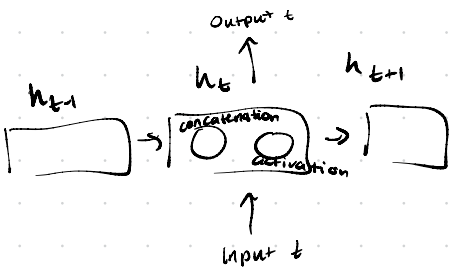
Efficient ML transformers

Part 1: Lecture 12

NLP: Discriminative Generative
Pos/Neg

RNN



hidden state t
takes input and prev hidden state

Problem w/ RNN / LSTM

↓ O(seq_length) for interactions btwn tokens

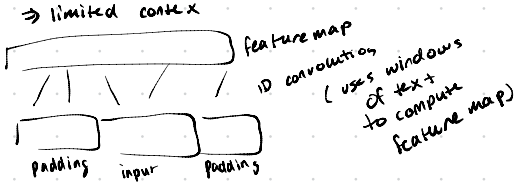
↓ images have locality but languages don't

↓ limited parallism (has dependencies)

Convolution Neural Network

⇒ no dependency btwn tokens
↓ better scalability

⇒ limited context



Transformer

Tokenization

110 words ⇒ 162 tokens
~ similar scale

Embeddings

① one hot encoding — each value is a #

② word representation in vector

Self Attention

Softmax
↳ logistic function
$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$

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

⇒ multiple Q and k to get inner product, normalize by d_k

⇒ $O(N^2)$ complexity

⇒ softmax ⇒ attention weights ⇒ so sum to 1 for each query ⇒ prob matrix

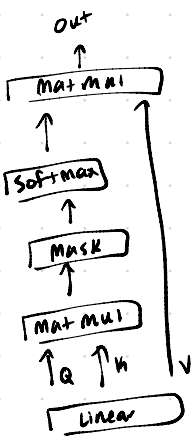
⇒ multiply attention weights by V (values for key)

↳ n x d (one for each query)

$Q, K, V = (N \times d)$ ↓ # of features

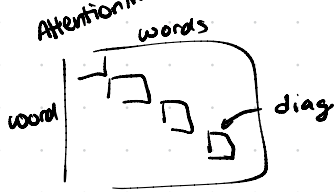
$QK^T = n \times n$

↳ now each query and key are similar



Attention = correspondence btwn words

QKV doesn't rely on prev so done in parallel



↳ write self attention in pytorch

```
class SelfAttention(nn.Module)
def __init__(self, n)
    super(SelfAttention, self).__init__()
    self.N = n
    self.q = nn.Linear(d,d)
    self.k = nn.Linear(d,d)
    self.v = nn.Linear(d,d)
    self.softmax = nn.Softmax(dim=2)
```

m batches
n words
d features per word

nn.Linear(d,d)
↳ take input dim d and project to output of dim d
↳ take input embeddings + put into Q, K, V
output = input × weights^T + bias

↳ softmax function applied on 3rd dim of vector

$m \times ((n \times d) \times (d \times n)) = m \times n \times n$ (m batches)

so $m \times n \times n$ ← applied here

```
def forward(self,x)
    queries = self.query(x)
    keys = self.key(x)
    values = self.value(x)
    score = torch.bmm(queries, keys.transpose(1,2))   # bmm = matmul
    attention = self.softmax(scores)
    weighted = torch.bmm(attention, values)
```

self.$d^{0.5}$

linear trained through back propagation + gradient descent

softmax(dim=2) → 3rd dim

so along each v_k should add up to 1

```
x = torch.tensor([
    [[ v1 ],
     [ v2 ],
     [ v3 ]],
    [[ v1 ],
     [ v2 ],
     [ v3 ]]
])
```

$head_i = Attention(Q W_q^{(i)}, K W_k^{(i)}, V W_v^{(i)})$

$d_k = d_v = d_{model}/h$

$W_{q,k,v} = d_{model} \times d_k$

class MultiHeadAttention(nn.Module)

def __init__(self, d_model)

embedding dim $= d_k \cdot h = d_v \cdot h$ $\quad d_{model} = d\_embeddings$

super(MultiHeadAttention, self).__init__()

self.W_Q = nn.Linear(d_model, $d_k \times h$) W_q

same for K, V

$d_k = d_v = \frac{d_{model}}{h}$

q

self.concat = nn.Linear($d_k \times h$, $d_k \times h$) W_o

already concatted but applying linear transformation via W^o

def forward(self, Q, K, V, mask) → can be x, x, x bc don't need to apply 2 linear transformations

