Linear models
loss: 平均 cost $\xi = \frac{1}{N}\sum L(y^i, t^i)$ J

① **Linear regression**: $y = W^T x + b$ ($x: D\times 1$) $\underline{y} = \underline{X}\underline{W} + b\underline{1}$ ($x: N\times D$)
$\xi(W) = \frac{1}{2N}\|XW - t\|^2$ $\nabla_W \xi = \frac{1}{N} X^T(XW - t)$ $W^* = (X^TX)^{-1}X^Tt$
Each GD cost $O(ND)$, $(\cdot)^{-1}$ cost $O(D^3)$

② **Logistic regression**: $z = W^T x$ $y = \sigma(z) = \frac{1}{1+e^{-z}}$ $L_{CE} = -t\log y - (1-t)\log(1-y)$

③ **Softmax regression**: $z = Wx$ $y = sm(z)$ $y_k = \frac{e^{z_k}}{\sum_m e^{z_m}}$, $L = -t^T\log(y)$
(W: $K\times D$, x: $D\times 1$, z: $K\times 1$)

multi-Layer perceptions
convex: convex: if $x_1, x_2 \in S$, $\lambda x_1 + (1-\lambda)x_2 \in S$
① $f(\lambda x_2 + (1-\lambda)x_1) \le \lambda f(x_2) + (1-\lambda) f(x_1)$ (lower cost better perform)
② nn: permutation symmetries
④ local min is global min (strict: only 1 global min)

Fully connected layer: $y_{n\times 1} = \phi(W_{n\times m} x_{m\times 1} + b_{n\times 1})$

ReLU: $\max(0, x)$ non differentiable at $x=0$

soft ReLU: $\sigma(z) = \log(1+e^z)$ compt expensive

Expressive Power: if all activation func are linear → NN can only represent linear function, no more than linear expression
Multi-layer feed forward nn with non linear activation func → **universal function approximator**: approximate any f 任意的
eg. binary input & output: each hidden unit is 1 situation

Back propagation
fully con $z_1, z_2, z_3 \to y_1, y_2, y_3$ $z \to y$
$\frac{\partial y_{(m\times 1)}}{\partial z_{(n\times 1)}} = \begin{pmatrix} \frac{\partial y_1}{\partial z_1} & \cdots \\ \vdots & \ddots \end{pmatrix}_{m\times n}$ $\begin{pmatrix} \frac{\partial y_1}{\partial z_n} \\ \vdots \\ \frac{\partial y_m}{\partial z_n} \end{pmatrix}$
$\bar{x} = \left(\frac{\partial y}{\partial z}\right)^T \bar{y}$ — quadratic time
$y = \exp(z)$ $\frac{\partial y}{\partial z}$ is diagonal $\bar{z} = \bar{y} \circ \exp(z)$ — linear time element wise.
$\underset{W}{\overset{b}{h}} \to y$, $y = Wh + b$ $\bar{W} = \bar{y}\cdot h^T$ $\bar{h} = W^T\bar{y}$
(y: $n\times 1$, W: $n\times m$, h: $m\times 1$; $\bar{W}$: $n\times m$, $\bar{y}$: $n\times 1$, h: $1\times m$; $\bar{h}$: $m\times n$, $\bar{y}$: $n\times 1$)
$W^{(2)}_{ki} = \bar{y}_k h_i$, $\bar{h}_i = \sum_k \bar{y}_k W^{(2)}_{ki}$ two add-mul per weight
the BP is expensive as 2FP

For multi-layer perceptron, the cost is linear in # of layer and quadratic in # of units (W: k^2 weights)

Auto Diff
VJP: $\frac{\text{scalar}}{d\times 1} = (\,)_{d\times 1}$ $\frac{n\times 1}{m\times 1} = (\,)_{n\times m}$ $\frac{n\times 1}{n\times m^T} = (\,)_{1\times m}$
input → output
def vjp(exp, lambda g, ans, x: g * ans)
$h \to z \to s \to t$, b
$\bar{y}$ y $\bar{h} = \ldots \bar{y} \ldots x$
$\bar{z} = \frac{\partial r}{\partial z}\bar{r} + \frac{\partial s}{\partial z}\bar{s} + \frac{\partial t}{\partial z}\bar{t}$ 就是 compute $\bar{z}\frac{\partial z}{\partial h}\cdots$

Distributed representation
language modeling ≃ Predict the next word
Markov assumption: Prob of the next word depends on a fixed # of prev words
$P(W_t | W_1, W_2, W_3, \ldots W_{t-1}) = P(W_t | W_{t-3}, W_{t-2}, W_{t-1})$
N-Gram model: localist representation → info not share bw N-grams.
Neural language model: distributed representation: words are represent as vectors in a continuous space.
Pros: • Semantic similarity • Generalization (unseen data) • Robustness • Efficiency (mem, comp) • scalability (data, task)
loss $-\sum_{i=1}^{V} t_i \log y_i$ (t_i: one hot encoder 1 for correct word, y_i: Predict Prob)
vs localist • share info between entity • complex to interpret • capture more nuanced info • robust to damage, noise • scale
index of word → embedding → hidden unit → softmax (each skip layer connection)

Global Vector Embedding (GLove)
coocurrence matrix: frequency of 2 words appear nearby
$X = R \times \tilde{R}^T$, $x_{ij} = r_i^T \tilde{r}_j$, $f(x_{ij}) = \begin{cases} (x_{ij}/x_{max})^\alpha, & \text{if } x_{ij} < x_{max} \\ 1, & \text{ow} \end{cases}$
($V\times K$ $V\times K$ $K\times V$)
$J(R) = \sum_{i,j} f(x_{ij})(r_i^T\tilde{r}_j + b_i + \tilde{b}_j - \log x_{ij})$,
(log) diminish high count importance, reduce count range, emphasis relative diff. ⊕-balance the influence of diff count, s.t. frequent, infrequent pair contribute meaningfully.
only non zero count matter → huge comp saving

