Deep learning = learn the feature & model from data
old ML = create a program that learn from examples
Design the features by hand.

Supervised Learning = labeled examples of correct behavior i.e. ground truth input/output
Reinforcement Learning = learning system receives a reward signal, tries to learn to maximize the reward system
Unsupervised Learning = no labeled examples, looking for interesting pattern
NN can learn useful representation in supervised, unsupervised, or reinforcement learning

es generate image, GPT2 (next world) robot

Linear Model

linear regression

$$X=\begin{pmatrix}\vec{x}_1^{(1)} & \cdots & \vec{x}_D^{(1)}\\ \vdots & & \vdots\\ \vec{x}_1^{(N)} & \cdots & \vec{x}_D^{(N)}\end{pmatrix} \quad \vec{t}=\begin{pmatrix}\vec{t}^{(1)}\\ \vdots\\ \vec{t}^{(N)}\end{pmatrix}$$

$$y^{(i)}=\vec{w}^T\vec{x}^{(i)}+b=(\vec{w}_1\ \cdots\ \vec{w}_D)\begin{pmatrix}\vec{x}_1^{(i)}\\ \vdots\\ \vec{x}_D^{(i)}\end{pmatrix}+b$$

loss: $L(\vec{y}^{(i)},\vec{t}^{(i)})=\frac{1}{2}(\vec{y}^{(i)}-\vec{t}^{(i)})^2$

cost: $\mathcal{E}(\vec{w})=\frac{1}{N}\sum_i L(\vec{y}^{(i)}-\vec{t}^{(i)})$

$$\vec{y}=X\vec{w}+b\vec{1}=\begin{pmatrix}\vec{x}_1^{(1)} & \cdots & \vec{x}_D^{(1)}\\ \vdots & & \vdots\\ \vec{x}_1^{(N)} & \cdots & \vec{x}_D^{(N)}\end{pmatrix}\begin{pmatrix}\vec{w}_1\\ \vec{w}_2\\ \vdots\\ \vec{w}_D\end{pmatrix}+\begin{pmatrix}b\\ b\\ \vdots\\ b\end{pmatrix}=\begin{pmatrix}\vec{y}^{(1)}\\ \vdots\\ \vec{y}^{(N)}\end{pmatrix}$$

$$\mathcal{E}(\vec{w})=\frac{1}{2N}\|X\vec{w}+b\vec{1}-\vec{t}\|^2=\frac{1}{2N}(\dots)^T(\dots)$$

$$=\frac{1}{2N}\|X\vec{w}-\vec{t}\|^2 \qquad \frac{\partial\mathcal{E}}{\partial\vec{w}}=\frac{1}{N}X^T(X\vec{w}-\vec{t})$$

Gradient Descent

① Direct Solution $\nabla_w\mathcal{E}(\vec{w})=0 \Rightarrow \vec{w}=(X^TX)^{-1}X^T\vec{t}$

② Gradient Descent Update Rule: $\vec{w}\leftarrow\vec{w}-\alpha\nabla_{\vec{w}}\mathcal{E}(\vec{w})$
Direction = negative of gradient's sign
size = proportional to gradient's magnitude

③ GD update rule for Linear Regression

$$\vec{w}\leftarrow\vec{w}-\frac{\alpha}{N}X^T(X\vec{w}-\vec{t})$$

$$\vec{w}\leftarrow\vec{w}-\frac{\alpha}{N}\sum_{i=1}^{N}x^{(i)}(\vec{w}^T\vec{x}^{(i)}-\vec{t}^{(i)})$$

④ more efficient: Each GD cost $O(ND)$, rather than $O(D^3)$ for matrix inversion. much cheaper if D is large (i.e. High dimensional data)

Feature Maps = learn non-linear relation using LR

① Polynomial feature mapping $y=w_0+w_1x+\dots+w_Mx^M$
$\psi(x)=[1\ x\ \cdots\ x^M]^T$ $y=w^T\psi(x)$ is linear in $\psi(x)$

② Generalization: to unseen data; • the degree of polynomial (M) is a hyperparameter • Tune it using validation set
Train → Validation → test
tune parameters ← tune hyperpara ← generalization error

Logistic Regression

- binary linear

Linear model: $z=\vec{w}^Tx$ logistic activation: $y=\sigma(z)=\frac{1}{1+e^{-z}}$
cross Entropy: $L_{CE}(y,t)=-t\log y-(1-t)\log(1-y)$
classification accuracy = discontinut → x GD
LCE: it is a proper scoring rule → opt y is true probability

Softmax regression

linear model: $\vec{z}=W\vec{x}$ Weighted matrix $D\times K$ (classes) K class
softmax activation: $\vec{y}=\text{softmax}(\vec{z})$ $y_k=\frac{\exp(z_k)}{\sum_{m=1}^{K}\exp(z_m)}$
CE loss: $L_{CE}(\vec{y},\vec{t})=-\vec{t}^T\log(\vec{y})$
$\frac{\partial y_k}{\partial z_k}=y_k(1-y_k)$

Multi-layer perceptrons

single neurons are linear classifier and have limited expressive power $\frac{\partial y_k}{\partial z_j}=-y_ky_j$ iff $j\neq k$
XOR not linear separable $\psi(x)=[x_1\ x_2\ x_1x_2]^T$

Not linear separable

Prove by contradiction: • Suppose there is a feasible solution • The average of all transaction of A is the vector (0.25, 0.25, ..., 0.25) • Therefore, we must classify this vector as A • The average of all translations of B is the vector above. Therefore, we must classify this vector as B • contradiction!

convex: $x_1,x_2\in S\Rightarrow\lambda x_1+(1-\lambda)x_2\in S \quad 0\le\lambda\le1$
$\lambda_1x_1+\dots+\lambda_Nx_N\in S,\ \lambda_i>0,\ \lambda_1+\lambda_2+\dots+\lambda_N=1$

Multilayer perceptron

$f((1-\lambda)x_0+\lambda x_1)\le(1-\lambda)f(x_0)+\lambda f(x_1)$

Directed acyclic graph; Feed forward NN

Fully connected layer
(N input unit, M output unit ⇒ M×N weight matrix) $y=\phi(Wx+b)$

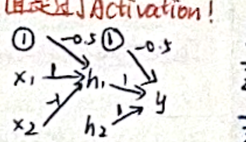
Activation

Relu $\sigma(z)=\max\{0,z\}$ nondifferentiable at 0

Soft Relu (softplus) $\sigma(z)=\log(1+e^z)$
• Does not produce zero gradient for negative inputs
• Differentiable everywhere • computational expansive

Learning XOR 别忘了每一个unit的值是过了Activation!

x_1	x_2	x_1-x_2	h_1	x_2-x_1	h_2	h_1+h_2	y
1	0	1	1	−1	0	1	1
0	1	−1	0	1	1	1	1
0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0



$J(\theta)=\frac{1}{4}\sum(y-t)^2$ $g(z)=\max(0,z)$

$$f(x)=\underbrace{\begin{bmatrix}1 & -2\\ 1 & -2\end{bmatrix}}_{W^T}\max\left\{0,\underbrace{\begin{bmatrix}1 & 1\\ 1 & 1\end{bmatrix}}_{W^T}x+\underbrace{\begin{bmatrix}0\\ -1\end{bmatrix}}_{b}\right\}+0$$

h y

Feature Learning: output layer: linear classification
Data is linear separable in the final features
Hidden layers: learn features

NN's Expressive Power

If all the activation func are linear, the NN can only represent linear function
- not more expressive than linear regression.