$x \rightarrow Q \rightarrow W_Q Q$ one linear transformation

batch_size, seq_length, d_model = .size()

queries = self.query(Q) → $Q \cdot W^Q$

queries = queries.reshape(batch size, seq length, h, d_k)

queries.transpose(1, 2) → (bs, h, seq length, d_k)

scores = torch.bmm(queries, key.transpose(-2, -1)) / $d\_k^{1.5}$

seq_length $\times d_k \rightarrow d_k \times$ seq length

scores = scores.masked_fill(mask=0, $-1 \cdot 10^9$)

y mask = 0 replace w/ $-1 \cdot 10^{10}$ y so then becomes 0 in softmax bc $e^{-\infty} = 0$

attention = nn.softmax(scores, dim=-1) → (bs, h, seq, seq)

weighted = torch.bmm(attention, values) → (bs, h, seq, d_k)

→ bs, seq, h, d_k

add dropout

weighted.transpose(1, 2)

weighted.reshape(bs, seq, $h \times d_k$) concat($h_1 .. h_n$)

y d_{model}

self.out(weighted) Concat $\times W_o$

Multi Head Attention

- each head captures dif semantics

each head produces attention map

$$\text{MultiHead}(Q, K, V) = \text{concat}(\text{head}_1, \ldots \text{head}_h)\, W^O$$

$$\text{where head}_i = \text{Attention}(Q W_i^Q, K W_i^K, V W_i^V)$$

$$W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_{K,V}}$$

Attention Masking

- one token can only see tokens before it
- causal self attention (masking) rather than global self attention (discriminate task)

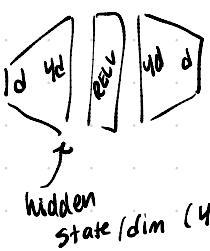
Feed Forward Layer

⇒ add non linearity

⇒ two layer MLP w/ larger hidden state (inverted bottleneck) + RELU/GELU activation

$$FFN(x) = \max(0, x W_1 + b_1) W_2 + b_2$$

max(0, x)



hidden state dim (4d)

Inverted Bottleneck

- inverted residual blocks
- expand dim then reduce to learn richer feat representation

Layer Norm / Residual Connection

⇒ performs normalization for each token then applies learnable affine transformation (linear mapping)

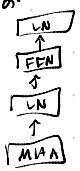
⇒ for each token embedding, mean = 0, var = 1

★ not batching like CNN

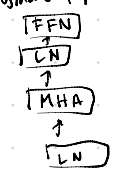
$$y = \frac{x - E[x]}{\sqrt{Var(x) + \varepsilon}} * \gamma + \beta$$

- $\frac{x - E[x]}{\sqrt{Var(x) + \varepsilon}}$: normalize
- $* \gamma + \beta$: learnable affine transformation (linear)

Post Norm



Pre Norm (more popular, better)



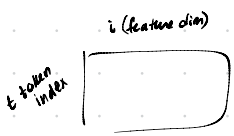
Positional Encoding

⇒ far away dif meaning

$$\vec{p}_t^{(i)} = f(t)^i = \begin{cases} \sin(w_k \cdot t) & \text{if } i = 2k \\ \cos(w_k \cdot t) & \text{if } i = 2k+1 \end{cases} \qquad w_k = \frac{1}{10000^{2k/d}}$$

i (feature dim)

t (token index)



Unique encoding across each token position in sentence and across dif features

⇒ Transformers better accuracy, less training cost

Design Variants

Encoder + Decoder (T5), encoder-only (BERT), decoder only (GPT)
(see everything)

Absolute Pos encoding ⇒ Relative Pos encoding

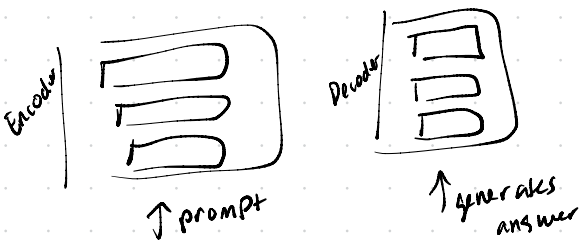
KV cache Optimization

FFN ⇒ GLU (gated linear unit)

Encoder - Decoder Architecture (T5 - Text to Text Transformer)

Text to Text model for transfer learning → pretrained for diverse set of tasks

- Translation
- Acceptability
- Semantic Similarity (0-5)
- Summarization



Encoder Only (BERT)

Two pretraining objectives

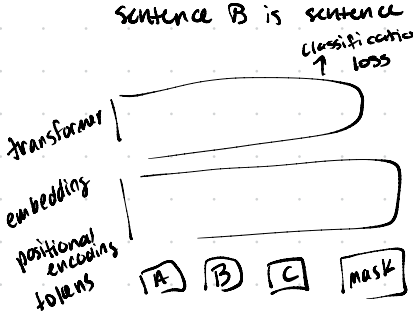
pretraining stage
⇒ train on unlabeled data to learn general language representations

(special token = mask token!)