Pros: • global statistics • semantic rela • quick trained • pre train ava
Cons: • static embedding • x-sign mem for store & comp • unseen w in train

optimization
local minima: nn is not convex (weight space symmetry)
⇒ random start, local minima are generally fine

saddle point: $f'=0$ $f''>0$ & $f''<0$ (in 2 direction)
eg. two h have same incoming & out coming weights
→ always same gradient → always identical
or all initial w, b are 0 → w be identical every step
⇒ break symmetry, random init, init to small random values

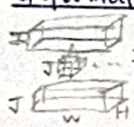
plateaux: flat region eg 0-1 loss, hard thresh, saturated or deadunits (zero gradient)
⇒ choose the scale of the random initializa (↙) of the weights s.t. activations are in the middle of their dynamic range
• Xavier: initialize the weights s.t. variance of the activations is the same across every layer
• add initial bais to small positive value for Relu

Bad conditioned curvature: curvature diff along diff directions eg ravine ⊙ ⇒ normalize input to zero mean, unit variance
• **Batch normalization**: center each hidden activation, normalize the activation per batch during training so act have 0 mean, 1 variance

convolutional neural network
• spatial struct • Translation invariance
Pros: fewer Parameter (↓overfit, effective computation, ↑generalization)
Translation invariance: shared weights allow the layer to detect feature regardless of their position in the image.

convolutional neural network
• Input: raw image, $h \times w \times$ # of channel
• convolutional layer (linear opt): a set of filters (kernal) → a set of feature map
• non linear activation
• Pooling layer: eg. avg, max, L_2 norm, weighted avg
- reduce dimension (↓# of Para) → ↓comp, ↑speed, ↓overfit
- support translation invariance
- higher layer cover larger region ← receptive field ↑ (region of input image affect output)

Size
# of units: store activations in memory for back prop train
# of weights → # of parameters → amount of overfitting train & test
# of connections → complexity of backprop (3 add multiply per connect) (2 back + 1 Forward)



	Fully connected	convolution
# output unit	WHI	(WHI)
# weights	IJW²H²	k²IJ
# connection	IJW²H²	(K²WHJ·I)

dimension of feature map: # of channel

Inception module: parallel convol with various filter size within a single layer, and concatenate the outputs
• capture feature at multiple scales w/ diff filter sizes
• 1×1 convol reduce dim of the feature map, allows for a deeper network w/o significantly more computation complex

optimization and Generation
stochastic gradient descent
Training curve (loss vs # of iteration or epoch)
• use a fixed subset (validation): otherwise introduce noise
• convergence not guarantee: could be local min, saddlepoint
SGD: update model parameters based on the gradient calculated from a single, randomly chosen training example
SGD: faster update, large data✓, noise update
Batch: mem↑, stuck local, guarantee converge, can't use optimized matrix operation
mini batch GD: compute gradient on randomly sampled subset of training data
Pros: • faster training (vectorized opt) • reduce gradient variance leads to smoother convergence • handle large dataset
optimize Batch size: $2^5/6/7/8$, validation error (train time) for diff batch sizes
• Tune other hyper para before, α • use GPU capacity by ↑ S
learning rate: when the error stops decreasing → ↓ α
• $\alpha\downarrow$ can caused a sudden drop in loss at expense of LT performance
The sudden drop is because of a reduction in upward fluctuation than a dramatic improvement.
contour map: 同一条线 y 相同 • 越密 gradient 越大 • gradient ⊥ contour line 并且指向 increasing function value (GD 指反方向)
SGD struggles when the loss is steep vertically but flat horizontally
• progress slowly along the flat direction • oscillates wildly along steep

Momentum: Add a fraction of the previous update to the current update
$\begin{cases} m_{k+1} \leftarrow \mu m_k + \alpha \frac{\partial J}{\partial \theta} \\ \theta \leftarrow \theta - m_{k+1} \end{cases}$ smooth out the updates (estimate main of gradient)
μ - damping factor
how quickly the contribution of prev gradient decay

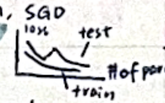
RMS prop (Root Mean Square propagation) Adjust the learning rate for each parameter j based on the moving avg S_j of squared gradient • An exponential moving average of the squared gradient $S_j \leftarrow (1-\gamma)S_j + \gamma\left[\frac{\partial J}{\partial \theta_j}\right]^2$ $\theta_j \leftarrow \theta_j - \frac{\alpha}{\sqrt{S_{j+\varepsilon}}}\frac{\partial J}{\partial \theta}$
• Divide the lr by the squared root of this average
g const large, → reduce α to prevent overshoot
g const small → ↑ α to speed up learning

Adam (Adaptive moment Estimation) like momentum
mv of past gradient (momentum) $m_{k+1} \leftarrow B_1 m_k + (1-B_1)\frac{\partial J}{\partial \theta}$
mv of past squared gradients $S_{k+1} \leftarrow B_2 S_k + (1-B_2)\left[\frac{\partial J}{\partial \theta}\right]^2$
correct the bias in mv $\hat{m}_{k+1} = \frac{m_{k+1}}{1-B_1^{k+1}}$, $\hat{S}_{k+1} = \frac{S_{k+1}}{1-B_2^{k+1}}$
update para $\theta_{k+1} \leftarrow \theta_k - \frac{\alpha}{\sqrt{S_{k+\varepsilon}}}\hat{m}_{k+1}$
Instability, 收敛 prog → momentum, α
Fluctuation: Decay α / iterate averaging