Non Linear Network: A multilayer feed-forward NN with non linear activation funcs is a universal function approximator
They can approximate any function arbitrary well
es threshold, logistic, Relu.

Universality for Binary Inputs & targets (记住所有)

- One hidden layer has 2^D hidden units
- The output units uses linear activation
- All the hidden units uses threshold activation

Idea: each hidden units responds to one input config

Back Propogation

For $i=1,\dots,N$
$V_i=\text{function}(\text{Parent}(V_i))$ → Forward Pass: compute loss

$\bar{V}_N=1$
For $i=N-1,\dots,1$ ← Backward pass: derivative
$\bar{V}_i=\sum_{j\in\text{children}(v_i)}\bar{V}_j\frac{\partial V_j}{\partial V_i}$ $\left(\frac{\partial V_j}{\partial V_i}\right)^T\bar{V}_j$ Jacobe vector

$x\to y_1,y_2\to z$ $\frac{dz}{dx}=\frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x}+\frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x}$
Parent → child $\bar{x}=\bar{y}_1\frac{\partial y_1}{\partial x}+\bar{y}_2\frac{\partial y_2}{\partial x}$

L_2 Regularized Regression

Forward:	Backward:
$z=wx+b$	$\overline{L_{reg}}=1$
$y=\sigma(z)$	$\bar{R}=\overline{L_{reg}}\lambda$
$L=\frac{1}{2}(y-t)^2$	$\bar{L}=\overline{L_{reg}}$
$R=\frac{1}{2}w^2$	$\bar{y}=\bar{L}(y-t)$
$L_{reg}=L+\lambda R$	

multi-layer perceptron

Forward:
$z_i=\sum_j w_{ij}^{(1)}x_j+b_i^{(1)}$
$h_i=\sigma(z_i)$
$y_k=\sum_i w_{ki}^{(2)}h_i+b_k^{(2)}$
$L=\frac{1}{2}(y_k-t_k)^2$

Backward:
$\bar{L}=1$
$\bar{y}_k=\bar{L}(y_k-t_k)$
$\overline{w_{ki}^{(2)}}=\bar{y}_kh_i$ $\overline{b_k^{(2)}}=\bar{y}_k$
$\bar{h}_i=\sum_k\bar{y}_kw_{ki}^{(2)}$ $\overline{w_{ij}^{(1)}}=\bar{z}_ix_j$
$\bar{z}_i=\bar{h}_i\sigma'(z_i)$ $\overline{b_i^{(1)}}=\bar{z}_i$

vectorized

$\bar{z}_j=\sum_k\bar{y}_k\frac{\partial y_k}{\partial z_j}$
$\bar{z}=\frac{\partial y}{\partial z}^T\bar{y}$ $\frac{\partial y}{\partial z}$ (Jacobian) $=\begin{pmatrix}\frac{\partial y_1}{\partial z_1} & \cdots & \frac{\partial y_1}{\partial z_n}\\ \vdots & & \vdots\\ \frac{\partial y_m}{\partial z_1} & \cdots & \frac{\partial y_m}{\partial z_n}\end{pmatrix}$ $m\times n$

z1 z2 z3 → y1 y2 y3 (fully connected) $z\to y$

Example:
$z=Wx$ $\frac{\partial z}{\partial x}=W$ $\bar{x}=W^T\bar{z}$
$y=\exp(z)$ $\frac{\partial y}{\partial z}=\begin{pmatrix}\exp(z_1) & \cdots & 0\\ \vdots & & \vdots\\ 0 & \cdots & \exp(z_D)\end{pmatrix}$ $\bar{z}=\exp(z)\circ\bar{y}$ element wise

MLP vectorized Back Prop

Forward:	Back:
$z=W^{(1)}x+b^{(1)}$	$\bar{L}=1$
$h=\sigma(z)$	$\bar{y}=\bar{L}(y-t)$
$y=W^{(2)}h+b^{(2)}$	$\overline{W^{(2)}}=\bar{y}h^T$ $\overline{b^{(2)}}=\bar{y}$
$L=\frac{1}{2}\|y-t\|^2$	$\bar{h}=W^{(2)T}\bar{y}$ $\bar{z}=\bar{h}\circ\sigma'(z)$
	$\overline{W^{(1)}}=\bar{z}x^T$ $\overline{b^{(1)}}=\bar{z}$

$x\to z\to h\to y\to L$ ($W^{(1)}$, $b^{(1)}$, $W^{(2)}$, $b^{(2)}$, t)

computational cost

Forward: 1 add-multiply op per weight
Back: 2
For a multi-layer Perceptron, the cost is linear in # of layer, and quadratic in # of units per layer

Autodifferentiation

compute derivatives exactly & efficiently

Autodifferentiation

Back prop is a special case of auto diff

Two types: Forward Mode: propagate derivative from inputs to outputs more efficient when #input < # outputs
Reverse mode: outputs → inputs, #In > #out

Implementation: • convert the program to primitive operations • Build the computational graph • Implement VJP for each primitive op
• perform the backward pass (t-x÷-)

Build computational graph

Autograd build by tracing the forward pass computation

$y=\text{logistic}(z)$ $z=1.5$

node z: value: 1.5, fun: None, parent: []
node t1: value: -1.5, fun: negative, parent: [z]
node t2: value: 0.223, fun: exp, parent: [t1]
node y

Vector Jacobian Products

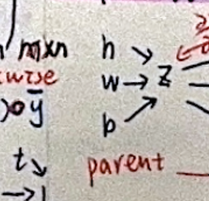
Implementation:
For each operation, define a VJP for each of its arguments ans = operation(argument)
VJP is a function that takes as inputs:
- the output error signal g
- the value of the node ans
- the arguments to the op

and returns:
the input error signal for corresp argu

es ans = log(x) $VJP(g,x)=g/x$

```
def vjp(log, lambda g, ans, x: g/x)
```

Perform Backprop



① Z receives message from each of its children, VJP corresponding to their
② Z sums all the message togeter to compute its error signal ③ Z computes the VJP w.r.t. each of its arguments and sends them as messages to each of its parent nodes