1. Masked (more popular)
   15% masked at random
   train to predict words

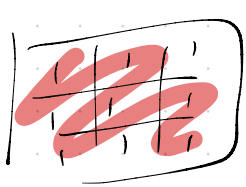
2. Next sentence Prediction
   sentence B is sentence after A

then pretrained model finetuned for downstream tasks



Full attention ⇒ can see all words, prev + future

Attention Mask



Decoder-Only Model (GPT)

↳ generative pretrained transformer

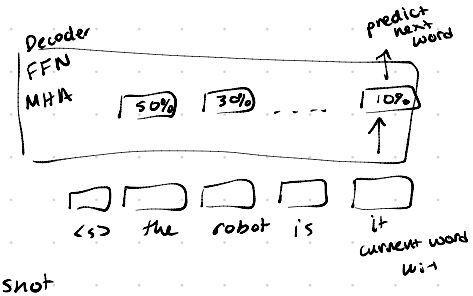
next word prediction

$$L_1(u) = \sum_i \log\left(P(u_i \mid u_{i-k} \dots u_{i-1}) \theta\right)$$

Attention Mask

→ only see prev words

→ generating next word



→ fine tune for downstream tasks OR zero shot/few shot (doesn't require finetuning)

Absolute Positional Encoding

↳ in embedding thus (K, Q, V)

Relative Positional Encoding

↳ impact attention score (bias to the attention map, or modifying query of key)

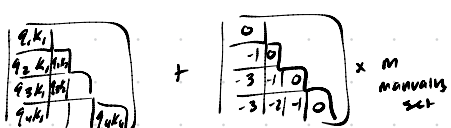
↳ can generalize to sequence length not seen in training

↳ train w/ short text

↳ test w/ long

⇒ ALiBi

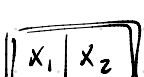
(offset to attention matrix



⇒ Rotary Positional Embedding

⇒ split embedding d into d/2 pairs ⇒ 2D coord

⇒ apply rotation according to pos m



⇒ $m\theta_i$ ← rotation angle

position 1, 2, 3 ..

$$\theta = \{\theta_i = 1000^{-2(i-1)/d}, \; i \in [1, 2 \dots d/2]\}$$

$$RoPE(q_j, m)$$

phase any of inner product of two complex vectors is phase difference

$$RoPE(q_j, m), RoPE(k_j, n) = RoPE(q_j k_j, m-n)$$

difference, relative

$$\theta = \{\theta_i = 1000^{-2(i-1)/d}, \; i \in [1, 2 \dots, d/2]\}$$

# of dif tokens in sequence (large for long text)

$$f_{\{q,k\}}(x_m, m) = R^d_{\theta,m} W_{\{q,k\}} x_m$$

$$R^d_{\theta,m} = \begin{pmatrix} \cos(m\theta_1) & -\sin(m\theta_1) & & \\ \sin(m\theta_1) & \cos(m\theta_1) & & \\ & & \ddots & \\ & & & \cos(m\theta_{d/2}) & -\sin(m\theta_{d/2}) \\ & & & \sin(m\theta_{d/2}) & \cos(m\theta_{d/2}) \end{pmatrix}$$

double length of sequence

↳ rotate by $\theta_i' = \theta_i / 2$

↳ differentiate by more tokens

$m \Rightarrow 2m$ tokens in sequence

KV cache optimizations

GPT - KV cache

⇒ store key values of all prev tokens

⇒ only need current query

Decoder

↳ only attends to prev tokens

Llama 2 7B, KV Cache Size

$$BS \times 32 \times 32 \times 128 \times N \times 2 \times 2 \text{ bytes} = 512\,KB \times BS \times N$$

