Natural Language Processing with Deep Learning

Himanshu Singh

Github

Representation of words

1) One hot vector

Problem:

- Orthogonal
- Highly Sparse
- No natural inheritance meaning

2) word2vec

Language model: Assign probability to a sequence of tokens.

- Unigram model $P(w_1, w_2, w_3, \dots w_m) = P(w_1) P(w_2) \dots P(w_m)$ not good model, no reference of past or future of the language.
- Bigram model $P(w_1, w_2, w_3, \dots w_m) = \prod_{i=2}^{m} P(w_i | w_{i-1})$
- n-gram model $P(w_1, w_2, w_3, \dots w_m) = \prod_{i=i}^{m} P(w_i | w_{i-n+1} \dots w_{i-1})$

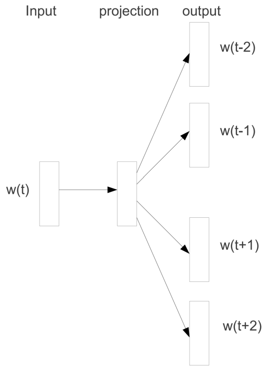
Skip-gram

the cat is eating the mouse eating here

“eating” is the centre word, “cat is” and “the mouse” are the context words.

Input takes one-hot vector $Z^{(1)} = w^{(1)} X$ //no activation function

Go through each position ‘t’ in the Corpus, which has centre word ‘c’ and context word ‘o’.



$$P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$

$$L = -\frac{1}{T} \sum_{t=1}^{T} \sum_{j=-m, j \neq 0}^{m} \log \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$

Also can be look as maximizing data likelihood.

$$L = \prod_{t=1}^{T} \prod_{j=-c, j \neq 0}^{c} P(w_{t+j} | w_t; \theta)$$

Objective function $= -\frac{1}{T} \log L(\theta)$

$$\frac{\delta}{\delta v_c} \log \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)} = \frac{\delta}{\delta v_c} \log \exp(u_o^T v_c) - \frac{\delta}{\delta v_c} \log \sum_{w \in V} \exp(u_w^T v_c)$$

$$= u_o - \sum_{w}^{V} \frac{\exp(u_w^T v_c)}{\sum_{w}^{V} \exp(u_w^T v_c)} u_w$$

$$= u_o - \sum_{w}^{V} P(w|c) u_w$$

Problem: In this function the denominator is expensive to compute

Solution:

- Sub sampling frequent words
- Context position Weighting
- Negative sampling

Sub-sampling frequent words: frequent words provide less information than rare words. To counter imbalance created by frequent words against rare, discard some frequent from the training dataset. Heuristics: w_i is the discarded with probability

$$P(w_i)=1-\left(\frac{t}{f(w_i)}\right)^{\frac{1}{2}}$$

<u>Negative sampling</u> Subsampling frequent words and context position waiting helps, but can we do better ? Noise Contrast estimation: posits a good model should be able to differentiate data from noise by means of Logistic regression.
Changing objective function for learning word vectors.
pair (w,c)coming from Data? 1 if yes, 0 if no
Maximize that all pairs come from data and minimise that does not come from data.

$$=\underset{\theta}{argmax}\prod_{(w,c)\in D}P(D=1|w,c,\theta)\prod_{(w,c)\in\tilde{D}}P(D=0|w,c,\theta)$$

$$=\underset{\theta}{argmax}\sum_{(w,c)\in D}\log\frac{1}{1+\exp(-v'^T_w v_c)}+\sum_{(w,c)\in\tilde{D}}\log\frac{1}{1+\exp(v'^T_w v_c)}$$

for negative sampling, $(w,c)\in\tilde{D}$ we can choose a ‘k’ word pair, where k ranges from 5 to 20.

$$\log\sigma(v'^T_{w_o}v_{w_I})+\sum_{i=1}^{K}\underset{w_i\sim P_n(w)}{E}[\log\sigma(-v'^T_{w_i}v_{w_I})]$$

$$J_{Negative-sampling}(u_o,v_c)=-\log\sigma(u_o^T v_c)-\sum_{k\in\text{K sampling Indices}}\log(-u_k^T v_c)$$

How to choose words from negative sampling?

$$P(w_i)=\frac{f(w_i)^{\frac{3}{4}}}{\sum_{i=1}^{V}f(w_i)^{\frac{3}{4}}}$$

Continuous Bag of Words
Predict the centre word given words in the context window. Here the input will be sum of one-hot vector of the context word.
Maximize $P(w_c|w_{c-m},w_{c-m+1},\ldots,w_{c-1},w_{c+1},\ldots w_{c+m})$
Hierarchical Softmax: Another method to improve calculation of probability of Predicting words. Binary tree representation of output layers. A random walk from the root to nodes assign probability to words. No output representation of words.

$$P(w|w_I)=\prod_{j=1}^{L(w)-1}\sigma\left([\![n(w,j+1)=ch(n(w,j))]\!].v_n'(w,j)^T v_{w_I}\right)$$

$[\![x]\!]=1$ if x = True else - 1
v_{w_I} Input representation of w_I
L(w) No. Of Nodes in the path to w
One representation of each were w, one for every inner node of the binary tree. The paper uses binary Huffman tree, short codes for frequent words.

GloVe Word-Word Co-occurence Matrix
Capture global information, simpler objectives/cost function. Ratio probabilities carry better information
$X_{ij}=$ No. of word j occurs in context of word i. $X\leftarrow|V|\times|V|dimension$
We want, $F(w_i,e_j,\tilde{w}_k)=\frac{P_{ik}}{P_{jk}}$
Finding appropriate function which help us find useful/good word embeddings. Word Embeddings are relative so absolute value of word embeddings does not matter much their difference does

$$F(w_i-w_j;\tilde{w}_k)=\frac{P_{ik}}{P_{jk}}$$

F could be complicate function like neural networks another choice dot product

$$F((w_i - w_j)^T \tilde{w}_k) = \frac{P_{ik}}{P_{jk}}$$

Reaching Symmetry

$$w_i^T \tilde{w}_k + b_i + \tilde{b}_k = \log(X_{ik})$$

w_i Input word Embedding
$\tilde{w}_k$ Context word Embedding
b_i Bias
$\tilde{b}_k$ Bias Context

$$\tilde{w}_i^T w_k + \tilde{b}_i + b_k = \log(X_{ki})$$

We want $F((w_i - w_j)^T \tilde{w}_k) = \frac{F(w_i^T \tilde{w}_k)}{F(w_j^T \tilde{w}_k)}$

To above equation, exponent function is a solution

$$\exp(w_i^T \tilde{w}_k) = \frac{X_{ik}}{X_i}$$

$$w_i^T \tilde{w}_k = \log\left(\frac{X_{ik}}{X_i}\right)$$

Cost function $J = \left(\underbrace{w_i^T \tilde{w}_k + b_i + \tilde{b}_k}_{\text{Prediction from Algo}} - \underbrace{\log(X_{ik})}_{\text{From data}}\right)^2$

Problem, if $X_{ij} = 0; \log(x_{ij})$ is not define

$$J = \sum_{i,j=1}^{|V|} f(X_{ij})\left(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log(X_{ij})\right)^2$$

Properties of F():

- F(0) = 0, it should vanish as x tends to 0; $\lim_{x \to 0} f(x) \log^2(x)$ = finite
- F(x) is non-decreasing
- F(x) relatively small for large values

$$f(x) = \left\{ \begin{matrix} (x/x_{max})^{\alpha}, \text{ if } x < x_{max} \\ 1, \text{otherise} \end{matrix} \right\}$$