Dead / saturated units (through Activation histogram): initial scale of w, ReLU
ill condition: Adam, second-order opt (use curvature information)

generalization: ways to prevent overfitting
① **Data Augmentation**: Augment training set (as transformation), not test data
② **Reduce Parameters**: Add linear bottleneck layer (eg. Auto Encoder, Vimb)
③ **Regularization** $\xi_{reg} = \xi + \lambda R$ $\underline{w} \leftarrow \underline{w} - \alpha(\frac{\partial \xi}{\partial \underline{w}} + \lambda\frac{\partial R}{\partial \underline{w}})$ L_1: $R = \frac{1}{2}\sum|w_i|$
L_2: $\xi_{reg} = \xi + \frac{\lambda}{2}\sum w_i^2 \Rightarrow w \leftarrow (1-\alpha\lambda)w - \alpha\frac{\partial \xi}{\partial w}$ (weight decay) ↑≥0 encourage w ↓≤0
④ **Early stopping**: # epochs ⑤ **Ensembles**: Avg or combine 多个 model prediction
⑥ **Stochastic Regularization**: avoid 记 noise → inject noise into network
dropout: randomly deactivate neurons during training (ensemble)
During training: each neuron is active i.p of the time, of each neuron
scale the weights by 1-p at test time approximate the expected activation
Other: drop Connect, Batch Normalization, SGD

overparameterization & Normalization
Double descent: regularization can ↓ model capacity effectively "smooth out" the double descent curve and eliminate the second descent.



Neural tangent kernal: First order taylor expansion $\Delta Z = J\Delta W$
GD update $\Delta W = -\alpha\nabla J(w)$, so $\Delta Z = J[-\alpha\nabla J(w)]$
$\nabla J(w) = \frac{\partial J}{\partial z}\frac{\partial z}{\partial w} = \frac{1}{N}J^T\nabla L(z)$ logit (output) Gradient of cost func
So $\Delta Z = -\frac{\alpha}{N}JJ^T\nabla L(z)$ $k = JJ^T$ (NTK) The nn's behavior around its current weights can be approximate by a linear model. (This linearization is approximate • Taylor expansion is only accurate for small Δ in W
• As W change during training, the Jacobian J also $\Delta \Rightarrow k\Delta$
Infinitely wide nn (width of hidden layer → ∞) ⇒ const J, k
⇒ nn behave like a linear model with a fixed kernal
⇒ Guarantee learn training data perfectly well, perform well on unseen data
limitation: based on infinit width network, full batch gradient descent, comp 贵

Batch Normalization
covariate Shift: the dist of input data Δ
Internal Covariate shift: the dist. of activation (output of each layer) changes from layer to layer during training ⇒ slower training, unstable gradients, difficult in hyperparameter tuning, reduced model performance, increase sensitivity to initialization.
Normalize preactivation within each mini batch to have 0 mean, 1 var
done for each feature independently
For a mini batch B of size m. for each unit in a hidden layer:
mean: $\mu_B = \frac{1}{m}\sum_i z_i$ variance $\sigma_B^2 = \frac{1}{m}\sum_i (z_i - \mu_B)^2$ normalize $\hat{z}_i = \frac{z_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$
scale & shift $z_i^{BN} = \gamma\hat{z}_i + B$

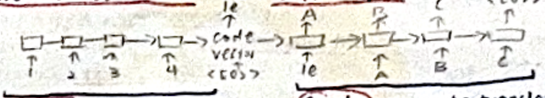
Implicit learning rate decay
Batch Normalization → Implicit learning rate decay
$\hat{W}_j^{(k+1)} = \frac{W_j^k - \alpha\nabla J(W_j^k)}{\|W_j^k - \alpha\nabla J(W_j^k)\|} \approx \frac{W_j^k - \alpha\nabla J(W_j^k)}{\|W_j^k\|} = \hat{W}_j^k - \alpha\|W_j^k\|^{-2}\nabla J(\hat{W}_j^k)$
So Effective lr = $\alpha\|W_j^{(k)}\|^{-2}$ (与 square of |w| 成反比)
$\|W_j^{k+1}\|^2 = \|W_j^k\|^2 + \frac{\alpha^2\|\nabla J(\hat{W}_j^k)\|^2}{\|W_j^k\|^2}$
So W magnitude Grows Monotonically $\|W_j^k\|^2 \propto \sqrt{1+k/k_0}$
So Effective lr: $\hat{\alpha}_k = \frac{\hat{\alpha}_0}{\sqrt{1+k/k_0}}$ ↓ ↑

As k↑ (more step), W↑, effective learning rate ↓ (each update 1 batch, 1 iteration) strategy to conter LR decay = • exp↑ LR • Weight decay • Normalize weight to unit vector after

Weight decay: Adam with L^2 Regularization
$g_t = \frac{\partial J}{\partial \theta}$ • add weight decay $g_{twd} = g_t + \lambda W_t$
• avg of grad $m_{k+1} \leftarrow B_1 m_k + (1-B_1)g_{twd}$
• avg of squared grad $S_{k+1} \leftarrow B_2 S_k + (1-B_2)g_{twd}^2$
lestimate var of gradient) sol ⇒ last step $w \leftarrow (1-\lambda)w - \alpha\partial\xi/\partial w$

Recurrent Neural network shared weights
language modelling using RNN
• Training with teacher forcing: use target output (ground truth) as input for next time step
• Generating use model output as inputs for next time step
• Each word is a one hot vector • predicts a dist.
↳ pmstep over the next word • CE loss • long distance depend

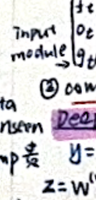
Neural Machine translation challenge: word order, sentence length
Sequence to sequence Architecture: • future words • full context
two distinct network with diff weights



Encoder: Process entire input sentence, compress into a code vector
Decoder: generate translation word by word

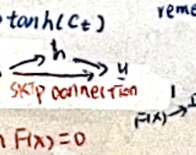
Exploding and vanishing Gradients: RNN suffer: during back Prop through time, gradients are multiplied by the same weight matrix repeatedly. This repeated multiplication causes the gradient to explode or vanish.
• **Forward**: squashes activations (挤压) • with non linear activation function • Protect against exponential growth.
• **Backward**: linear operation compound → highly non linear behavior
RNN has inherent gradient problem for individual exp
⇒ use mini Batch during training helps stabilize grad. update.