$\bar{z}=\frac{\partial r}{\partial z}\bar{r}+\frac{\partial s}{\partial z}\bar{s}+\frac{\partial t}{\partial z}\bar{t}$

np.dot
np.inner $R^{n\times2}$
np.outer

$\tanh'(x)=1-\tanh^2(x)$ $\tanh(x)=2\sigma(2x)-1$
$\left(\frac{\partial y}{\partial z}\right)_{ij}=\frac{\partial y_i}{\partial z_j}=\begin{cases}\tanh'(z_i) & \text{if } i=j\\ 0\end{cases}=\text{diag}(\tanh'(z))$
$\text{Duf}(x,y)=\nabla f(x,y)\cdot u=\|\nabla f(x,y)\|_2\cos\theta$
MLE $\prod_{i=1}^{n}P(t^{(i)}|x^{(i)},w)$ $L_{CE}=-\sum_n^N\sum_k^K t_{nk}\log($

Distributed Representation

Language modelling → Predicting next word

Build a generative model:
- the observation model, $P(\alpha|s)$
- The prior, $p(s)$
- Infer $P(s|\alpha)$ using Bayes rules (posterior prob)

Learn a distribution $P_\theta(S)$ of sentences
Given some observed sentences $s^{(1)}, \dots, s^{(N)}$
learn θ using maximum likelihood that maximize the probability of the observed s.

Prob. of sentences: $P(s^{(1)}, \dots, s^{(N)}) = \prod_{i=1}^{N} P(s^{(i)})$

use log to prevent prob becoming vanishingly small:

$$\log \prod_{i=1}^{N} P(s^{(i)}) = \sum_{i=1}^{N} \log P(s^{(i)})$$

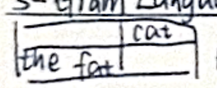
$P(s) = P(w_1, \dots, w_T) = P(w_1)P(w_2|w_1)\dots P(w_T|w_1, \dots, w_{T-1})$

Markov Assumption: The probability of the next word depends on a fixed # of prev words.

memoryless model: $P(w_t|w_1, \dots w_{T-1}) = P(w_t|w_{t-3}, w_{t-2}, w_{t-1})$

Auto Regressive Model: supervised prediction prob.
Auto: use its prev output/values to predict the next value. The model is self referential
Regressive: regression. predict the current value based on past values.

3-Gram Language Model



$P(W_3 = \text{Cat} | W_1 = \text{the}, W_2 = \text{fat}) = \frac{\text{count(the fat cat)}}{\text{count(the fat)}}$

Limitation: (now) many N-grams are rarely seen in training data
Data sparsity = N↑, # of possible N-gram exp↑
sol: • shorter context • smooth prob by adding imaginary count
• combine predictions from N-gram models with diff value of N

N-Gram is a localist representation: Each N-gram is treated as a distinct & independent entity
The info is not shared btw N-gram es the dog sit / the cat sit

Neural Language Model

Distributed Representation: Words are represented as vectors in a continuous space.
word → attribute → next word

loss function: $-\sum_{i=1}^{V} t_i \log y_i$ ← prediction.
↑ true (0 for wrong, 1 for correct)

t_i: one hot encoded target
y_i: the predicted probability

Bengio's Neural Language model:
Input layer → Embedding layer → Hidden layer → output layer.
previous k word — each word vector — NN — Prob distrib of next word.

Localist: a piece of info is store in one place
Distributed: info is shared between related entities
• more complex and difficult to interpret • capture more nuanced info
• more robust to damage or noise • large dataset

Global vector (GloVe) Embeddings

Learning word embeddings from global word-word co-occurences.
V is the vocabulary size (the frequency with which two words appear nearby)

Co-occurence Matrix X (VxV matrix)

$$X_{V\times V} = R_{V\times k} \tilde{R}^T_{k\times V} \qquad x_{ij} = r_i^T \tilde{r}_j$$

Cost function $J(R) = \sum_{ij} f(x_{ij}) (r_i^T \tilde{r}_j + b_i + \tilde{b}_j - \log x_{ij})^2$
($r_i^T \tilde{r}_j$ → x_{ij}; $\log x_{ij}$ → actual)

weights: $f(x_{ij}) = \begin{cases} \left(\frac{x_{ij}}{x_{max}}\right)^\alpha, & \text{if } x_{ij} < x_{max} \\ 1, & \text{if } x_{ij} \geq x_{max} \end{cases}$
count → more balanced

Why log count • Emphasize the diminishing importance of very high counts • reduce the range of counts, learning more stable
• emphasize the relative diff in counts

Pros: • capture global statistics
• capture semantic relationship between words
• can be trained relatively quickly on large dataset
• Pretrained Glove Embeddings available

Cons: • static embedding — words representation does not change based on its context.
• The global co M requires significant memory for storage & computation • Struggles with words not seen during training

optimization

Local minima

convex: any local is global
strong convex: unique global
non convex: mult local, not global

NN are not convex:
weight space symmetry
Suppose we are at a local min θ
swap two hidden units with its weights and biases and get θ'
θ' must also be local min
⇒ random start & local min is fine.

Saddle point

The gradient (1st deriva) = 0
The Hessian matrix has both +,- eigenvalues

es ① two hl, h2 have same incoming & outgoing weights
aft GD, still same ⇒ forever; ② all init w-b = 0
all w be identical ⇒ no learning
⇒ Initial the weights to small random values.

plateaux

saturated units — output value close to extreme → vanishing gradients
Deadunits — close to zero ⇒ zero gradient

⇒ choose the scale of random init of the weights so that activation are in the middle of their dynamic range
• Xavier: initialize the weights s.t. variance of weights activations is same across every layer

⇒ Relu: initial the biases to a small + value e.g. 0.1

Badly conditioned Curvature

Gradient along w_1 is Large
GD takes large steps
we should take small steps

⇒ avoid ravine: ① normalize your inputs to zero mean and unit variance
② Batch normalization explicitly center each activation. speed up training. so that activation have zero mean and unit variance.