Perplexity : Standard evaluation metric for Language Models

$$\text{Perplexity} = \prod_{t=1}^{T} \left[\frac{1}{P_{LM}(x^{(t+1)} | x^{(t)}, \dots, x^{(1)})} \right]^{\frac{1}{T}}$$

This is equal to exponential of cross entropy

$$\prod_{t=1}^{T} \left(\frac{1}{\hat{y}^{(t)}_{x_{t+1}}} \right)^{\frac{1}{T}} = \exp\left[\frac{-1}{T} \sum_{t=1}^{T} \log \hat{y}^{(t)}_{x_{t+1}} \right]$$

The lower perplexity, the better

Neural Network

Non-linearities:

- Sigmoid $(1+\exp(-z))^{-1}$
- tanh $\frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$
- hard tanh $\begin{cases} 1, & x\geq 1 \\ x, & -1\leq x\leq 1 \\ -1, & x\leq 1 \end{cases}$
- ReLU $max(z,0)$
- Leaky ReLU
- Swish $x.\sigma(x)$
- GeLU $x.P(X\leq x) \quad X\sim N(0,1)$ $\approx x.\sigma(1.702x)$

Regularization

Classical view: Regularization works to prevent over-fitting when we have a lot of features.

Now: Regularization produces model that generalize well when we have a “big” model we don't care that our model overfits on the training data even though they are hugely overfit.

DropOut: It prevents feature co-adaptation = good regulation

Parameter Initialization

- Zero Initialization (does not work) leads to identical gradient and prevents neuron from learning diverse features
- Uniform Initialization can work but effectiveness depends upon architecture and problem. The choice of range is critical for training. If it is too small, gradients during back-propagation may vanish.
- Xavier Initialization: Variance inversely proportional to fan in(n_{in}) and fan out (n_{out})

$$var(w)=\frac{2}{n_{\text{in}}+n_{out}}$$

Optimizer

- SGD (Works just fine)
- Adagrad (Simplest, but tend to “stall early”)
- RMSprop
- Adam (Fairly good, safe place to begin in many cases)
- AdamW
- NadamW

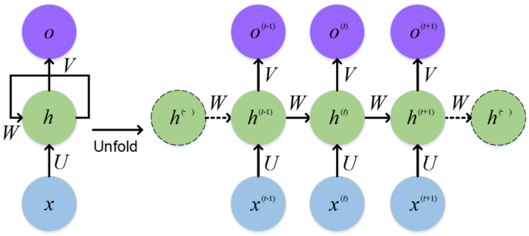
Recurrent Neural Network (RNN)

$$h^{(t)}=\sigma(W h^{(t-1)}+U x^{(t)}+b_1)$$

$$o^{(t)}=V h^{(t)}+b_2$$

Advantages:

- Can process any length input
- Model size does not increase for longer context
- In theory, computation for step t, can use information from many step back
- Same weights applied on every time step



Disadvantages:

- Computation is slow.

Loss function on step t

$$J^{(t)}(\theta)=-\sum_{w\in V} y_w^{(t)}\log y_w^{(t)}=-\log \hat{y}_{x_{t+1}}^{(t)}$$

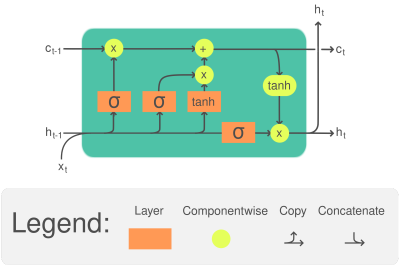
$$J(\theta)=\frac{1}{T}\sum_{t=1}^{T} J^{(t)}(\theta)=-\frac{1}{T}\sum_{t=1}^{T}\hat{y}^{(t)}_{x_{t+1}}$$

RNNs can greatly improve perplexity over what came before.

<u>Problems with RNN</u>

Vanishing Gradient Problem: Gradient signal from far away is lost because its much smaller than gradient signal from close by. So model weights are basically updated only wrt near effects, not long term effects. Exploding gradient problem can be fixed by gradient clipping but vanishing gradient is still a problem. (Difficult to preserve information over many time steps)

<u>Long Short Term Memory (LSTM)</u>



$$f^{(t)}=\sigma\left(W_f h^{(t-1)}+U_f x^{(t)}+b_f\right)$$
$$i^{(t)}=\sigma\left(W_i h^{(t-1)}+U_i x^{(t)}+b_i\right)$$
$$o^{(t)}=\sigma\left(W_o h^{(t-1)}+U_o x^{(t)}+b_o\right)$$

Forget Gate: Control what is kept vs forgotten from previous cell state.

Input Gate: Control what part of new cell content are written to cell

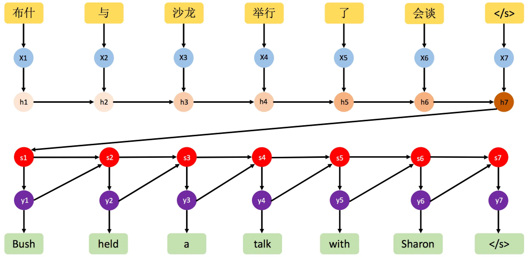
Output Gate: Control what part of cell are output to hidden state.

$$\tilde{c}^{(t)}=\tanh\left(W_c h^{(t-1)}+U_c x^{(t)}+b_c\right)$$
$$c^{(t)}=f^{(t)}\odot c^{(t-1)}+i^{(t)}\odot\tilde{c}^{(t)}$$
$$h^{(t)}=o^{(t)}\odot\tanh c^{(t)}$$

LSTMs solve vanishing Gradients?

LSTM architecture make it much easier for an RNN to preserve information over many timesteps. If forgot gate is set to 1 for a cell and input gate is set to 0, then all the information of that cell is preserve indefinitely. In contrast its harder for Vanilla RNN and to learn current weight Matrix W_h that preserve info in hidden state.

Multi layer or stacked RNN allow a network to compute more complex representation, they work better than just have one layer of hidden dimensional encodings.



(Sutskever et al., 2014)

Neural Machine Translation e2e NN
Architecture seq2seq (involves to RNN)

How do we evaluate machine translation?
BLEU (Bilingual Evaluation Understudy)
BLEU compare machine written translation to one or several human return translations and compare a similar score based on:
1) Geometric mean of n-gram precision (1-, 2-, 3-, 4- gram)
2) Plus a penalty for two short system translation
BLEU is useful but imperfect, many ways to translate a sentence, a good translation can get a poor BLEU because it has low n-gram overlap with human evaluation.