Mitigating Exploding / Vanishing gradients
$\frac{\partial z}{\partial w}$ • **Gradient clipping**: if $\|g\| > \gamma$: $\vec{g} = \frac{\eta g}{\|g\|}$ bias vs stable
• **reverse input seq** (在 Encode 里): learn ST dep before LT
• **Identity RNN**: initial W to be identity matrix & ReLU
• **Long term short term (LSTM)** make short memory last longer
$C_{t+1} = f(\text{forget}) * C_t + i(\text{input gate}) * \text{new\_input}$
① i=0, f=1: retain previous mem ② i=1 f=1: add new input to mem
③ i=0, f=0: erase ④ i=1, f=0: overwrite gates are learned, not hard code

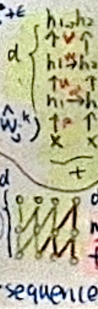


① Compute the gate value $\begin{pmatrix} i_t \\ f_t \\ o_t \\ g_t \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} y_t \\ h_{t-1} \end{pmatrix}$ → [-1, 1] ⇒ add
② update memory cell state $C_t = f_t C_{t-1} + i_t * g_t$ (how much to store; what might we want to remember)
③ compute hidden state $h_t = o_t * \tanh(C_t)$

Deep Residual network
$y = X + F(x) = X + \text{transform}(h)$
$z = W^{(1)}x + b^{(1)}$ $h = \phi(z)$ $y = W^{(2)}h + x$
• If opt mapping is Identity, just learn $F(x) = 0$
• If $\frac{\partial F}{\partial x}$ vanish, gradient can still propogate backwards through the skip connection $\overline{x^l} = \overline{x^{l+1}} + \overline{x^{l+1}}\frac{\partial F}{\partial x} = \overline{x^{l+1}}(I + \frac{\partial F}{\partial x})$



cost and parallelism
• **Flops** - # of connections • **mem** - train: # of activations, test: # of parameters
• **sequential complexity**: # of sequential operation
The hidden dim needs to grow with the sequence length (t) to remember long term information

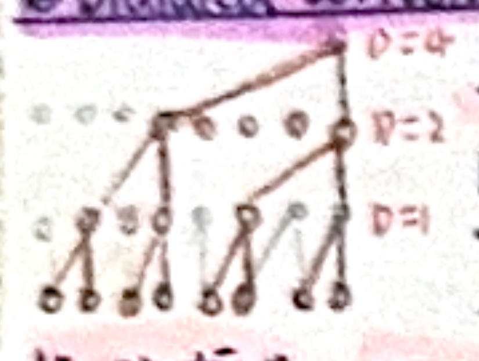


Sequential complexity is bad for long sequence
The hidden state must store all critical info about the input sequence, redundantly across many time step. & long seq ⇒ large time step ⇒ comp 慢
Convolutional Autoregressive model
① **Causal convolution** don't require for loops
depends only on earlier inputs in seq. Implemented by masking or setting zero weights on connections to future inputs.
t-sequence length d-# of layer, k-# of feature per layer, w-kernal width

No temp dep →	CNN	RNN
Connection	k^2wdt	k^2dt
param	k^2wd	k^2d
activation	kdt	kdt
seq(train)	d	$d+t$
seq(gen)	dt	dt

• RNN has mem • CNN limited contex • RNN expensive for long seq • CNN better parallel • Generating for both is expensive

② Dilated convolution

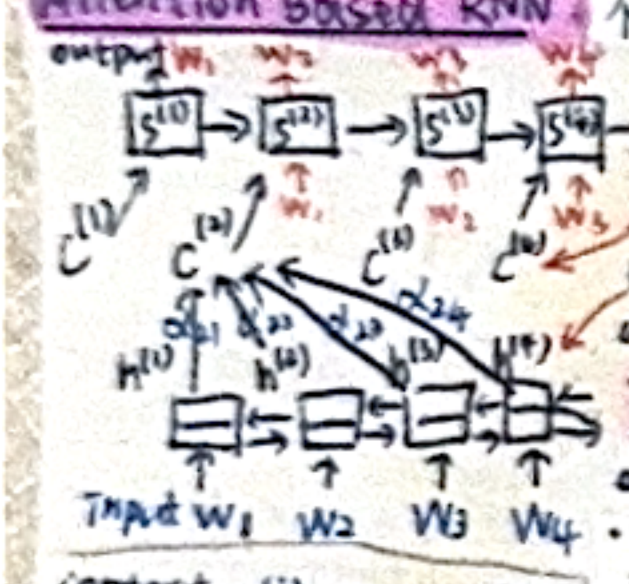


Dramatically increase CNN's receptive field by using dilated convolutions with progressively larger dilation rates in successive layer
• receptive field ↑ (# of layer, # of param)
• effecient computation (not change kernal size

Limitation: ① causal: • only use past information, limit ability to consider future context • Early time steps have less context available, potentially resulting in lower quality output at seq begining
② Dilated: • can create checkboard artifacts if dilation factors aren't carefully designed • May miss fined grained local pattern when d is high

Attention Based RNN

不压缩 compression, model selectively focus



Decoder • predict one word at a time • use output as input • receives a context at each time step. a weighted sum of the encoder annotation • Context: a vector of weighted sum of encoder annotation

Encoder • create an annotation for each input word • Bidirectional RNN • the annotation concatenate two hidden state. Each hidden unit is a Gated Recurrent unit (update, resetgate to control info flow)

context $c^{(i)} = \sum_j a_{ij} h^{(j)}$ (attention weights, encoder annotation j)

$a_{ij} = \frac{\exp(e_{ij})}{\sum_{j'}\exp(e_{ij'})}$ $e_{ij'} = a(s^{(i-1)}, h^{(j)})$ (decoder state)

additive attention: $a(s^{i-1}, h^j) = V_a^T\tanh(W_a s_{i-1} + U_a h_j)$ (alignment model) — Query, key

Content Based Addressing - Annotation function depends on annotation vector $h^{(j)}$. Focus on semantic content rather than position
• direct connection between any input and output.
• Recompute the context vector at each time step by adding tk^2 cost at each step.