⇒ Momentum

$m_{k+1} \leftarrow \mu m_k + \alpha \frac{\partial J}{\partial \theta}$
$\theta \leftarrow \theta - m_{k+1}$

μ - damping parameter
How quickly the contribution of previous gradients exp decay (0.9, 0.99)

Second order information

RMSprop (Root mean Square Propagation)

An exp ma of the squared gradient

$$S_j \leftarrow (1-\gamma) S_j + \gamma \left[\frac{\partial J}{\partial \theta_j}\right]^2$$

$$\theta_j \leftarrow \theta_j - \frac{\alpha}{\sqrt{S_j + \varepsilon}} \frac{\partial J}{\partial \theta_j}$$

Adam (Adaptive Moment Estimation)

$$m_{k+1} \leftarrow \beta_1 m_k + (1-\beta_1) \frac{\partial J}{\partial \theta}$$
$$S_{k+1} \leftarrow \beta_2 S_k + (1-\beta_2) \left[\frac{\partial J}{\partial \theta}\right]^2$$
$$\hat{m}_{k+1} = \frac{m_{k+1}}{1-\beta_1^{k+1}} \qquad \hat{S}_{k+1} = \frac{S_{k+1}}{1-\beta_2^{k+1}}$$
$$\theta_{k+1} \leftarrow \theta_k - \frac{\alpha}{\sqrt{\hat{S}_{k+1} + \varepsilon}} \hat{m}_{k+1}$$

convolutional NN and Image Classification

Convolutional layer

Small region of the image
the weights are shared

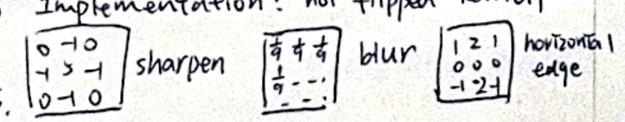
Pros: • Fewer parameters
Effective computation, better generalization
• preserve spatial structure of input
• Translation Invariance: shared weights allows the layer to detect features regardless of their position

convolution Operation

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n) K(i-m, j-n)$$

flipped kernal

Implementation: not flipped kernal



Convolutional Network Structure

Input layer → convolution → Nonlinear Activation → Pooling → Next convolution

convolution is a linear operation
pooling: reduce the dimensions of the feature maps. replace the output with a summary statistics

Benefits of pooling layer • reduce D...

(reduce # of parameters, reduce computational load faster to train. prevent overfitting)
• support translation Invariance:
• Pooling layers expands receptive field.: helps higher-layer cover larger regions and enable deeper layers to interact with more of the input.

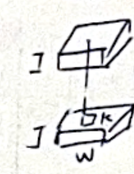
Image classification

Conv Net architecture

Size of a Conv Net

(training for back prop)
• # of units: store activations in memory
• # of weights: for training & testing
• # of connections: complexity of backprop
(3 add-multiply per connection 1 forward 2 backward)

For a fully connected layer with M input, N output units
it has MN connections, MN weights



	Fully connected layer	Convolution layer
# output weights / unit	IWH	IWH
# weights	W^2H^2IJ	K^2IJ
# connections	W^2H^2IJ	WHK^2IJ

Alex Net • Relu activation (faster training & convergence)
• weight Decay (L^2 regularization) prevent overfitting
• Data augmentation (image translate) size of patch
• SGD with momentum (with previous update)
• Dropout randomly set a fraction of input units to prevent overfitting

GoogleNet ↓ # of weights
• No fully connected layer
• Break down convolutions into multiple smaller convolutions

Inception Module
use parallel convolutions with various filter sizes
• capture features at multiple scales with
• 1x1 convolutions reduce dimensions of feature maps
• allows for a deeper network without significaty more computational complexity.

$(A+B)^T = A^T + B^T$
$(rA)^T = rA^T$
$(AB)^T = B^T A^T$
$I^T = I$
$AA^{-1} = A^{-1}A = I$
$(rA)^{-1} = \frac{1}{r}A^{-1}$
$(A^{-1})^{-1} = A$
$(A^T)^{-1} = (A^{-1})^T$

$(AB)^{-1} = B^{-1}A^{-1}$
$I^{-1} = I$
$f(w) = W^TAW + b^TW + y$
$\nabla f = \frac{1}{2}(A^TA)W + b$
$= aw + b$ (symb)
$f = W^TAW$
$\nabla f^2 = 2A$

$\frac{\partial x^Ta}{\partial x} = \frac{\partial a^Tx}{\partial x} = a$
$\frac{\partial a^Txb}{\partial x} = ab^T$
$\frac{\partial x^TBx}{\partial x} = (B+B^T)x$
$\frac{\partial (X-AS)^TW(X-AS)}{\partial S} = -2A^TW(X-AS)$
$\bar{X} = 2W(X-AS)$
$\bar{A} = -2W(X-AS)S^T$

Optimization and Generalization

Stochastic Gradient Descent

batch training: $\frac{\partial \varepsilon}{\partial w} = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial L(y^{(i)}, t)}{\partial w}$ summing over all training examples

SGD: update model parameter based on the gradient calculated from a single, randomly chosen training ex.

BT: moves directly downhill on avg.

SGD: take steps in a noisy direction, but moves downhill

faster update (SGD), more mem / stuck local / guaranteed conv (BT)

Mini Batch Gradient Descent

compute gradients from small, randomly sampled subsets of the training data

- faster convergence
- smoother conv (variance↓)
- handle large dataset
- vectorized operations

mini batch size: S hyperparameter & saturate at larger size

GPU favor larger batch size ⇒ afford more parallelism

larger batch size require more mem, Diminishing returns in convergence improvement vs. computation cost.

(faster conv; local min; GPU saturate) steps

optimizing batchsize: $2^5, 2^6, 2^7, 2^8$. Tune S by plotting validation error against training time for diff S.

- Tune other para before α and S
- use GPU by increase S

learning rate hyper

grid search

- start at a higher learning rate for quick progress
- If the error↑ or oscillates wildly → α↓
- If error↓ consistently, but slowly → α↑
- At the end of mini batch → α↓ to remove fluctuation; when error stop↓ → α↓; loss fluctuate → α↓ (ε vs epoch)

α↓ could cause sudden ↓ in loss at the expense of LT bc a reduction in upward fluctuation than dramatic improve.

decaying α · step decay · exponential decay $\alpha = \alpha_0 e^{-kt}$ · 1/t decay: $\alpha = \alpha_0/(1+kt)$ · use a grid search on log scale (0.1, 0.001..)