Attention

Core idea: On each step of decoder, use a direct connection to the encoder to focus on a particular part of source sequence

$h_1, h_2, \ldots, h_N \in \Re^n$ //Encoder Hidden state

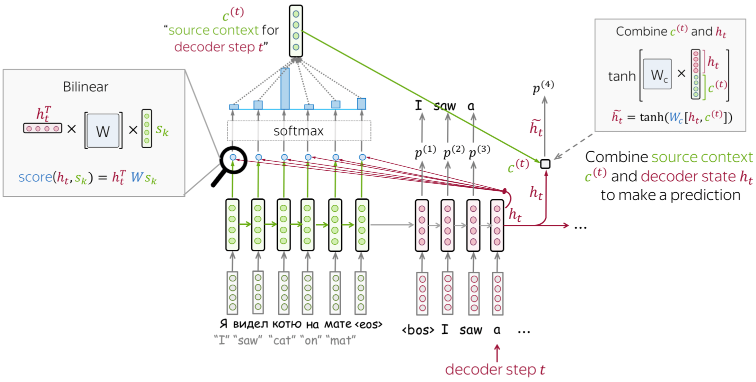
$s_t \in \Re^n$ //Decoder hidden state at step t

We get attention score

$$e^t=[s_t^T h_w,\dots,s_t^T h_N]\in\Re^N$$
$$\alpha^t=\text{softmax}(e^t)\in\Re^N$$
$$a_t=\sum_{i=1}^{N}\alpha_i^t h_i\in\Re^n$$

We use α^t to take a weighted sum of the encoder hidden state to get attention output a_t. We concatenate attention output a_t with decoder hidden state s_t and proceed as in non-attention seq2seq models

$$[a_t;s_t]\in\Re^{2n}$$



$c^{(t)}$
"source context for decoder step t"
Bilinear
h_t^T
×
W
×
s_k
score$(h_t, s_k) = h_t^T\, W s_k$
softmax
I saw a
$p^{(1)}$ $p^{(2)}$ $p^{(3)}$
$p^{(4)}$
$\tilde{h}_t$
$c^{(t)}$
h_t
h_t
...
Combine $c^{(t)}$ and h_t
tanh
W$_c$
×
h_t
$c^{(t)}$
$\tilde{h}_t$ = tanh$(W_c[h_t, c^{(t)}])$
Combine source context $c^{(t)}$ and decoder state h_t to make a prediction
Я видел котю на мате <eos>
"I" "saw" "cat" "on" "mat"
<bos> I saw a ...
decoder step t

Attention Variants:

Several ways to compute $e\in\Re^N$ from $h_1,\dots,h_N\in\Re^{d_1}$ and $s\in\Re^{d_2}$

1) Basic dot product: Assume $d_1=d_2::e_i=s^T h_i\in\Re$

2) Multiplicative attention (better called as Bi-linear attention) $e_i=s^T W h_i\in\Re\,;W\in\Re^{d_2\times d_1}$ is weight matrix.

3) Reduce-Rank multiplicative attention $e_i=s^T(U^T V)h_i=(Us)^T(Vh_i)::U\in\Re^{k\times d_2}V\in\Re^{k\times d_1}k\ll d_1,d_2$

4) Additive attention $e_i=V^T\tanh(W_1 h_1+W_2 s)::W_1\in\Re^{d_3\times d_1}W_2\in\Re^{d_3\times d_2}V\in\Re^{d_3}$

More general definition of attention: Given a set of factor values and query. Attention is a technique to compute a weighted sum of the values, dependent on the query

Issues with RNN/LSTM:

1) Linear Interactive Distance: Nearby words often affect each others meaning. O(seq_length) steps for distant word pair to interact. Which means hard to lean long-distance dependencies

2) Lack of parallelizability: Forward and backward passes have O(seq_length) unparallelizable operations but future RNN hidden states can't be computed in full before past RNN hidden states have been computed.

You can think of attention as performing fuzzy lookup in a key-value store. In a lookup table we have table of keys that map to a value. The query matches one of the keys returning its value. In attention the query matches all key softly to a weight between 0 and 1, the keys values are multiplied by weights and summed.

Self-attention

$W_{1:n}$ be the sequence of word in vocabulary V. For each word w_i, let $x_i=Ew_i;E\in\Re^{d\times|V|}$ Embedding Matrix.

1) Transform each word embedding with weight matrix Q, K, V each in $\Re^{d\times d}$

2) Compute pairwise-similarities between keys and queries, normalize with softmax

$$e_{ij}=q_i^T k_j$$
$$\alpha_{ij}=\frac{\exp(e_{ij})}{\sum_l \exp(e_{il})}$$

3) computer output for each word as weighted sum of values

$$o_i=\sum_j \alpha_{ij}v_j$$

Fixing first self attention problem:

1) Sequence order: since self attention does not built in order information we need to encode the order of the sentence in our keys queries and value. Considered representing each sequence index as a vector $p_i\in\Re^d\quad i\in\{1,2,..,n\}\quad \tilde{x}_i=x_i+p_i$

<u>Sinusoidal position representation</u>

$$p_i=\begin{bmatrix} \sin\left(i/10000^{\frac{2*1}{d}}\right) \\ \cos\left(i/10000^{\frac{2*1}{d}}\right) \\ \vdots \\ \sin\left(i/10000^{\frac{2*d/2}{d}}\right) \\ \cos\left(i/10000^{\frac{2*d/2}{d}}\right) \end{bmatrix}$$

Pros:

- Priodicity indicates that maybe “absolute position” isn't as important.
- maybe we can extrapolate to longer sequence as periods restart!

Cons:

- not learnable; extrapotion does not really work!

Learned absolute positional representation

Let all p_i be learnable parameters. $p\in\Re^{d\times n}$ p_i: column of p

Pros:

- Flexibility: each position gets to be learnt to fit the data.

Cons:

- Definitely can't extrapolate the indices outside 1 ... n

2) Adding non-linearities in Self Attention, since there are no non-linearity in self attention adding MLP layer post self attention $m_i=MLP(output)=W_2 ReLU(W_1 output_i+b_1)+b_2$

We can stack self attention + MLP on top of each other

3) Masking future in self attention: To enable parallelisation for future words by setting attention score to -∞

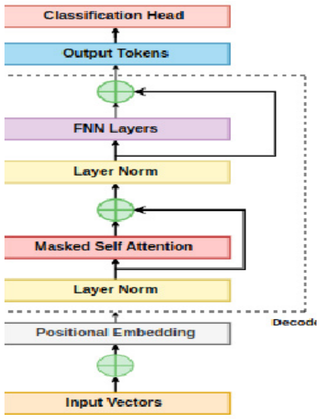
Transformers

Decoder-only Transformers: Replace self attention with Masked multi-head attention.

Sequence stack from attention $X=[x_1,x_2,\ldots,x_n]\in\Re^{n\times d}$ be concatenation of input vectors.

$XK\in\Re^{n\times d}$ $XQ\in\Re^{n\times d}$ $XV\in\Re^{n\times d}$.

The output is defined as $softmax(XQ(XK)^T)XV\in\Re^{n\times d}$



Multihead Attention

For token ‘i’, self attention looks where $X_i Q^T K X_j$ is high, but maybe we ant to focus on different j for different reasons.

$Q_l, K_l, V_l\in\Re^{n\times\frac{d}{h}}$ where ‘h’ is number of heads, $l\in\{1,2,\ldots,h\}$

$output_l=softmax(XQ_l K_l^T X^T)XV_l$. Output of all heads are combined

$output=[output_1, output_2,\ldots,output_h]Y, Y\in\Re^{d\times d}$

Even though we compute ‘h’ many attention had it is not really more costly. Compute $XQ\in\Re^{n\times d}$, reshape to $\Re^{n\times h\times\frac{d}{h}}$ likewise for XV and XK. Transpose to $\Re^{h\times n\times\frac{d}{h}}$ (head axis is like batch axis). Everything else identical and matrices are of same size.