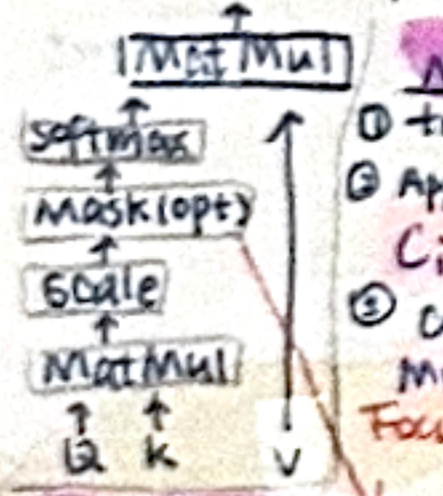
	traing Comp	Train mem	Test Com	Test mem	Seq opt	MaxPath across t
RNN	k^2dt	kdt	k^2dt	kd	t	t
RNN+atte	k^2dt^2	kdt^2	k^2dt^2	kdt	t	1

Self attention & transformer

① $XW^Q = Q$ ($n\times d$, $d\times d_k$) $XW^K = K$ ($n\times d_k$, "k" "t" = $n\times d_k$) $XW^V = V$ ($n\times d_v$)

② $Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V = Z_{n\times d_v}$ (weighted sum of vector values); $n\times n$; scale to improve gradient stability → attention weight

Self attention: dot product - similarity. d_k↑ larger → small gradient in σ

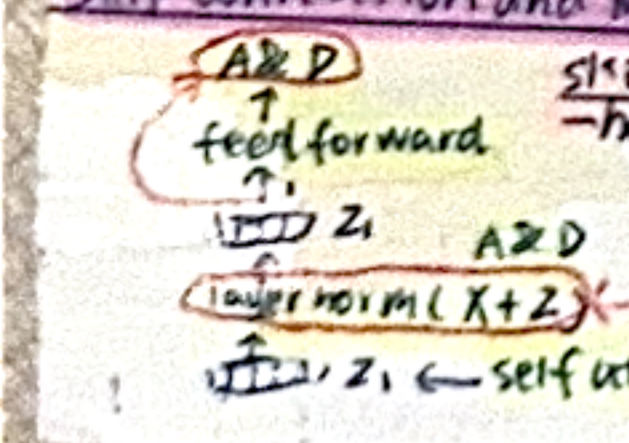


Multihead attention
① transform the inputs linearly
② Apply attention multiple times in parallel $C_i = attention(QW_i^Q, KW_i^K, VW_i^V)$
③ concatenate the output and compress: $MultiHead(Q,K,V) = concat(C_1, C_2, \dots C_n)W^O$ ← trained
Focus on diff part of the input simultaneously

Masking Future Input → Preserve Autoregressive Property: score $= qk = -\inf$
• Encoder - attention is all to all: Each token attends to the entire sequence (including itself)
• Decoder: causal (mask): each token attends to past token 和自己 (gen)
Decoder use 2 seperate mechanisms for attending to encoder output vs ↑

Positional encoding since self attention 不 capture word order
$PE_{pos,2i} = \sin(\frac{pos}{10000^{2i/d_{emb}}})$ $PE_{pos,2i+1} = \cos(\frac{pos}{10000^{2i/d_{emb}}})$
Pos: position in the seq. d_{emb}: the embedding dim size. create a unique vector values for each position

Skip connection and Normalization: Add & Normalization



Skip connection / residual connection - help with gradient flow
Layer norm: Normalize across neuron dimension
"one encoder block" go through mul of them

Transformer

Cross Attention = Decoder generates a query for each output token • The query is compared with key vectors from the encoder output • Based on similarity scores, the decoder assigned attention weights to diff part of en output.

Decoder block: ⊕ → self att → A&D → En.De Att → A&D → FF → A&D →

	MLP	Self Atten	
Connection	k^2dt	kt^2d	k: total dim of the atten head proj
Para	k^2d	k^2d	← $W \to k^2$
activation	kdt	$ktd + t^2d$	← Q, K, V & Atten weights (t^2)
seq train	d	d	← process token parallel in 1 self layer
seq gen	dt	dt	← process one token at a time

	CNN	RNN	Transformer
Propogate info over long dist.	Strugle (need dilated conv)	Good	Good.
Scaling to long sentence	limited receptive field	sequential process. Batch N	quadratic cost in sequence length ($k+t^2d$)
Training Com	Efficient d	Expensive (seq grad comp) $d+t$	efficient d
gen com	all challenging		

Scaling up Transformers

Scaling Trends: Transformer performance improve math regular and predictable pattern as resource ↑ (eg. compute, data, para)

Sparse Attention • mask out most token pairs and keeping only a small subset (s) • reduce the complexity from quadratic to linear in t ($t\sqrt{t}$)

Activation checkpointing • For MLP with d layer, K units per layer {# addmul = $3Kd^2$ (1F+2B); # values store = $2kd$ (pre act, act)} ⇒ mem challenge of Backprop

Check Pointing: save intermediate activations at strategic layers during the Forward Pass. • Recompute intermediate values on demand from the nearest checkpoint during BP. eg MLP store activation every $\sqrt{d}$ layer. Add multiple $4kd^2$ (↑) memory: $2k\sqrt{d}$

Recursive checkpointing: $\frac{d}{2}$ … $\frac{3d}{4}$ → d 回来路上把 mid 存了. Add mul $(3+\log d)kd^2$ mem $2k\log d$