SGD + Momentum

contour map · 同条线上相同 · 越密 gradient 越大 · Gradient 上于 contour line 并且指向 increasing function value (GD 指反方向)

Ravine ⇔ steep gradient — take large step — should↓; shallow gradient — take small step — should↑

- Momentum: steep: dampens oscillation, cancel g↑↓; shallow: flat: build up velocity, accumulating small g

$m_{k+1} \leftarrow \mu m_k + \alpha\frac{\partial J}{\partial \theta}$ $\theta \leftarrow \theta - m_{k+1}$

μ: damping parameter, how quickly the prev g decay exp

RMS Prop (Root Mean Square Propagation)

adjust α based on mv avg of the squared gradient

$S_j \leftarrow (1-\gamma)S_j + \gamma\left[\frac{\partial J}{\partial \theta_j}\right]^2$ $\theta_j \leftarrow \theta_j - \frac{\alpha}{\sqrt{S_j+\epsilon}}\frac{\partial J}{\partial \theta_j}$ ($g\uparrow \Rightarrow \alpha\downarrow$)

Adam (Adaptive Moment Estimation)

smooth out the updates

ma of past gradient $m_{k+1} \leftarrow B_1 m_k + (1-B_1)\frac{\partial J}{\partial \theta}$

ma of past squared gradient $S_{k+1} \leftarrow B_2 S_k + (1-B_2)\left[\frac{\partial J}{\partial \theta}\right]^2$ est var of gradients

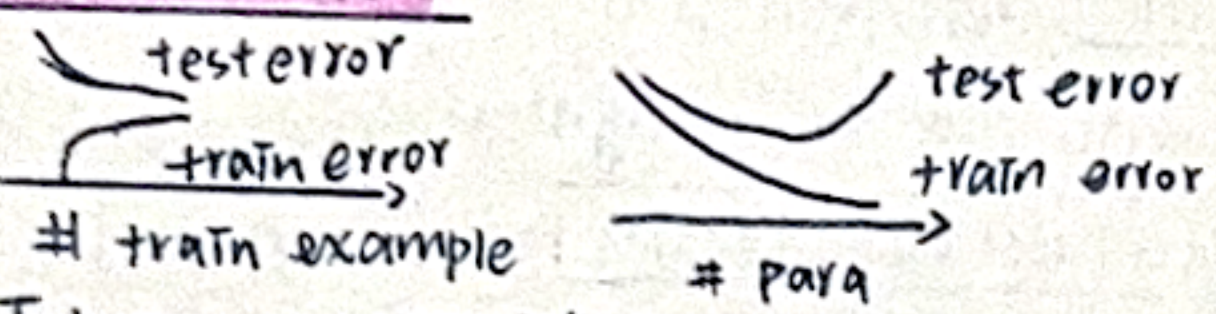
correct the bias in ma: estimates are initial 0 w/o bia, the estimates are sys lower than their true value

$\hat{m}_{k+1} = \frac{m_{k+1}}{1-B_1^{k+1}}$ $\hat{S}_{k+1} = \frac{S_{k+1}}{1-B_2^{k+1}}$

update parameter: $\theta_{k+1} \leftarrow \theta_k - \frac{\alpha}{\sqrt{\hat{S}_{k+1}+\epsilon}}\hat{m}_{k+1}$ ε prevent ÷0

Problems	Diagnostics	Work around
Incorrect gradient	finite diff	Fix, or auto diff
local min	(hard)	random restart
symmetries	visualize W	Initial W randomly
slow progress	slow, linear training curve	α↑, momentum
Instability	cost increases	α↓, momentum
Oscillations	fluctuation in training curve	α↓, momen
Fluctuation		decay α; iterate avg
dead/saturated unit	Activation histogram	Initial scale of W; Relu
ill condition	(hard)	Adam, second order opt

Generalization



Technique to prevent overfitting

① Data Augmentation: Augment on training set, not test

② Reduce para: little bottleneck layer, #connection↓

③ Weight decay (λW→引起noise, Amply express power↓ the influence of individual input features, cause oscillation during training and exacerbate vanishing, exp g)

Regularization $\varepsilon_{reg} = \varepsilon + \lambda R = \varepsilon + \frac{\lambda}{2}\sum_j w_j^2$

$\vec{w} \leftarrow \vec{w} - \alpha\left(\frac{\partial \varepsilon}{\partial \vec{w}} + \lambda\frac{\partial R}{\partial \vec{w}}\right) = (1-\alpha\lambda)\vec{w} - \alpha\frac{\partial \varepsilon}{\partial \vec{w}}$

L^2 reg: $\varepsilon_{reg} = \varepsilon + \frac{\lambda}{2}\sum_j w_j^2$ encourage N 个非0 weight; $\varepsilon_{reg} = \varepsilon + \frac{\lambda}{2}\sum_j |w_j|$ encourage w=0

④ Early stopping: W starts small and grow, nn gradually becomes more non linear. mul model

⑤ Ensembles: Avg or combine the prediction from diff random initialization; diff subset of training data (bagging); diff arch or hyperpara

⑥ Stochastic regularization: inject noise into network

Dropout: randomly deactivate neurons during training with probability P. At test, scale w by 1-p → expected Activation of each neuron.

Drop connect

Batch normalization

The random in SGD updates act like a regularizer.

out size = (in size − K size + 2×pad)/stride + 1
T: os = stride × (input size − 1) + KS − 2×pad + out padding
pad = (KS−1)/2 outpad = int(X%2==1)

Overparameterization & Normalization

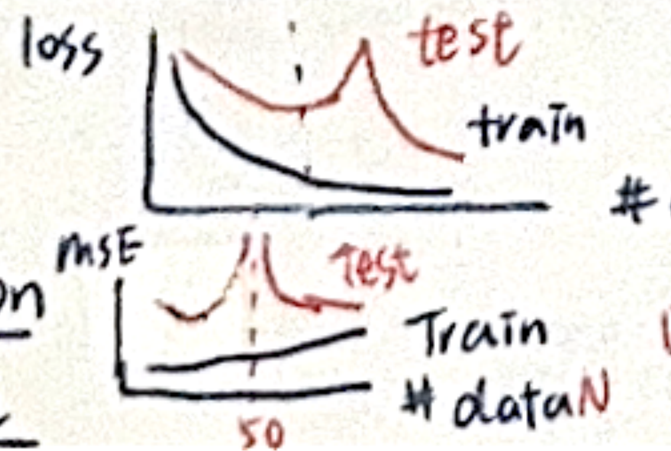
Non convexity & overparameterization

NN can generalize well even overparameterized. Factors beyond capacity control play a crucial role.

Non convexity: many local minima, saddle, plateaus

over para: more para than data points, optima landscape with infinitely many possible global

Double descent



Linear regression

Double descent

Minimum singular values reflects the "compressibility" of the data, can be well-approx by a lower-dim subspace

- Explicit regularization to discourage large weights, reduce model's capacity, effectively smooth out the double descent curve and eliminate the second descent.

Neural Tangent Kernel

capture how network's output changes with tiny weight changes.

$\Delta z = J\Delta W$ $\Delta W = -\alpha\nabla J(W)$

$\Delta z = J\Delta W = J[-\alpha\nabla J(W)]$ $J: \frac{\partial z}{\partial w}$

$\nabla J(W) = \frac{1}{N}J^T\nabla L(z)$

$\Delta z = J\left[-\frac{\alpha}{N}J^T\nabla L(z)\right] = -\frac{\alpha}{N}JJ^T\nabla L(z)$ (JJ^T = K)

When width → ∞, J, K become constant → nn become like a linear model with a fixed kernel

under certain condition, nn is guaranteed to learn the training data perfectly; work well on unseen data

limitation: infinite width network; full batch gradient descent; compute NTK expensive

GD can find a global minimum of the training loss for sufficiently over parameterized nn.