Scaled dot product

When dimensionality ‘d’ becomes large, dot product between vectors tend to become large, because of this, input to softmax can be large making gradient small. We divide attention score by $\sqrt{d/h}$ to stop the score from being large.

$$output_l=softmax\left(\frac{XQ_l K_l^T X^T}{\sqrt{d/h}}\right)XV_l$$

Two optimization tricks:

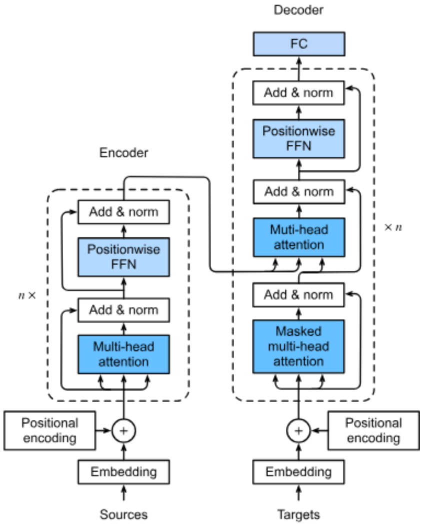
- Residual connection

- Layer normalization

These are often written as “add and norm”

Layer normalization Cut down on uninformative variation in hidden vector values by normalizing to unit mean and standard deviation within each layer

Encoder-decoder Transformers



We use normal transformer encoder but Decoder modified to perform cross attention to the output of the encoder. Let $h_, h_2, \ldots h_n$ be output vector from Transformer encoder. Let $z_1, z_2, \ldots, z_n$ be input vector from Transformer decoder, then keys and values are drawn from encoder $k_i = Kh_i \quad v_i = Vh_i$. Queries are drawn from decoder

Drawbacks of Transformers:

- Quadratic compute in self attention : Computing all interactions means are computation grows quadratic with sequence length, for RNN/LSTM it goes linearly
- Positional representation

Efficient Transformers

N : Sequence length

h : No. of heads

d_{model}: Embedding size

$d_k \quad : d_{model}/h$

$dim(X) = N \times d_{model}$

$dim(W^Q) = dim(W^K) = dim(W^V) = d_{model} \times d_k$

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d}}\right)V$$

$dim(QK) = N^2 O(N^2 d)$ Time comlexity to calculate QK

Storing Attention matrix for each head $O(N^2 h)$

If $N \gg d_k$, space and time complexity $O(N^2)$

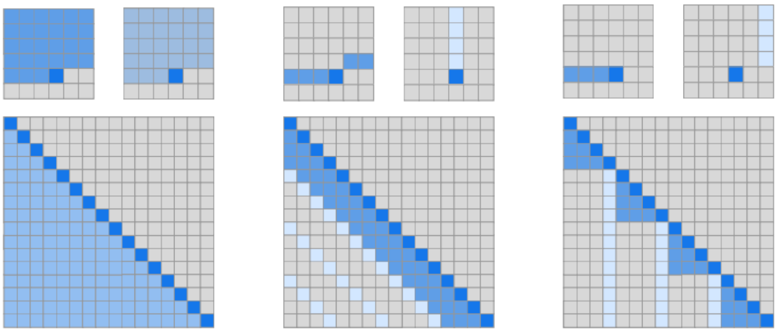
Sparse Transformers

$S = S_1, S_2, \ldots, S_n; S_i$: Input vector that i^{th} vector attends to

$Attend(X, S)\{a(x_i, S_i)\}_{i \in \{1,2,..n\}}$

$$a(x_i, S_i) = softmax\left(\frac{(W^Q x_i) K_{S_i}^T}{\sqrt{d}}\right) V_{S_i}$$

$$\left\{\begin{matrix} K_{S_i} = (W^K S_j)_{j \in S_i} \\ V_{S_i} = (W^V S_j)_{j \in S_i} \end{matrix}\right\}$$



(a) Transformer (b) Sparse Transformer (strided) (c) Sparse Transformer (fixed)

There can be ‘P’ separate attention heads, attending to different subsets of input tokens. The paper chose $|S_i^m| \propto \sqrt[p]{n}$

1) Strided Attention : Each tokens attends to every k^th token.

$S_i^1 = \{t, t+1, \ldots i \text{ for } t = max(0, i-l)\}$

$S_i^2 = \{j : (i-j) mod\, l = 0\}$

2) Fixed Attention

$$S_i^1 = \left\{j : \left\lfloor \frac{j}{l} \right\rfloor = \left\lfloor \frac{i}{l} \right\rfloor\right\}$$

$S_i^2 = \{j : j mod\, l\, l \in \{t, t+1, \ldots, l\}\}$

Time complexity reduces from O(N^{2}d) to O(Nkd)

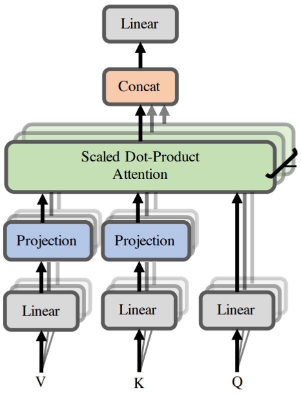
Linformer

Project K_i and V_i with E_i and $F_i \in \Re^{k\times n}$ into $\hat{k}_i = E_i K_i, \hat{v}_i = F_i V_i \in \Re^{k\times d_k}$

Softmax requires $O(Nk)$ computational cost

Dimension of Attention matrix $N\times k$. Hence Storage $O(Nkh)$ which is linear in N$(k, h \ll N)$

Theorem: For any $Q_i, K_i, V_i \in \Re^{n\times d}$ and $W_i^Q, W_i^K, W_i^V \in \Re^{d\times d}$, if $k = min\{\Theta(9d\log(d))/\epsilon^2\}$, then *exist* matrices E_i and $F_i \in \Re^{n\times k}$ such that for any row vector w of matrix $QW_i^Q(KW_i^K)^T/\sqrt{d}$, *we have*



$$Pr(\|softamx(wE_i^T)F_i V W_i^V - softmax(w) V W_i^V\| < \epsilon \|softmax(w)\| \|V W_i^V\|) > 1 - o(1)$$

Linearized Attention

$$Y_i = softmax\left(\frac{Q_i K}{\sqrt{d_k}}\right) V$$

$$Y_i = \frac{\exp(\frac{Q_i K}{\sqrt{d_k}}) V}{\sum_{j=1}^{N} \exp(\frac{Q_i K_j}{\sqrt{d_k}})}$$

$$Y_i = \frac{\sum_{j=1}^{N} \text{sim}(Q_i, K_j) V_j}{\sum_{j=1}^{N} \text{sim}(Q_i, K_j)}$$

where $\text{sim}(q, k) = \exp(\frac{q.k}{\sqrt{d_k}})$

The only constraint on sim(.) is no negativity, Any $K(x, y): \Re^{2\times d_{model}} \rightarrow \Re_+$ works

Let $\phi(x)$ be feature representation of such a kernel, then

$$Y_i = \frac{\sum_{j=1}^{N} \phi(Q_i)^T \phi(K_j) V_j}{\sum_{j=1}^{N} \phi(Q_i)^T \phi(K_j)}$$

$$Y_i = \frac{\phi(Q_i) \sum_{j=1}^{N} \phi(K_j) V_j^T}{\phi(Q_i)^T \sum_{j=1}^{N} \phi(K_j)}$$

Now, $\sum_{j=1}^{N} \phi(K_j) V_j^T$ and $\sum_{j=1}^{N} \phi(K_j)$ can be computed once and reused.

Causal Masking : Maskign Attention computation such that i^th^ position can only be influenced by position j, if and only if $j \le i$

$$Y_i = \frac{\phi(Q_i) \sum_{j=1}^{i} \phi(K_j) V_j^T}{\phi(Q_i)^T \sum_{j=1}^{i} \phi(K_j)} = \frac{\phi(Q_i)^T S_i}{\phi(Q_i)^T Z_i}$$

$$S_i = \sum_{j=1}^{i} \phi(K_j) V_j^T$$
$$Z_i = \sum_{j=1}^{i} \phi(K_j)$$

Byte-Pair encoding algorithm

- Start with vocabulary containing only characters and an “and of word” symbol
- Using Corpus of text find the most common adjacent character
- Replace instances of character pair with new subwords, repeat until desired vocabulary size

Subword Modelling paradigm is to learn vocabulary of part of words. At training and testing time each word is split into a sequence known as subword

Earlier we had pre-trained word embeddings on which a model (RNN/LSTM/Transformer) has been trained. Issues :

- Training data from downstream task must be sufficient to teach all contextual aspect of language.
- Parameters in our network are randomly initialise.