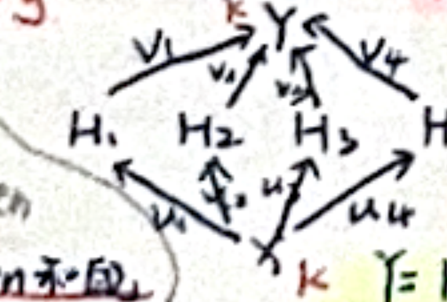
Distributed Training
Using multiple GPUs with minimal communication

Data Parallelism - Each GPU process diff training example. Parameter server store the model parameter. Expensive gradient computations are distributed but the inexpensive parameter updates remain centralized. Trade off: Each of W workers compute a gradient on a Batch of size B = compute a gradient on a Batch of size WB. Arithmetic operation: W× that of a single GPU. Communication cost: 2W copies of para in each iteration. (send & receive) Synchronous SGD is beneficial for architectures with a large computation to para ratio like CNN (shared weights & local connection for each w)

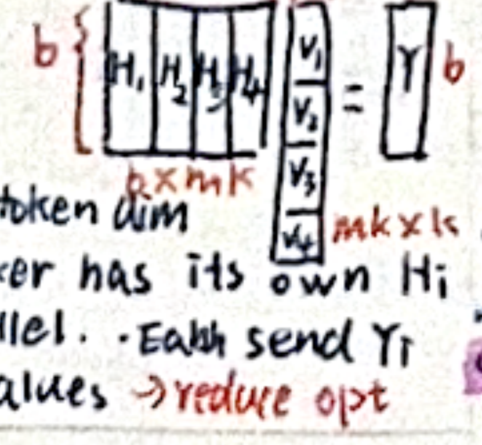
Effective Batch size (WB): Parallelism gains continue until you reach the critical batch size • beyond that, less exploratory gradient update, leading to diminishing returns. The critical batch size rep the opt trade off between computational efficiency and optimization quality. **Limitation** • only benefits training up to the critical batch size • communication bandwidth becomes a bottleneck for models with many parameters • Each worker requires a full copy of the model parameter.

Model parallelism - Each GPU handles diff parts of the model

Intra-layer parallelism - share units and para in a layer among multiple GPUs. reverse bottleneck

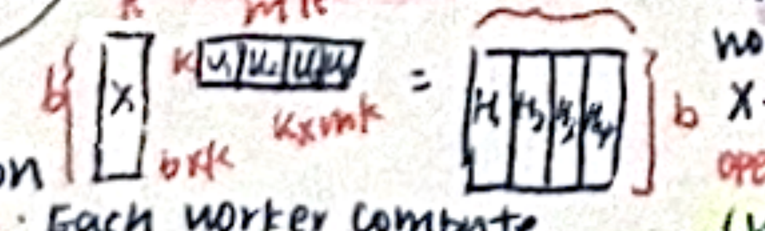


Projection layer: (k): input & output size (dimension) (b): combines the Batch & token dim



$Y = HV = \sum_i H_iV_i$. Each worker has its own H_i and computes $Y_i = H_iV_i$ in parallel. • Each send Y_i to one worker, which sums the values → reduce opt

Expansion layer



• Assume the input X is on one worker at the start. • Distribute X to all workers through a scatter operation • communication cost $(W-1)\cdot size(X) = (W-1)bk$
• Each worker compute $Z_i = XU_i$, $H_i = \phi(Z_i)$ in parallel
Assign subset of hidden units to diff worker.
Forward pass communication and computation:
• Scatter input X across workers $(W-1)bk$
• Each worker: $Z_i = XU_i$, $H_i = \Phi(Z_i)$, $Y_i = H_iV_i$ # of tx: $2\times(bmk^2/W)$
• Reduce $Y = \sum_i Y_i$, communication $(W-1)bk$

Arithmetic opt: expansion & projection $2bmk^2/W$ operation
communication: scatter & reduce $2(W-1)bk$ operation.
Parallelism is beneficial iff the hidden layer is very wide $2bmk^2 \gg 2bk$ or $mk \gg 1$
• Effects: Duplicate on every worker: input X, out Y_i (size bk) • split among workers: Parameter matrices U, V (size mk^2), Activation (mbk) matrix H • mem benefit: 1/W ↓ in parameter mem per worker (suit for LLM); 1/W ↓ in activation mem when $m \gg W$.

Pipelining: place diff layers on diff machine. Each worker handle both forward, backward passes for its assigned layer, includ para update. Pros • Linear scaling for para mem. Each worker stores para for its own layer • Nearly linear scaling for arithmetic opts. If GPUs are kept busy. • Reduce communication overhead than intralayer.

Autoencoders
Feed forward nn, bottleneck layer, take X, predict X

Deep Autoencoder • Deep nonlinear auto encoder learn to project the data, not onto a subspace, but onto a non linear manifold. This manifold is the image of the decoder • This is non linear dim reduction
Limitation: They are not generative model, so they don't define a dist. • How to choose the latent dim? tradeoff: k↑ loss↓, identity↑, k↓ learn Interpretable features

Variational AutoEncoder: Generator Network: sample code vector z from a fixed, simple dist. → (deterministic) generator network compute a 函数 G map z to an $x = G(z)$
• observational model: consider train a generator nn with maximum likelihood $p(x) = \int p(z)p(x|z)dz$. If z is low dim & decoder is deterministic, then $p(x) = 0$ almost everywhere (The model only generate sample over low dim sub-manifold of x. ⇒ Noisy obs model $p(x|z) = N(x; G_\theta(z), \eta I)$). G_θ is the function computed by the decoder with para θ → complicate (integration) ⇒ maximize a lower bound on $\log p(x)$

Variational Inference: convex function: $E[h(x)] \ge h[E(x)]$; $\log z$ is concave, so $E[\log(x)] \le \log(E(x))$

$\log p(x) = \log\int p(z)p(x|z)dz = \log\int q(z)\frac{p(z)}{q(z)}p(x|z)dz \ge \int q(z)\log[\frac{p(z)}{q(z)}p(x|z)]dz = E_q[\log\frac{p(z)}{q(z)}] + E_q[\log p(x|z)]$ (② and ①)