Batch normalization

covariate shift: a Δ in the distribution of the input data

Internal covariate shift: the distribution of activations changes from layer to layer during training

consequences: · slower training · unstable gradient · unstable Gradient · difficulty in hyper tuning · reduced model performance · ↑sensitivity to initialization

Normalizes pre-activations within each mini-batch to have 0 mean and 1 variance

Done for each feature independently

For a mini-batch B of size m, for each unit in a hidden layer:

mean $\mu_B = \frac{1}{m}\sum_i z_i$ normalization $\hat{z}_i = \frac{z_i - \mu_B}{\sqrt{\sigma_B^2+\epsilon}}$

variance $\sigma_B^2 = \frac{1}{m}\sum_i (z_i-\mu_B)^2$ scale & shift $z_i^{BN} = \gamma\hat{z}_i + B$ hyper

$x_1 \to z_1^{(1)} \to h_1^{(1)} \to y_1$; $x_2 \to z_2^{(1)} \to h_2^{(1)} \to y_2$

$\bar{u} = \frac{1}{T}\sum_{t=1}^{T}\bar{z}^t x^t$ $\bar{v} = \frac{1}{T}\sum_{t=1}^{T}\bar{r}^t h^t$ $\bar{w} = \frac{1}{T}\sum_{t=1}^{T}\bar{z}^{t+1} h^t$

Implicit learning rate decay

Tikhonov Regularization $J_{reg}(\vec{w}) = J(\vec{w}) + \frac{\lambda}{2}\|\vec{w}\|^2$

$\vec{w} \leftarrow (1-\alpha\lambda)\vec{w} - \alpha\frac{\partial J}{\partial \vec{w}}$

Batch Norm is Invariant to Weight Scaling

Scale $\vec{w}_j$ by γ>0, preactivation z scale by γ, normalized z will remain same

Effective learning rate

$\hat{\vec{w}}_j = \vec{w}_j/\|\vec{w}_j\|$ $\vec{w}_j^{(k+1)} = \frac{\vec{w}_j - \alpha\nabla J(\vec{w}_j^{(k)})}{\|\vec{w}_j - \alpha\nabla J(\vec{w}_j^{(k)})\|}$ very small

$\hat{\vec{w}}_j^{(k+1)} = \frac{\vec{w}_j^{(k)} - \alpha\nabla J(\vec{w}_j^{(k)})}{\|\vec{w}_j^{(k)}\|} = \hat{\vec{w}}_j^{(k)} - \alpha\|\vec{w}_j^{(k)}\|^{-1}\nabla J(\vec{w}_j^{(k)})$

$= \hat{\vec{w}}_j - \alpha\|\vec{w}_j^{(k)}\|^{-2}\nabla J(\hat{\vec{w}}_j^{(k)})$

(since $\nabla J(\gamma\vec{w}_j^{(k)}) = \gamma^{-1}\nabla J(\vec{w}_j^{(k)})$)

Effective learning rate: $\alpha\|\vec{w}_j^{(k)}\|^{-2}$

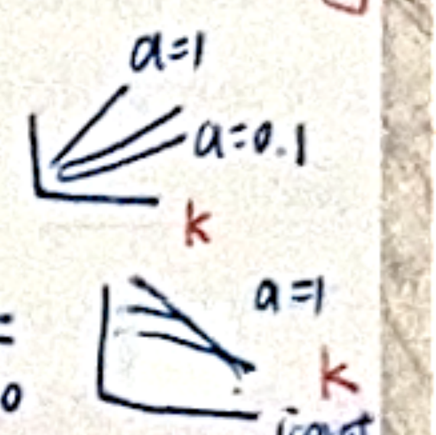
$\|\vec{w}_j^{(k+1)}\|^2 = \|\vec{w}_j^{(k)} + \alpha\nabla J(\vec{w}_j^{(k)})\|^2$ $\vec{w} \perp \nabla J(\vec{w})$

$= \|\vec{w}_j^{(k)}\|^2 + \alpha^2\|\nabla J(\vec{w}_j^{(k)})\|^2$ Weight magnitude grows monotonically

$= \|\vec{w}_j^{(k)}\|^2 + \frac{\alpha^2\|\nabla J(\hat{\vec{w}}_j^{(k)})\|^2}{\|\vec{w}_j^{(k)}\|^2}$

$\|\vec{w}_j^{(k)}\|^2 \propto \sqrt{Hk/k_0}$ for some k_0

Effective learning rate: $\hat{\alpha}_k = \frac{\alpha_0}{\sqrt{Hk/k_0}}$



As weight grows, the effective learning rate shrink

Strategy to counter the LR decay

- exp increase LR schedule
- weight decay (L2 Reg)
- Normalize weight to unit vector after each update

Weight decay

$\varepsilon_{reg} = \varepsilon + \lambda R = \varepsilon + \frac{\lambda}{2}\sum_j w_j^2$

$\vec{w} = (1-\alpha\lambda)\vec{w} - \alpha\frac{\partial \varepsilon}{\partial \vec{w}}$

Adam with L_2 Reg

$g_t = \frac{\partial J}{\partial \theta}$ adapts α for each para individually

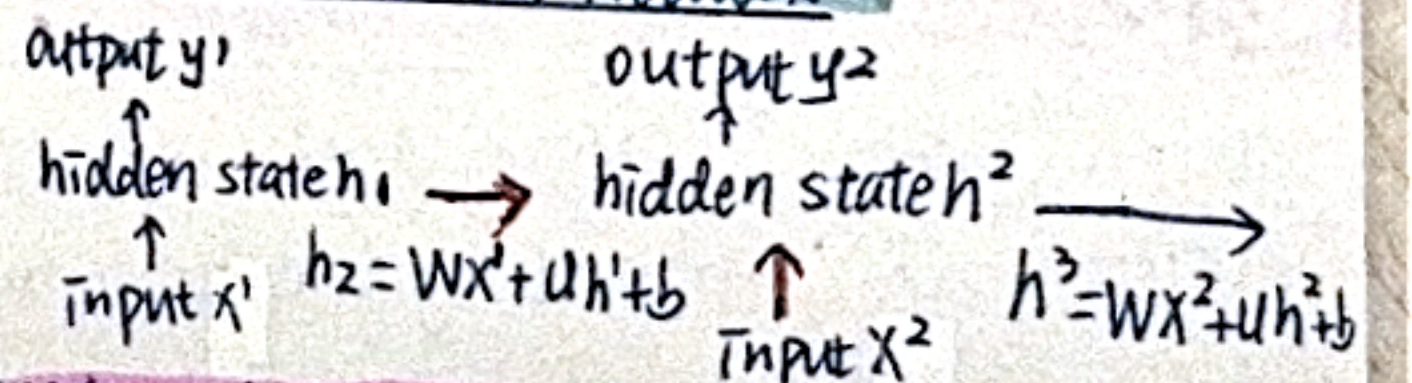
add weight decay $g_{twd} = g_t + \lambda W_t$