BS: minibatch ↓ # of users; 32: layers; 32: heads; 128: emb length; N: sequence length ie # of tokens; 2: K,V

bs=1 n-seq = 512 1.25 GB of mem

bs=1 n-seq = 4096 10 GB (~a paper)

bs=16 n-seq = 4096 160 GB (two A100 GPU!) → GPU = 8 GB of memory = cost $20k

4K words

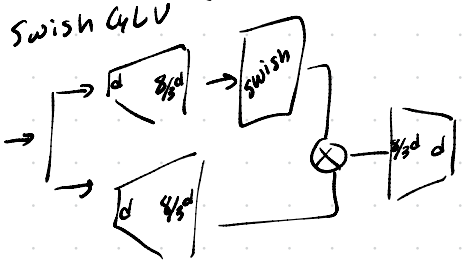
	Heads for Query	Heads for KV	
Multi head Attention	N	N	
Multi Query Attention	N	1	⇒ avg val for all KV into 1
Grouped Query Attention (btwn multi head + multi query)	N	G = N/8	⇒ group every two heads, matches accuracy of multi head attention

FFN ⇒ GLU (Swish + Gated linear unit)

$$(x, W, V, W_2) = (\text{Swish}(xW) \otimes xV)W_2$$

FFN SwiGLU (3 layer)

Swish GLU



GLU

$$y = \frac{x}{1 - e^{-1.702x}}$$

swish

$$y = \frac{x}{1 + e^{-x}}$$

similar shapes

Llama introduced Swish + GLU, RoPE ↗ relative positioning

Emergent: large models

modified arithmetic, word unscrambling

few shot

Original Transformer

- attention is all you need
- encoder + decoder → specific task

- Dropout to output of sublayer before added to sublayer input + normalized (LN) — Pdrop = .1

In residual Connection:

residual_input = x + dropout(sublayer(x))

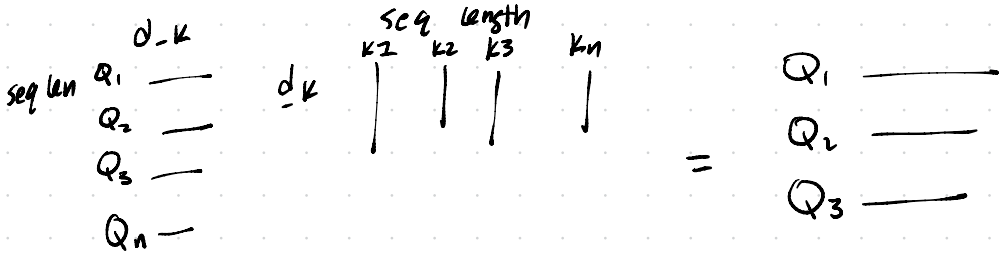
output = LN(residual_input)

In Sublayer:

- → after pos encoding
- → after RELU in FFN ↗ but before second so features not lost due to dropout
- → after softmax in Multihead attention

} after activation function

- before multiplying times values



attention scores → query seq-length × key sequence length

× value → $(q\_seq \times key\_seq) \times (v\_seq \times d\_model)$

Cross entropy loss

→ ∞ if predict 0 for actual

$$\sum_{x \in X} p(x) \log(q(x))$$

$p(x)$: actual, $q(x)$: pred

super(class, self).__init__()

embeddings

nn.Embeddings(vocab-size, d_model)

positional encoding

even_indices(2i): $PE(pos, 2i) = \sin\left(\frac{pos}{1000^{2i/d\_model}}\right)$

odd_indices(2i+1): $PE(pos, 2i+1) = \cos\left(\frac{pos}{1000^{2i/d\_model}}\right)$

↳ add to embeddings, absolute PE

PE pos (token pos [0-seq-length)) i (feature pos (0-d_model))

self.dropout = nn.Dropout(Pdrop) → .1

position = torch.arange(0, seq_length).unsqueeze(1) → seq_length x 1

div = torch.exp(torch.arange(0, dmodel, 2).float() · math.log(1000) / dmodel)

i going from 0 to d_model, steps of 2 $e^{\log(1000) \cdot 2i / d\_model}$

pe = torch.zeros(seq_length, d_model)

pe[:, 0:2] = torch.sin(pe / div_term)

↓

[:, 1:2]

↑

odd bc 1 out of 2

pe = pe.unsqueeze(0) → (seq_length, d_model) → (1, seq_length, d_model)

self.register_buffer('positional-encoder', pe) → not a model param / not learnable but stored as part of the module

↳ self.positional_encoder

def forward(self, x)

x = x + self.pe[:, seq_length, :].requires_grad(False)

return self.dropout(x)

LN

normalize token then linear transformation

LayerNorm(x + sublayer(x))

residual output
use both x + sublayer(x)
to avoid the vanishing
gradient problem

$$\alpha \left(\frac{x - E[x]}{\sqrt{var(x) + \varepsilon}} \right) + self.bias$$