① we assume Gaussian obs model: $\log p(x|z) = \log N(x; G_\theta(z), \eta I) = \log[\frac{1}{(2\pi\eta)^{D/2}}\exp(-\frac{1}{2\eta}\|x - G_\theta(z)\|^2)] = -\frac{1}{2\eta}\|x - G_\theta(z)\|^2 + const$ ⇒ Expected squared error in 重新 constructing x from z. ⇒ reconstruction term

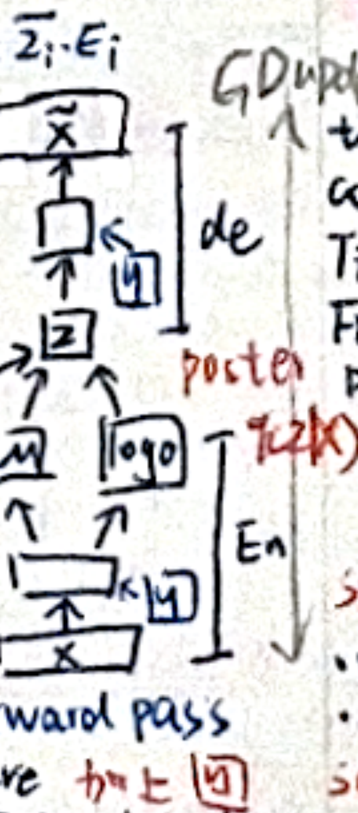
② It is $-D_{KL}(q(z)\|p(z))$, where D_{KL} is the Kullback-Leibler (KL) divergence. $D_{KL}(q(z)\|p(z)) \triangleq E_q[\log\frac{q(z)}{p(z)}]$ ⇒ measure of distance between probability distributions. Typically $p(z) = N(0,1)$, encourage q to be N ⇒ maximize the variational lower bound or variational free energy.

$\log p(x) \ge F(\theta, q) = E_q[\log p(x|z)] - D_{KL}(q\|p)$ (lower bound) ↑
$\log p(x) - F(\theta, q) = D_{KL}(q(z)\|p(z|x))$ ⇒ let q to be as close to posterior distribution $p(z|x)$
①: $E_q[\log p(x|z)] = -\frac{1}{2\sigma^2}E_q[\|x - G_\theta(z)\|^2] + const.$
$z^* = \arg\min_z\|x - G_\theta(z)\|^2$ ⇒ q is point mass ⇒ infinite KL, so KL force q to spread out.

reparameterization trick: to fit q, assign it a para form $q(z) = N(z; \mu, \Sigma)$ where $\mu = (\mu_1, \mu_2, \dots \mu_K)$, $\Sigma = diag(\sigma_1^2, \sigma_2^2, \dots, \sigma_K^2)$. For Gaussian q, $z_i = \mu_i + \sigma_i\epsilon_i$, where $\epsilon_i \sim N(0,1)$ ⇒ $\bar{\mu}_i = \bar{z}_i$, $\bar{\sigma}_i = \bar{z}_i\epsilon_i$

Decoder:
• Fit q to approx the posterior $p(z|x)$ for the current x by doing many GD on F • update decoder para θ with GD on F ⇒ expensive to iterate every exp

Encoder Amortization: amortize the cost of inference by learning an inference network which predicts (μ, Σ) as a function of x • The output are μ and $\log\sigma$ (ensure σ > 0) • If σ ≈ 0, the network compute z deterministically, But KL encourage σ > 0. So in general z will be noisy. The $q(z|x)$ emphasis q depends on x, it is not a conditional prob.



unlike Autoregressive, generation only require 1 forward pass

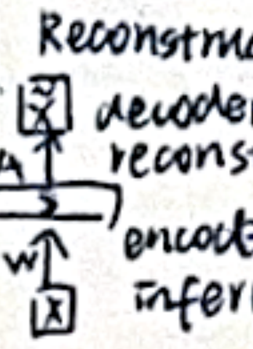
class-conditioned VAE - z can focus on learn other feature 加上 [y]
• By vary two latent dim (i.e. dim of z), hold y fixed ⇒ visualize latent space
• By vary y, hold z → solve image analogies (gen new style) • Interpolation

Polysemanticity 多义性 max activation. **Superposition hypothesis**: neurons are polysematic bc model try to represent more than neurons

Sparse coding: sparse - any particular feature activates only rarely. basis function $\{a_k\}_{k=1}^K$ combined to model each patch. Each patch is approximated as a linear combination of a small # of basis functions (Dictionary). over complete representation. $K > D$ $x \approx \sum_{k=1}^K s_k a_k = As$ (sparse vector)
$\min_s \frac{1}{2}\|x - As\|^2 + \beta\|s\|_1$, encourage sparseness. tradeoff recon error vs sparse.
learn dictionary: $\min_{\{s_i\}, A}\sum_{i=1}^N \frac{1}{2}\|x - As_i\|^2 + \beta\|s_i\|_1$ s.t. $\|a_k\|^2 \le 1$ for ∀k
• normalization constraint on a_k: not s small, but sparse
• reconstruction term can be $\|X - AS\|_F^2$, S contain all s_i

Sparse Auto Encoder: s - sparse vector, alternating minimization. impose non negative constraint on s, s_{-j} (the assignments of all the coefficients except for s_j), the optimal value for s_j is $s_j = \max\left(0, \frac{(x - A_{-j}s_{-j})^T a_j - \beta}{a_j^T a_j}\right)$ linear function of x followed by ReLU

Amortize the inference, Train a ReLU layer $s = \max(0, Wx + b)$. Reconstruction $\hat{x} = As$ $x \to s \to \hat{x}$



• This is just a ReLU auto Encoder with an over complete hidden layer • object: square error of reconstruction + L_1 penalty on s. Interpretable higher level features • Dictionary vectors are constrained to unit form $\|a_j\|^2 \le 1$