avg of gradient $m_{k+1} \leftarrow B_1 m_k + (1-B_1)g_{twd}$

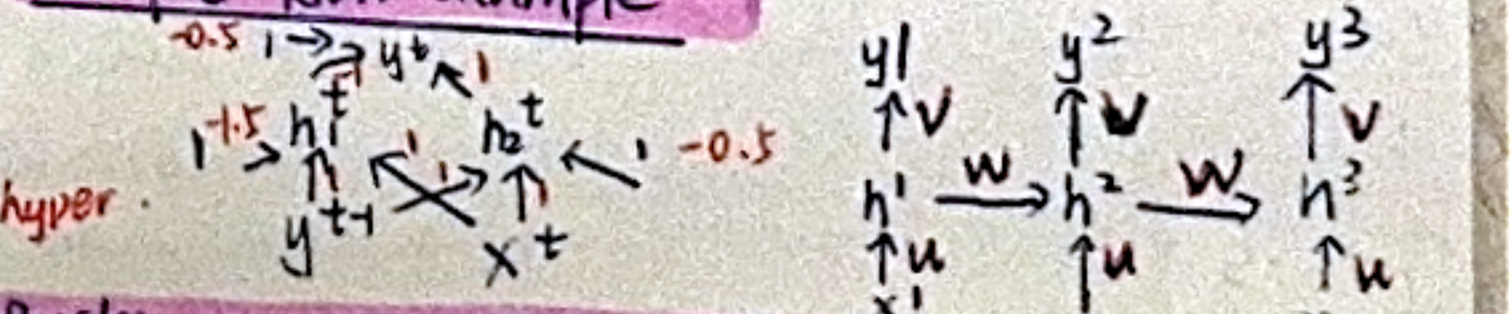
avg of squared grad: $S_{k+1} \leftarrow B_2 S_k + (1-B_2)g_{twd}^2$

last step: $\vec{w} \leftarrow (1-\lambda)\vec{w} - \alpha\,\partial\varepsilon/\partial\vec{w}$

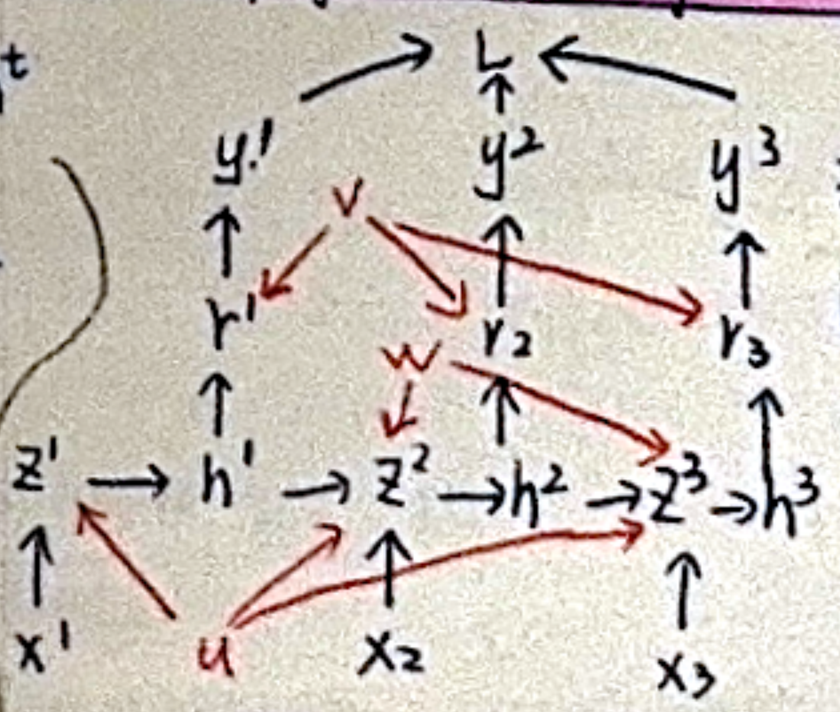
Recurrent Neural Network



Sample RNN example



Backpropagation Through time



Forward:

$z^t = wh^{t-1} + ux^t$

$h^t = \phi(z^t)$

$r^t = vh^t$ $y^t = \phi(r^t)$

$L(y^1, y^2, y^3, \ldots, t^1, t^2, t^3\ldots)$

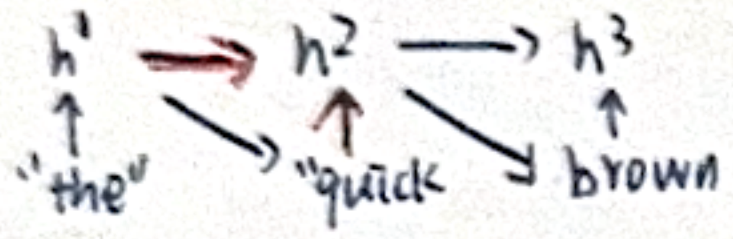
Back: $\bar{L} = 1$ $\bar{y}_t = \bar{L}L'(y_t)$ $\bar{r}_t = \bar{y}^t\phi'(r^t)$

$\bar{h}^t = \bar{r}^t\bar{v} + \bar{z}^{t+1}w$ $\bar{z}^t = \bar{h}^t\phi'(z^t)$

Language Modelling using RNN

Generating using model output as input next step

Training the model with techer forcing: use Target output from the training data as input



• word boundaries

Challenge: • vocabulary scale • Out of vocabulary

character level challenge: ↑ sequence length

• multiplicate interactions to model character combinations - typically require deeper network

limitation • maintain local grammer structure

• Lack global semantic coherence

Neural Machine Translation

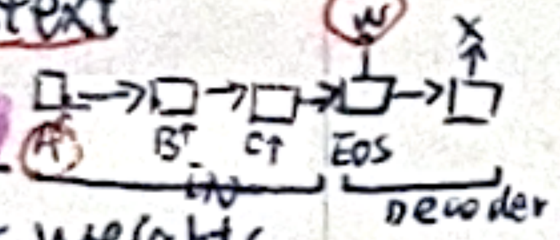
- word by word

Challenge: word orders, sentence length

• need future words to resolve ambiguity

• Translation require full sentence context

Sequence to Sequence Architecture



Two distinct neworks with different weights

Encoder: process entire input sentence, compressing it into a code vector

Decoder: generate French translation word by word

• handle different sentence length naturally

• learns language specific word order

• capture complete sentence context

Exploding and Vanishing gradients

why RNN suffer ↑? During BP through time, gradients are multiplied by the same weight matrix repeatedly. This repeated multiplication causes the gradients to explode or vanish.