ε = .00001 to avoid divide by 0

```
α = nn.Parameter(torch.ones(1))      # 1 dim tensor
b = nn.Parameter(torch.zeros(1))

def forward(self):
    mean = x.mean(dim=-1, keepdim=True)   # dont change # of input dim
    std = α.std(dim=-1, keepdim=True)
```

FFN

$(max(0, xW_1 + b_1))W_2 + b_2$

dropout

$d_{ff} = d\_model \times 4$

```
self.linear1(d_model, dff)
self.dropout(Pdrop)
self.linear2(dff, dmodel)

forward(x)
    self.linear2(self.dropout(torch.relu(self.linear1(x))))
```

Residual connection

residual_output = x + dropout(sublayer(x))

```
forward(x, sublayer)
    self.layerNorm(x + self.dropout(sublayer))
```

x = residual

Encoder

N=6 layers

↳ each layer is encoder block

Class Encoder

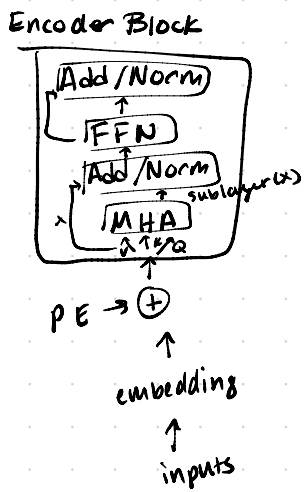
self.layers = nn.ModuleList(...)

↑ EncoderBlock range(N)

forward (x, mask)

for layer in layers

layer(x, mask)



Class Encoder Block

self.MHA

self.FFN

self.residual connections = nn.ModuleList([ResidualConnection() for _ in range(2)])

forward (x, mask)

attention (x, x, x) ← dont need to compute QKV separately

q ↑ k ↑ v

$W_Q x = Q$

dont need two linear transformation (redundant)

→ residual [0] → FFN → residual [1]

---

src_mask (encoder input)

tgt_mask (decoder input)

Decoder Block

output prob

↑ softmax

↑ linear

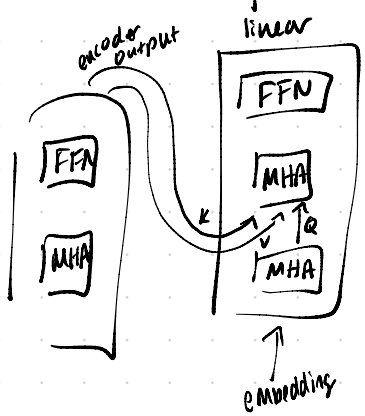
Class Decoder

self.layers = nn.ModuleList(DecoderBlock range(N))

forward

for layer in layers

layer(x, encoder_output, tgt_mask, src_mask)



Decoder Block

MHA1
MHA2
FFN
residual = nn.ModuleList([ResidualConnection.. range(3)])

forward

x = MHA(x, x, x, tgt_mask) → residual
x = MHA(x, encoder_output, encoder_output, src_mask)
→ residual
x = FFN(x) → residual

Projection Layer

def __init__
self.proj = nn.Linear(d_model, vocab_size)
def forward(x) → x.shape = [bs, seq_length, d_model]
torch.log_softmax(self.proj(x), dim = -1)

Transformer

def __init__()
self.encoder ← nn.module
self.decoder so call
self.encoder(x) will run forward
self.src_emb
self.tgt_emb
self.src_pos
self.tgt_pos
self.proj_layer

def encode(self)
embed → pos → encoder
def decode(self)
embed → pos → decoder
def project
self.projection_layer

build transformer

```
for p in transformer.parameters()
    if p.dim()>1 → only for tensors w/ dim >1
        nn.init.xavier_uniform(p) ← glorot initialization
                                    maintain variance throughout
                                    params to prevent vanishing gradient
```

tokenizer

Check if tokenizer file otherwise tokenize

```
tokenizer = Tokenizer(WordLevel(unk_token = "UNK"))
                                  ↑ new word level
trainer = WordLevelTrainer(special_tokens = ["[UNK]", "[PAD]"
                                             "[SOS]", "[EOS]", min_frequency = 2)
                                                    ← min # of times token appears in corpus to be in vocab
tokenizer.train_from_iterator(sentences, trainer=trainer)
tokenizer.save(filepath)
tokenizer.encode(sentences)
```

get Data

```
load data set
DataLoader(data, batch_size, shuffle=True)
```

Causal mask

```
mask = torch.triu(torch.ones(1, size, size), diagonal=1).type(torch.int)
```

```
model = build_transformer(vocab_size_src, vocab_size_tgt, seq_len_src, seq_len_targ, d_model)
```