Deep reinforcement learning — Policy gradient

Markov decision Process: Assumption: all relevant info is encap in the current state. policy, transition, reward ind. of past given curr • fully observable environment • rollout / trajectory $\tau = (s_0, a_0, \dots s_T, a_T)$
$p(\tau) = p(s_0)\pi_\theta(a_0|s_0)p(s_1|s_0, a_0)\cdots p(s_T|s_{T-1}, a_{T-1})\pi_\theta(a_T|s_T)$
$r(\tau) = \sum_{t=0}^T r(a_t, s_t)$ $R = E_{p(\tau)}[r(\tau)]$ transition, Reward sto

Reinforce: max expected return R ① sample a rollout τ ② calcul gradient ③ adjust policy (θ - policy parameter)
$\partial E_{p(\tau)}[r(\tau)]/\partial\theta = \frac{\partial}{\partial\theta}\sum_\tau r(\tau)p(\tau) = \sum_\tau r(\tau)\frac{\partial}{\partial\theta}p(\tau) = \sum_\tau r(\tau)p(\tau)\frac{\partial\log p(\tau)}{\partial\theta} = E_{p(\tau)}[r(\tau)\frac{\partial}{\partial\theta}\log p(\tau)]$ compute by sampling rollout
Instead of calculate true E over all τ, we approx by sample ind. τ. large reward, make τ more likely. less → less.
$\frac{\partial\log p(\tau)}{\partial\theta} = \frac{\partial}{\partial\theta}\log[p(s_0)\prod_{t=0}^T\pi_\theta(a_t|s_t)\prod_{t=1}^T p(s_t|s_{t-1}, a_{t-1})] = \frac{\partial}{\partial\theta}\log\prod_{t=0}^T\pi_\theta(a_t|s_t)$ (init state dist., transition dist.)
⇒ It make all action in that τ more likely ⇒ no **Credit Assignment**: which action contributed most to the final reward

"reward to go" ① Sample a rollout $\tau = (s_0, a_0, s_1, \dots, s_T, a_T)$ ② For $t = 0, 1, \dots, T$: $r_t(\tau) = \sum_{k=t}^T r(a_k, s_k)$ $\theta \leftarrow \theta + \alpha r_t(\tau)\frac{\partial}{\partial\theta}\log\pi_\theta(a_t|s_t)$

optimizing discontinuous objectives (reward discontinuous 0/1)
$\theta \leftarrow \theta + \alpha r(a,t)\frac{\partial}{\partial\theta}\log\pi_\theta(a|x) = \theta + \alpha r(a,t)\frac{\partial}{\partial\theta}\log y_a = \theta + \alpha r(a,t)\sum(a_k - y_k)\frac{\partial}{\partial\theta}z_k$
z_k: k class logit, $y = softmax(z)$. t - integer target, t_k - one hot. a_k: sampled action of class k (0 or 1). y_k: predicted prob of class k
vs LoR: $\theta \leftarrow \theta + \alpha\sum_k(t_k - y_k)\frac{\partial}{\partial\theta}z_k$ diff ① R use sampled action, LoR use target. ② R scale the update by $r(a,t)$ → binary $\theta \leftarrow \theta + \alpha r(a,t)\frac{\partial}{\partial\theta}\log\pi_\theta(a|x)$

Reward Baseline: Add a const offset to the reward, and we get an equivalent optimization problem. ⇒ We subtract a baseline b from reward without biasing the gradient ⇒ good reward positive, bad negative ⇒ $E[r(\tau)]$ a good choice of baseline. clear grad signal

Tricks • Natural Policy gradient • estimated future return • TRPO PPO
backprop & GD • do credit assignment. Reinforce: not do CA ⇒ random walk
Policy gradient • handle discontinuous cost functions • no need of explicit model of the env (i.e. reward, dynamic are black box) model free RL. sample inefficient ↑

RL from Human Feedback
• **Bradley-Terry model**: latent score function r^* from human (Pairwise) $Pr(choose\ \tau_1) = \exp(r^*(\tau_1)) / (\exp(r^*(\tau_1)) + \exp(r^*(\tau_2)))$ 2 trajectory
Boltzmann rationality: human make stochastic choices that tend to favor their preferred outcome. Assume: ① Existence of r^* ② consistency over time ③ form of function σ ④ no uncertainty
$\tau_{i,k} \to (\tau_{i,1}, \tau_{i,2}) \to y_i$ **Preference model** $r_\eta(\tau) = \sum_t r_\eta(s_t, a_t, s_{t+1})$
Fit η to maximize the likelihood of all the observed rollouts and preference labels $\{(\tau_i, y_i)\}_{i=1}^N$, $\arg\max_\eta\log\prod_{i=1}^N p(y_i|\tau_{i,1}, \tau_{i,2}) = \arg\min_\eta\sum_{i=1}^N -\log\frac{\exp(r_\eta(\tau_{i,y_i}))}{\exp(r_\eta(\tau_{i,1})) + \exp(r_\eta(\tau_{i,2}))}$ (Actual pref, Pair of rollout)
Supervise cross entropy loss. Reward Hacking
• once PM is fit, use it to generate reward signal
• use Policy gradient to optimize θ to max $E_{p(\tau)}[r_\eta(\tau)]$ fast. samples are cheap because they use the PM, not from human • PM will only be correct for τ similar to those in training set.
Partial sol: train η and θ jointly, continue to add new τ to the training set and update PM. **Exploration Problem**: How an agent can deliberatly collect experience that will help it learn. Iterative Improvement

RLHF for LLMs: supervised fine tuning: collect prompt & response, train on it. ⇒ build dataset & how to tell bad response. Pretrain (self supervise generate text) → long ... HHH: helpful, harmless, honest. Elo score. outcome based reward. capacity, grammar, predict next. RL from AI feed back (no infinite regress). fine tune + supervised, RL, RLHF, custom. Align model with Human pref / App