looks at attacked and zebra gets mostly their action/object information.
hungry	[3.85, 0.09]	Attends to lion, whose value is [1,0], scaled by the attention weight (0.88), shows modifier behavior

Now this output goes through the next layer of transformer, which we'll discuss later