In modern NLP, all parameters are initialized via pre training pre-training. Pre-training method hide parts of input from the model and train the model to reconstruct these part. This has been exceptionally efficient at building strong:

- Representation of language
- Parameter initialization for strong NLP model
- Probability distribution over language that we can sample from

Why should pre-training and fine-tuning help from “training Network network” perspective

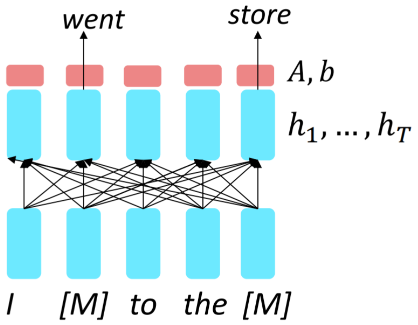
- Provide $\hat{\theta}$ by approximating $\min_{\theta} \mathcal{L}_{pretrain}(\theta)$
- Then fine-tune by approximating $\min_{\theta} \mathcal{L}_{finetuning}(\theta)$, starting at $\hat{\theta}$

The pre-training may matter because SGD stick close to $\hat{\theta}$ during fine-tuning. So maybe fine-tuning local minimum near $\hat{\theta}$ tend to generalize well and/or maybe gradient of fine-tuning loss near $\hat{\theta}$ propagate nicely.

Pre-training Encoders

Encoder gets bi-directional context, so we can’t do language modelling! Replace some fraction of words in input with special mask [MASK] token predict these words.

$$h_1, h_2, \ldots, h_T = Encoder(w_1, w_2, \ldots, w_T)$$
$$y_i \sim Ah_i + b$$



BERT Bi-directional Encoder Representation from Transformer

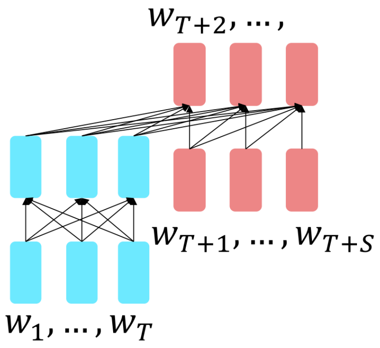
Mask LM for BERT, predict random 15% of (sub)words tokens

- Replace input word with [MASK] 80% of the time
- Replace input word with a random token 10% of the time
- Leave input word unchanged 10% of the time (but still predict it)

If you are task involve generating sequences, consider using decoder BERT and other don't naturally lead to nice auto-regressive generation method.

Pre training Encoder-Decoder

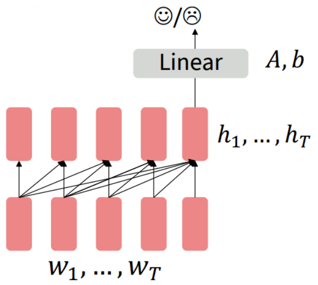
The encoder model benefit from bi-directional context. Decoder portion is used to train the whole network through language modelling.



Pre-training Decoder

When using language model, pre-trained decoded we can ignore that they were trained to model $P(w_t|w_{1:t-1})$. We can fine-tune them by training a classifier on the last word hidden state.

$$h_1, h_2, \dots, h_T = Decoder(w_1, w_2, \dots, w_T)$$
$$y \sim Ah_T + b$$



t is natural to pre-train Decoder as a language model then use them as generator, fine-tune with $p_\theta(w_t|w_{1:t-1})$. This will helpful in task where output is a sequence with vocabulary like that at pre-training time

- Dialogue (context = Dialogue history)
- Summarization (context = document)

GPT Generative pre-train Transformers

- 12 layers, 117M parameters
- 768 dimensional hidden state, 3072 dimensional feed forward layer
- Byte-pair Encoding with 40000 mergers
- Trained on book Corpus (7000 books)

GPT-2 (1.5B) trained on more data

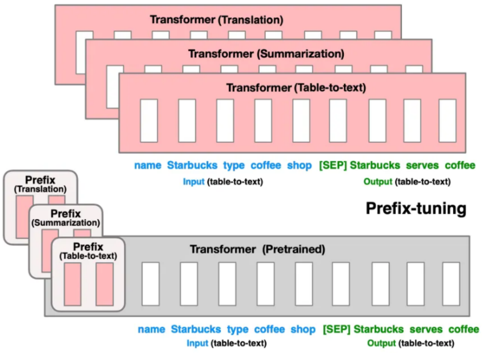
GPT-3 (175B) seems to perform some kind of learning without grading steps simply from examples you provide within the contexts. The incontext example seems to specify the task to be performed and the conditional distribution mocks performing the task to certain extent

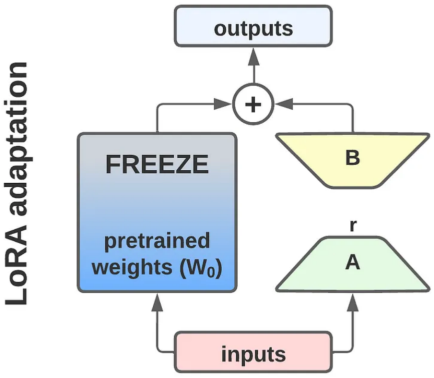
Fine-tuning vs Parameter Efficient Fine-Tuning

Fine-Tuning every parameter on a pre-trained model works well, but it's memory intensive. But light-weight fine-tuning methods adapts pre-trained model in a constrained way, leads to less over-fitting and/or more efficient find tuning and inference

Parameter Efficient Fine-Tuning

1) Prefix tuning adds a prefix to parameters and freeze all the pre-trained parameters. The prefix is processed by the model just like real words would be. Advantage: each element of the batch at inference could run a different tune model.


Transformer (Translation)
Transformer (Summarization)
Transformer (Table-to-text)
name Starbucks type coffee shop [SEP] Starbucks serves coffee
Input (table-to-text)
Output (table-to-text)
Prefix-tuning
Prefix (Translation)
Prefix (Summarization)
Prefix (Table-to-text)
Transformer (Pretrained)
name Starbucks type coffee shop [SEP] Starbucks serves coffee
Input (table-to-text)
Output (table-to-text)



2) Low Rank Adaptation : Learns a low rank difference between pre trained and fine-tuned weight matrices

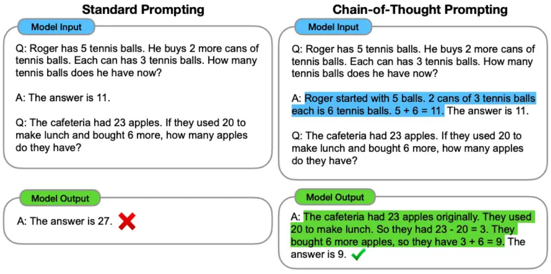
Post Training

Language model made to rudimentary modelling of agents, believes an actions

1) Emergent zero short learning: One key emergent ability in GPT-2 is zero short learning, ability to do many task with no example and no gradient updates.

2) Emergent few short learning: Specify a task by simply prepending examples of the task before your examples. Also called in-context learning to stress that gradient up dates are performed when learning and new task.