Process Data(Dataset)

def __init__(self tokenizer_src, tokenizer_tgt, seq_length)

def __getitem__():

pair

y get padding

src_tokens = tokenizer_src.encode(src).ids

tgt_tokens

src_padding = seq_len - 2 ← SOS EOS

tgt_padding = seq_len - 1 ← SOS

SOS = torch.tensor (tokenizer_tgt.token_to_id("[SOS]")

y check if too long ie padding < 0

get encoder input

torch.cat SOS tensor (tokens), EOS, padding

get decoder input

SOS tensor padding

get label (correct decoder)

tensor, EOS, padding

return {

encoder input } dim = seq_length

decoder input

encoder mask → encoder_input ≠ padding token

decoder mask → not padding + causal_mask(seq_length)

label

src_text, tgt_tex

}

get_next_tokens()

sos_idx = tokenizer_tgt.token_to_id('[SOS]')

y token for SOS

eos_idx

encoder_output = model.encode(source, source_mask)

decoder_input = torch.empty(1,1).fill(sos_idx).type_as(source).to(device)

```
while True
    decoder_input.size(1) = max length
        break

    decoder_mask = causal_mask (decoder_input.size(1))

    out = model.decode ( decoder_input, encoder_output, tgtmask, srcmask)
    prob = model.projection (out[:, -1])   # just for last decoder input

    max_val, next_word = torch.max (prob, dim=1)   # item: tensor -> python
    decoder_input = torch.cat ([decoder_input, torch.empty(1,1).fill (next_word.item())
                                .to(device)])

    if next_word = eos_idx
        break

return decoder_input.squeeze(0)
        # make a sequence of tokens


def run_validation
    model.eval ()
    with torch.no_grad ()

        batch [encoder_input]

        batch [encoder_mask]

result = get_next_tokens (model. -)

        batch[src_text]
        batch[tgt_text]
        model_out_tex = tokenizer.tgt .decode (result .detatch ().cpu() .numpy())
```

train model

vocab_size = tokenizer.get_vocab_size()

optimizer = torch.optim.Adam

loss

label smoothing $\epsilon_{ls} = .1$

↓ hurts perplexity

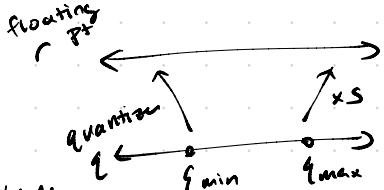
↑ increase accuracy (BLEU score)

↓

Bilingual Evaluation Understudy

↓ add prob to all classes, correct class less, ⇒ regularization

Part 2: Lecture 13



Quantization

14 pt range ⇒ integer values

⇒ important for CNN, destroy performance for LLMs

⇒ LLMs have outliers in activations so can't quantize (0-70)

↳ weights easy to quantize, range 0-1

Smooth Quant / W8A8

⇒ activation outlier mitigation

Accuracy remains the same

$Y = XW$ (10)

enlarge weight, make activation smaller

* used in post training (not pretraining hard to learn quantization levels in pretraining ⇒ degradation)

weight ⇒ harder to quantize

activation ⇒ easier to quantize

$\hat{X} = X\,\mathrm{diag}(s)^{-1}$

$\hat{W} = \mathrm{diag}(s)\,W$

⇒ no extra overhead runtime

⇒ computed during compile time

X = | 1 | -6 | 2 | 6 |
| --- | --- | --- | --- |
| -2 | 8 | -1 | 9 |

W =	2	-1	-2
1	-1	-1
2	-1	-2
1	1	-1

s = | 1 | 4 | 1 | 3 |
| --- | --- | --- | --- |

fold into activation function

compute offline

$$Y = (X\,\mathrm{diag}(s)^{-1})(\mathrm{diag}(s)\,W) = XW = \hat{X}\hat{W}$$

$$s_j = \max(|X_j|)^{\alpha} / \max(|W_j|)^{1-\alpha} \quad j = 1, 2 \ldots C_i$$

↳ scaling factor is constant #

⇒ Reduces Memory By 1/2

⇒ lower latency

⇒ use quantization in all compute intensive operations, Linears, BMMs

lower perplexity ⇒ better

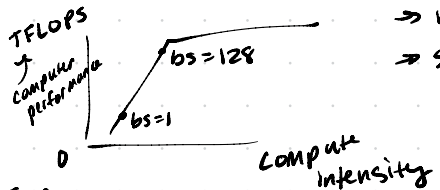
↳ -log likelihood

↳ perplexity = X ⇒ choosing btwn X equally likely options

↳ measurement for accuracy of predicting the next word

Single batch serving

W8A8 quantization good for batch serving



TeraFlop
↳ floating pt operations per second

→ W8A8 good for batch serving

→ small batch ⇒ memory bounded ⇒ need low bit weight only quantization
(time to complete restricted by memory)