3) Chain of thought prompting



Standard Prompting

Model Input

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: The answer is 11.

Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

Model Output

A: The answer is 27. ✗

Chain-of-Thought Prompting

Model Input

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. 5 + 6 = 11. The answer is 11.

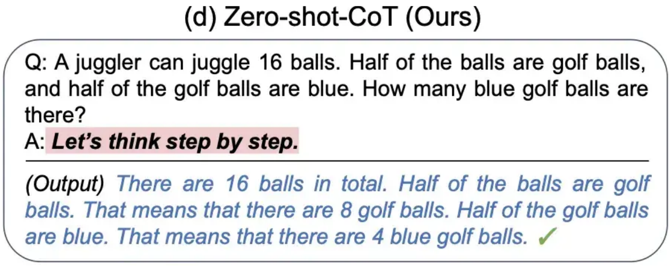
Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

Model Output

A: The cafeteria had 23 apples originally. They used 20 to make lunch. So they had 23 - 20 = 3. They bought 6 more apples, so they have 3 + 6 = 9. The answer is 9. ✓

4)Zero-shot chain of tought prompting
Zero-shot and few-shot in-context learning:

- No fine tuning needed, prompt engineering (CoT) can improve performance.
- Limits to what you can fit in context.
- Complex task will probably need gradient steps.

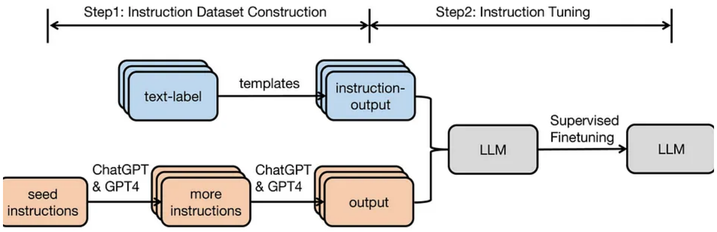


(d) Zero-shot-CoT (Ours)

Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there?
A: ***Let's think step by step.***

(Output) *There are 16 balls in total. Half of the balls are golf balls. That means that there are 8 golf balls. Half of the golf balls are blue. That means that there are 4 blue golf balls.* ✓

Instruction fine tuning: Supervised fine-tuning trains models on input examples and their corresponding outputs, instruction tuning augments input-output examples with instructions, which enables instruction-tuned models to generalize more easily to new tasks.
1) You can generate data synthetically from bigger LLMs
2) You don't need many samples to instruction tune.
3) Crowdsourcing can be pretty effective



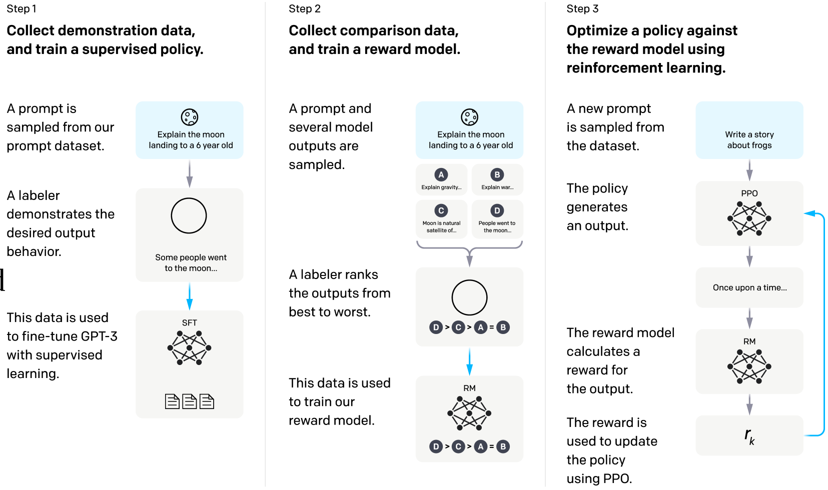
Limitation of instruction fine tuning:

- It is expensive to collect ground truth data for tasks.
- Tasks like open ended creative generation have no right answer.
- Language modelling penalize all the token level mistake equally but some errors are worse than other.
- Human generate sub optical answers.

Optimising for human preferences
We are training LM on some task, for an instruction 'x' and the LM sample 'y'. we have a better way to obtain a human reward of summary $R(x, y) \in \Re$ (higher is better).
Maximize expected reward of samples from our LM

$$\underset{\hat{y} \sim p_\theta(y|x)}{E} [R(x, \hat{y})]$$

High level instantiation 'RLHF' pipeline
Problems:
1) Humans in the loops is expensive: Instead of directly asking humans for preferences, model they are preferences as a separate NLP problem. Train $RM_\phi(x, y)$ to predict human reward from annotated data set, than optimise for RM_ϕinstead.
2) Human judgement are noisy and miss calibrated: Instead of asking for direct ratings, ask for pair-wise comparison with can be more reliable.



Step 1
Collect demonstration data, and train a supervised policy.
A prompt is sampled from our prompt dataset.
A labeler demonstrates the desired output behavior.
This data is used to fine-tune GPT-3 with supervised learning.

Step 2
Collect comparison data, and train a reward model.
A prompt and several model outputs are sampled.
A labeler ranks the outputs from best to worst.
This data is used to train our reward model.

Step 3
Optimize a policy against the reward model using reinforcement learning.
A new prompt is sampled from the dataset.
The policy generates an output.
The reward model calculates a reward for the output.
The reward is used to update the policy using PPO.

Explain the moon landing to a 6 year old
Some people went to the moon...
SFT
Explain the moon landing to a 6 year old
A
Explain gravity...
B
Explain war...
C
Moon is natural satellite of...
D
People went to the moon...
D > C > A = B
RM
D > C > A = B
Write a story about frogs
PPO
Once upon a time...
RM
r_k

$$J_{RM}(\phi) = - \underset{(x, y^w, y^l) \sim D}{E} \left[\log \sigma \left(RM_\phi(x, y^w) - RM_\phi(x, y^l) \right) \right]$$

$$y^w : \text{Winning Sample}$$
$$y^l : \text{Losing Sample}$$

RLHF: Optimizing learned reward model
We have the following:

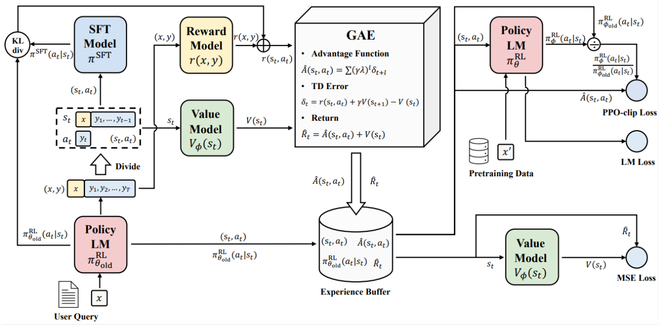
1) Pretrand model LM $p^{PT}(y|x)$
2) Reward model $RM_\phi(x,y)$, produces scalr reward for LM outputs
Now to do RLHF, copy model $p_\theta^{RL}(y|x)$ with parameters θ we want to optimise

$$\underset{\hat{y}\sim p^{RL}(\hat{y}|x)}{E}[RM_\phi(x,\hat{y})]$$

Problem: learned reward model are imperfect, this quantity can be imperfectly optimised. Add penalty for drifting too far from the any initialization.

$$\underset{\hat{y}\sim p_\theta^{RL}(\hat{y}|x)}{E}[RM_\phi(x,\hat{y})-\beta\log\left(\frac{p_\theta^{RL}(\hat{y}|x)}{p^{PT}(\hat{y}|x)}\right)]$$

RLHF can be Complex computationally expensive and tricky:
1) Fitting on value function
2) Online sampling is slow
3) Performance sensitive to Hyper-parameter


KL div
$\pi^{SFT}(a_t|s_t)$
SFT Model π^{SFT}
(x,y)
Reward Model $r(x,y)$
$r(x,y)$
$r(s_t,a_t)$
GAE
Advantage Function
$\hat{A}(s_t,a_t)=\sum(\gamma\lambda)^l\delta_{t+l}$
TD Error
$\delta_t=r(s_t,a_t)+\gamma V(s_{t+1})-V(s_t)$
Return
$\hat{R}_t=\hat{A}(s_t,a_t)+V(s_t)$
(s_t,a_t)
s_t x $y_1,\dots,y_{t-1}$
a_t y_t (s_t,a_t)
s_t
Value Model $V_\phi(s_t)$
$V(s_t)$
Divide
(x,y) x $y_1,y_2,\dots,y_T$
$\pi_{\theta_{old}}^{RL}(a_t|s_t)$
Policy LM $\pi_{\theta_{old}}^{RL}$
(s_t,a_t)
$\pi_{\theta_{old}}^{RL}(a_t|s_t)$
x
User Query
$\hat{A}(s_t,a_t)$ $\hat{R}_t$
(s_t,a_t) $\hat{A}(s_t,a_t)$
$\pi_{\theta_{old}}^{RL}(a_t|s_t)$ $\hat{R}_t$
Experience Buffer
(s_t,a_t)
Policy LM π_θ^{RL}
$\pi_\phi^{RL}(a_t|s_t)$
$\pi_{\phi_{old}}^{RL}(a_t|s_t)$
$\frac{\pi_\phi^{RL}(a_t|s_t)}{\pi_{\phi_{old}}^{RL}(a_t|s_t)}$
$\hat{A}(s_t,a_t)$
PPO-clip Loss
x'
Pretraining Data
LM Loss
$\hat{R}_t$
s_t
Value Model $V_\phi(s_t)$
$V(s_t)$
MSE Loss

Direct preference Optimisation(DPO)

Derive $RM_\phi(x,y)$ in terms of $p_\theta^{RL}(\hat{y}|x)$ and then optimise parameter θ by fitting $RM_\theta(x,y)$ to preference data instead of $RM_\phi(x,y)$.
We want to maximize

$$\underset{\hat{y}\sim p_\theta^{RL}(\hat{y}|x)}{E}[RM_\phi(x,\hat{y})-\beta\log\left(\frac{p_\theta^{RL}(\hat{y}|x)}{p^{PT}(\hat{y}|x)}\right)]$$

This has a closed from solution

$$p^*(y|x)=\frac{1}{Z}(x)\,p^{PT}(\hat{y}|x)\exp\left(\frac{1}{\beta}RM(x,\hat{y})\right)$$

$$\Rightarrow RM(x,\hat{y})=\beta\log\frac{p^*(\hat{y}|x)}{p^{PT}(\hat{y}|x)}+\beta\log Z(x)$$

Hold true for arbitrary LMs

$$RM_\theta(x,\hat{y})=\beta\log\frac{p_\theta^{RL}(\hat{y}|x)}{p^{PT}(\hat{y}|x)}+\beta\log Z(x)$$

Reward model

$$J_{RM}(\phi)=-\underset{(x,y^w,y^l)\sim D}{E}\left[\log\sigma(RM_\phi(x,y^w)-RM_\phi(x,y^l))\right]$$

We only need difference between reward for y^w and y^l

$$RM_\phi(x,y^w)-RM_\phi(x,y^l)=\beta\log\frac{p_\theta^{RL}(y^w|x)}{p^{PT}(y^w|x)}-\beta\log\frac{p_\theta^{RL}(y^l|x)}{p^{PT}(y^l|x)}$$

Partition function cancels out. We have a simple classification loss function that connects preference data to language model parameters directly
Limitation of RL + reward model:

- Human preferences are unreliable.
- Reward hacking is a common problem in RL.
- Chatbots are rewarded to produce responses that seems authoritative and helpful regardless of truth.

Natural language generation (NLG)

NLG focuses on system that produces fluent coherent and useful language output for human consumption. Uses:

- Machine Translation
- Digital assistant

- Summarization system
- Creative story writing
- Data-to-text
- Visual description

Categorization of NLG task:
1) Open-ended generation: output distribution has high freedom.
2) Non open-ended generation: input mostly determine output generation.
One way of formalizing categorization is by entropy: For non open -ended tasks (e.g. Machine Translation) typically uses and encoder decoder system where this auto regressive model serve as a decoder, we have another bi-directional encoder for encoding the input. For open-ended tasks the autoregressive model is often the only component (e.g. story generation)

Trained to maximize probability of next token y_t^*given preceding tokens $\{y^*\}_{<t}$

$$\mathscr{L}=-\sum_{t=1}^{T}\log P(y_t^*|\{y^*\}_{<t})$$

At inference time, our decoding algorithm defines function to select token from this distribution. $\hat{y}_t=g(P(y_t|\{y_{<t}\}))$. “Obvious” decoding algorithm is to greedily choose the highest probability. While the basic algorithm works, to do better the two main avenues are:

- Improve decoding
- Improve the training

Overall, maximum probability decoding is a good for low entropy task like machine translation and summarization. The most likely string is repetitive for non open-ended generation, the more repeat we have the more confident model become for these repeats even scale does not solve this problem.
How do we stop repetition ?
Simple option: Don't repeat n-gram
More complex:
1) Use different training objective:

- Unlikelihood objective penalise generation of already seen tokens
- Coverage laws prevents attention mechanism from attending to same word

2) Use different decoding objective:

- Contrastive decoding searches for string, that maximize $\log\ \text{prob\_largeLM}(x)-\log \text{prob\_smallLM}(x)$

<u>Top-k sampling</u>: Vanilla samplings makes every token in vocabulary an option, which is a problem. Many tokens are probably wrong in current context. The tail of the distribution could be very long and in aggregate have considerable mass. Solution is to sample from top-k tokens in Probability distribution. Increasing ‘k’ yields more diverse but risky output, decreasing ‘k’ yields more safe but generic outputs.
<u>Top-p (nucleus) sampling</u>: The probability distribution be sample from our dynamic. When distribution P_t is flatter, a limited k removes many vaiable options, when distribution P_t is peakier, a high k allows for too many options to have a chance of being selected.
Solution: Sample all tokens in top-p community probability mass.
<u>Temperature</u> we can apply a temperature hyper-parameter $\tau$to the softmax to rebalance P_t

$$P_t(y_t=w)=\frac{\exp(S_w/\tau)}{\sum_{w'\in V}\exp(S_{w'}/\tau)}$$

Raise $\tau>1$, P_t becomes more uniform : more diverse output
Lower $\tau<1$, P_t becomes more spiky : less diverse output.
Improving decoding: Re-ranking: When we decode bad sequence from a model, decode bunch of sequence. Define a score, an approximate quality of sentence and re-rank by the score.

- Perplexity
- Re-rankers can score variety of properties: style, discourse, entailment, factuality.
- Compose multiple re-ranker together

Model training

MLE models learns bad mode of text distribution. Training with teacher forcing leads to exposure bias and generation time. During training, our model's input are gold context tokens from the real human generated text.

$$\mathscr{L}_{MLE} = -\log P(y_t^* | \{y^*\}_{<t})$$

At generation time are model inputs are previously decoded tokens.

$$\mathscr{L}_{dec} = -\log P(\hat{y}_t | \{\hat{y}\}_{<t})$$

Exposure Bias solution:

1) Schedule sampling: With the probability p. Decode a token and feed that as the next input rather than gold token. Increase p over the course of training.