Why weight only quantization:

→ main bottleneck is storing weights (memory)

→ SmoothQuant (W8A8) good for batch not bs=1

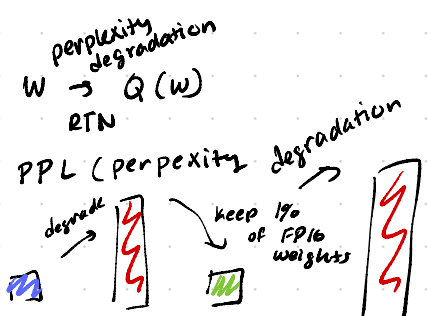
→ focuses on quantizing memory intensive high precision format (FP16)

Low bit weight only Quantization

↳ leaves activations in high precision format (weights bigger matrix)
x W = y

↳ compute cheap, memory expensive

↳ RTN (round to nearest)



How to select important weights?

weight is mult times activations

so look at activations to chose which to save

↓

keep big outliers

AWQ for low-bit weight Quantization

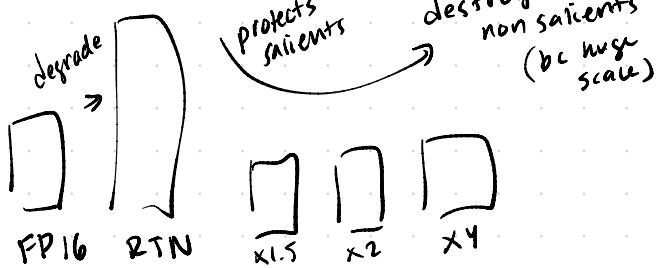
protecting salient weights by scaling

$Wx \rightarrow Q(Ws)(s^{-1}X)$

multiply salient channels w/ $s>1$ reduces quantization error

↑ has significant impact on output (identify via activation)

better than prev bc before 1% are unquantized ⇒ hard to write kernel



AWQ Match:

Consider linear layer: $y = wx \rightarrow y = Q(w)x$

$Q(w) = \Delta \cdot \text{Round}(w/\Delta) \quad \Delta = \frac{\max(|w|)}{2^{N-1}}$

scaled version: $Q(w \times s)(x/s)$

AWQ

also works for multimodal LLMs

Tiny Chat

→ light weight chatbot for LLM on the edge

→ 4 bit AWQ model

↑

quantize

Pruning and Sparsity

$S = |W| \cdot \|X\|$

Pruning (Wanda)

→ consider activation when pruning weights

→ |weight| * ||activation|| as pruning criteria

↳ not just weights

Attention Sparsity (spAttention)

↳ don't need all tokens

↳ small cumulative attention score ⇒ pruned tokens away

Auto regressive

⇒ automatically predict next tokens from previous inputs to system

Cascade Pruning

↳ prune tokens + heads

If QK^T is small don't fetch V and can prune away

→ progressive quantization ⇒ gradually reduce our time to lower precision

⇒ low precision first if not confident, no one that is significant then fetch the high precision

Attention Sparsity

→ only keep local tokens and heavy hitter (H_2) tokens in KV cache

⇒ if attention score is small ⇒ prune away

Contextual Sparsity

↳ context matters

↳ DejaVu (input dependent sparsity)

→ static sparsity (locations of 0 weights are predefined pretraining)

⇒ hurts accuracy w/ medium high sparsity ⇒ replace w/ dynamic sparsity

Dynamic Sparsity:

→ contextual sparsity (small, input dependent sets of redundant heads)

→ accele rate inference w/out hurity model quality

Mixture of Experts

MoE

→ for each token don't activate entire network

→ same total params, reduce inference cost per token

More experts → larger total model size → better perplexity/lower loss

Capacity factor C $\Rightarrow$ $\frac{\text{tokens per batch}}{\text{number of experts}} \times$ capacity factor

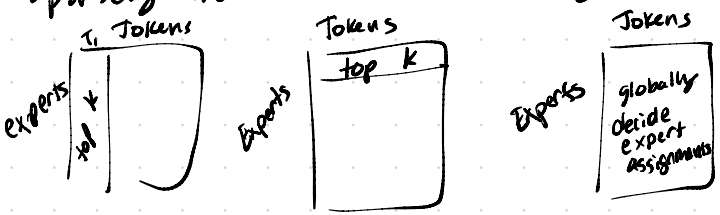
6 tokens, 3 experts:

C=1, every expert can process $\frac{6}{3} \times 1 = 2$ tokens at most

C=1.5 $\frac{6}{3} \cdot 1.5 = 3$ tokens

$\Rightarrow$ have extra slacks

Sparsely Activated Token routing mechanisms



VLLM and Paged Attention

$\rightarrow$ runtime allocation of memory is slow

$\rightarrow$ preallocate $\rightarrow$ but wasteful

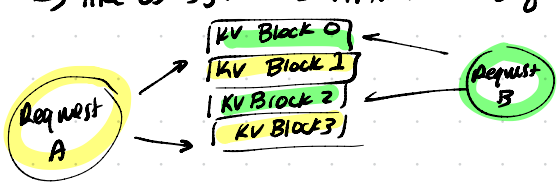
waste

$\rightarrow$ internal fragmentation (over allocation due to unknown output length)

$\rightarrow$ reservation (not used in current step but future step)

$\rightarrow$ external fragmentation (diff. seq. lengths)

$\rightarrow$ like OS system $\Rightarrow$ virtual memory + paging

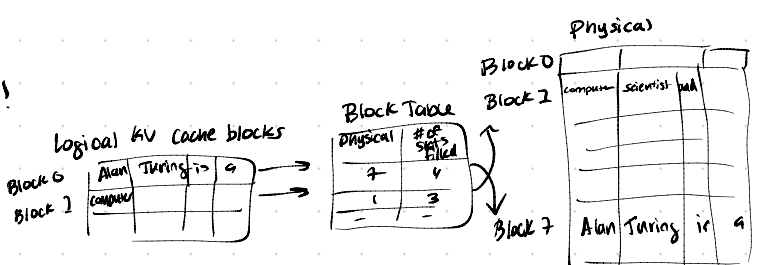


Paged Attention

4 tokens per block

Maximum waste is 3!

$\Rightarrow$ no longer worry about fragmentation!



Paged Attention is good for
Several Suggestion, parallel sampling

Streaming LLM

→ Memory grow exponentially

→ multi round dialog

→ Window Attention → memory constant → perplexity grows → $O(T^2)$ PPL↓
→ 4 tokens ahead

→ Dense Attention → everything → bad eviciency + performance → $O(TL)$ PPL↓

→ sliding window w/ recomputation of KV cache (bad perpexity)
for each incoming token → $O(TL^2)$ PPL↑

Attention Sink (found in Sp Attention paper)

→ heavy attention to first token

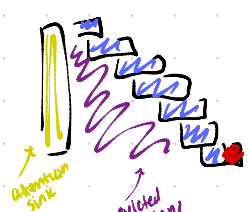
→ due to softmax function → have to 2 for all contextual tokens

→ Initial tokens become sinks due to high visibility

→ if not quite related, dump attention on first token

Solution → Streaming LLM

→ always keep KV + sliding windows (w/ out recomputation)
of attention sinks KV to stabilize



How many attention sink need

→ 4 - 8 → plateau so use 4

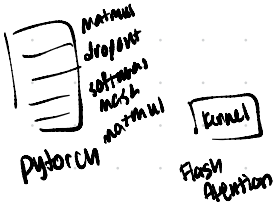
→ 1 attention sink → same performance

Flash Attention

N × N attention matrix

↳ slow high bandwith memory

↳ kernel fusion ⇒ merge into one kernel



Speculative Decoding

Small model ⇒ auto regressive, one by one

decoding → token by token ⇒ memory bounded

large model ⇒ parallel, verification of words ⇒ many tokens at once, no memory bottleneck
↳ so decoding in parallel ⇒ accept, reject correct

Draft Model ⇒ small

Target model ⇒ big (try to accelerate, slow w/ lots of params)

LoRA

low rank adaptation of LLM

instead of full rank → learn low rank ⇒ lower number of training params

QLoRA

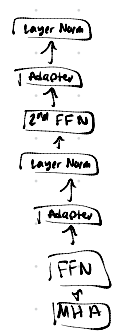
↳ LoRA w/ quantized base model weights

↳ double quantization, quantize scaling factors

↳ new data type for LLM weight quantizer NF4

LoRA Adapter

→ train to new scenerios w/ out changing original params



Prompt Tuning

↳ train prompt for diff tasks ↗ add prompt to task in training

- prompt engineering
- model gets larger ⇒ same accuracy
- mix learned prompts into single batch
- same model to handle diff tasks
  - trained w/ task prompt

Tensor RT Public Release

- all of these methods + tensor parallism + pipeline parallism