2) Dataset Aggregation (Dagger): At various interval during training, generate sequences from your current model. Add these sequences to your training set as additional examples.

3) Retrieval Augmentation: Learn to retrieve tokens from existing corpus of human written prototypes, learn to edit the retrieved sequences by adding, removing and modifying tokens in the prototype - this will result in more "human like" generation.

4) Reinforcement learning: Cast your text generation as Markov Decision Process

- State(s): model representation of preceding context.
- Action(a): words that can be generated
- Policy(π) as decoder
- Reward(r) provided by external score

Reward estimation:

- BLEU (machine translation)
- ROUGE (summarization)
- CIDEr (Image captioning)
- SPIDEr (Image captioning)
- Human preference RLHF

Be careful about optimising for the task as opposed to "gaming" the reward. We can also tie behaviour to rewards:

- Cross modality consistency
- Sentence simplicity
- Temporal consistency
- Formality
- Utterance Politeness

Types of Evaluation method for tax generation:

- Content overlap Metrics
- Model based metrics
- Human Evaluation

1) Content overlap Metrics: Compute a score that indicates the lexical similarity between generated and gold standard text.

- Fast and efficient and widely used
- N-gram overlap matrix (BLUE, ROUGE, METEOR, CIDEr)

They are not ideal for machine translation. They get progressively much worse for task that are more open-ended then machine translation.

2) Model based Metrics: Use learned representation of words and sentences to compute semantic similarity between generated and reference text. The embeddings are pretrained, distance metrics used to measure similarity can be fixed.

MAUVE score: Similarity between generated text and human text. Score is calculated using KL divergence between the text distribution in a quantized embedding space of LLM.
3) Human Evaluation: evaluate quality of generated text along some specified dimension:

- Fluency
- Coherence/Consistency
- Factuality and Correctness
- Common sense
- Style/Formality
- Grammatically
- Redundancy

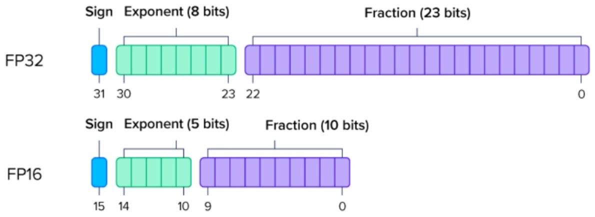
Efficient Neural Network Training

Mixed Precision Training

$$(-1)^S \times 2^{E-Bias} \times (1+M)$$

FP16 has less range 10^{-5} tot 10^4 Smaller precision leads to rounding error, for neural networks :

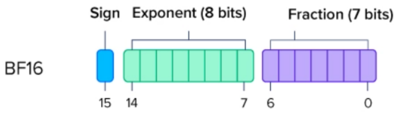
- Gradients can underflow
- Weight updates are imprecise



Recipe for Mixed precision Training:

1. Maintain copy of model parameters and FP32 (Master Weights)
2. Run forward pass in FP16
3. Scale loss by large values (to artificial increase gradients)
4. Computed gradient in FP16
5. Copy gradients in FP32 and divide by scale factor
6. Update master weights in FP32
7. Copy into FP16 version

BF16 can represent much smaller and much larger number (greater dynamic range) but less precision.



Multi-GPU Training

GPU VRAM has:

- NN: Model parameters (in FP16)
- Optimizer: Master weights (FP32) + Adam Momentum(FP32) + Adam Variance (FP32)

Distributed Data parallel (DDP)

- Each GPU has a synchronisation copy of the model with its owns slice of data
- Run forward pass in parallel
- Run backward pass while communicating gradients for upstream parameters
- “All reduce” operation
- Optimizer will get commutative gradients from all GPU

2 bytes for FP16 parameters,2 bytes for FP16 in backward pass gradients, 4 bytes for FP32 master weights, 4 bytes for FP32 Adam Momentum, 4 bytes for FP32 Adam variance.

$$\text{Memory consume} = (2+2+K)*\psi$$

Naive DDP has poor memory scaling. Zero Redundancy Optimizer (ZeRO) technique to optimise memory usage and improve the memory efficiency.
ZeRO stage 1: Optimisers state sharding P_{OS}
Each GPU has full set of FP16 model parameters, compute gradients on subset of data. Each GPU has a shard copy of full optimiser state, and is responsible for updating full parameters

- Each worker computes gradients on its subset of data.
- Perform reduce-scatter so that each worker gets the full gradient corresponding to their parameter shard

- Each worker updates its parameter
- Perform all-gather to synchronise params

$$\text{Memory consume} = 2\psi+2\psi+\frac{K\psi}{N_d}$$

ZeRO stage 2: Optimisers state + gradient sharding P_{OS+g}

- Worker performs a backward pass layer by layer in the computational graph
- Suppose workers is at layer-j
  - Take up stream gradient, compute gradient for all parameters at layer-j
  - Immediately send this gradient to correct worker (reduce)
  - Deallocate memory for parameter gradient
- Worker updates parameters shard using corresponding gradients + state
- Perform all-gather to synchronise

$$\text{Memory consume} = 2\psi+\frac{(2+K)\psi}{N_d}$$

ZeRO stage 3: Full FSDP P_{OS+g+p} When even model parameters won't fit
For ZeRO stage 1 and ZeRO stage 2, communication overhead was “free”

- Divide model parameters into FSDP units
- Shard each unit across multiple GPU
- Run forward pass
  - Perform all gather so that GPU gets all pieces of a module
  - Run a forward pass
  - Discard Param shard
- Run backward pass
  - Perform all gather to get all pieces of the module
  - Each GPU computes gradients on its data chunk
  - Do a reduce scatter to send full gradient piece to right GPU

Each GPU will update its own shard using full gradients received.

Parameter efficient Fine Tuning (PEFT)
Update small subset of model parameters. Why ?

- SOTA models are massively over parametrised
- Unsustainable rate of growth in AI computing scale

Full Fine-tuning

- Assume pre-trained auto regressive language model $P_\phi(y|x)$
- Adapt these pre-trained model to downstream task; training data $\{(x_i, y_i)\}_{1,\ldots N}$
- During full fine-tuning, we update ϕ_0 to $\phi_0+\Delta\phi$ by following gradients to maximise conditional language modelling objective

$$\max_{\phi}\sum_{(x,y)}\sum_{t=1}^{|y|}\log\left(P_\phi(y_t|x,y_{<t})\right)$$

This is expensive and challenging for storing and deploying the independent instances
Key Idea: Encode the task specific parameter $\Delta\phi=\Delta\phi(\theta)$ increment by smaller sized set of parameters $\theta, |\theta|\ll|\phi_0|$

$$\max_{\Theta}\sum_{(x,y)}\sum_{t=1}^{|y|}\log\left(P_{\phi_0+\Delta\phi(\theta)}(y_t|x,y_{<t})\right)$$

Low rank parametrized updates matrices : Update to weight have low “intrinsic rank” during adaptation $W_0\in\Re^{d\times k}$: pretrained weight matrix. Constraint its Update with low rank decomposition $W_0+\Delta W=W_0+\alpha BA$, where $B\in\Re^{d\times r} A\in\Re^{r\times k}, r\ll min(d,k); \alpha$ is trade of between pre-trained “knowledge” and task specific “knowledge”
Often LoRA is applied to weight matrices in self attention module. Adapting W_Q and W_V gives best performance when applying LoRA to transformer.