BP: Linear operations compound multiplicatively

• thin line between vanishing and exploding gradients

FP: • squashes the activations

• with non-linear activation: functions

• protection against exp growth.

Iterated Logistic Map $f(x) = rx(1-x)$

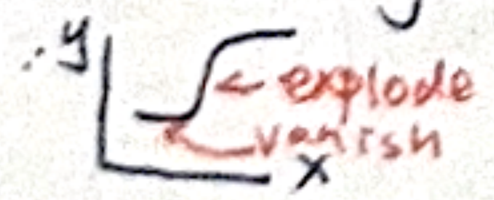
composing the same function many times can result in highly non linear behavior.

Attractors 

• Inside each color, g vanish

• At boundary, g explode

RNN must balance stability and flexibility

tanh Activation 

gradient.

use mini batch during training helps stablize

Mitigating Exploding Vanishing Gradient

① Gradient clipping

If $\|\vec{g}\| > \eta$: $\vec{g} = \frac{\eta \vec{g}}{\|\vec{g}\|}$ rescale the g

Tradeoff: bias vs stable

② Trick for sequence to sequence model

• reverse the input sequence ABC → CBA

• shorten the path between matching input-output pairs

• Progressive learning: Learn short term dependencies before long term ones.

③ Identity RNN than vanilla RNN

won't explode g • initialize weight matrix to identity matrix • use ReLu activation function

• Learn much longer dependency • classify MINST by processing the image one pixel at a time

④ Long term Short term Memory (LTSM)

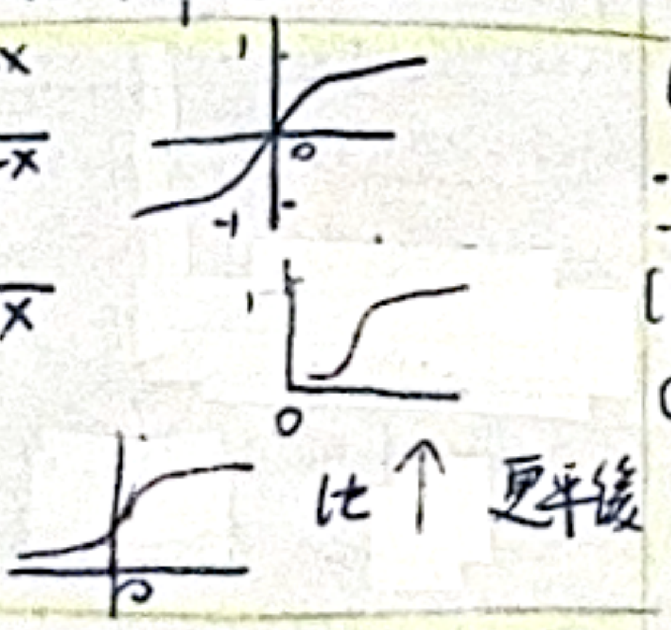
$C_{t+1} = f(\text{forget gate})\, C_t + i(\text{input gate})\, \text{new input}$

1. $i=0, f=1$: remember
2. $i=1, f=1$: add new input to memory
3. $i=0, f=0$: erase reset mem to 0
4. $i=1, f=0$: overwrite replace mem with new input.

$\tanh = \frac{\sinh}{\cosh} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

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

$\text{Softmax}(z_i) = \sigma(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$



lt ↑ 更平缓

1. compute gate values

$$\begin{pmatrix} i_t \\ f_t \\ o_t \\ g_t \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} y_t \\ h_{t-1} \end{pmatrix}$$

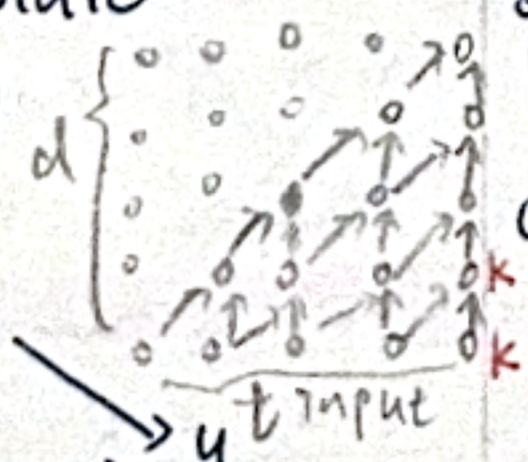
Input modulation → g_t

2. Update the memory cell state

$C_t = f_t \circ C_{t-1} + i_t \circ g_t$

3. Compute the hidden state

$h_t = o_t \circ \tanh(C_t)$



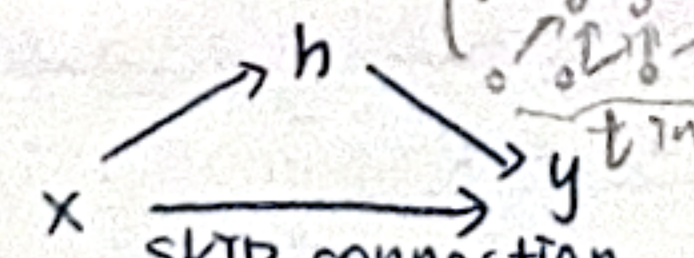
Deep residual networks

residual blocks

$z = W^{(1)}x + b^{(1)}$

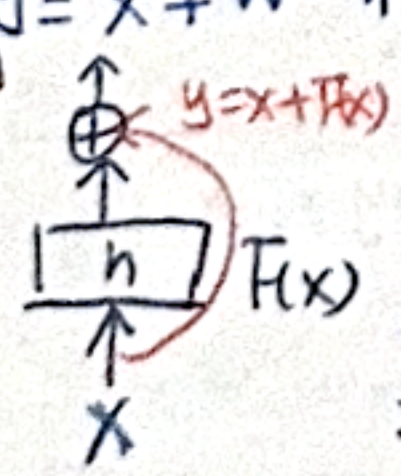
$h = \phi(z)$

$y = x + W^{(2)}h$



$y = x + F(x)$

$= x + \text{transformed}(h)$



Address vanishing gradient:

1. If opt mapping is identity, $F(x)=0$
2. If $\frac{\partial F}{\partial x}$ vanishing, gradient propogate backward through skip connection $\bar{x}^{l} = \bar{x}^{l+1} + \bar{x}^{l+1}\frac{\partial F}{\partial x} = \bar{x}^{l+1}(H\frac{\partial F}{\partial x})$

Autoregressive Models

RNN: computational cost and Parallelism

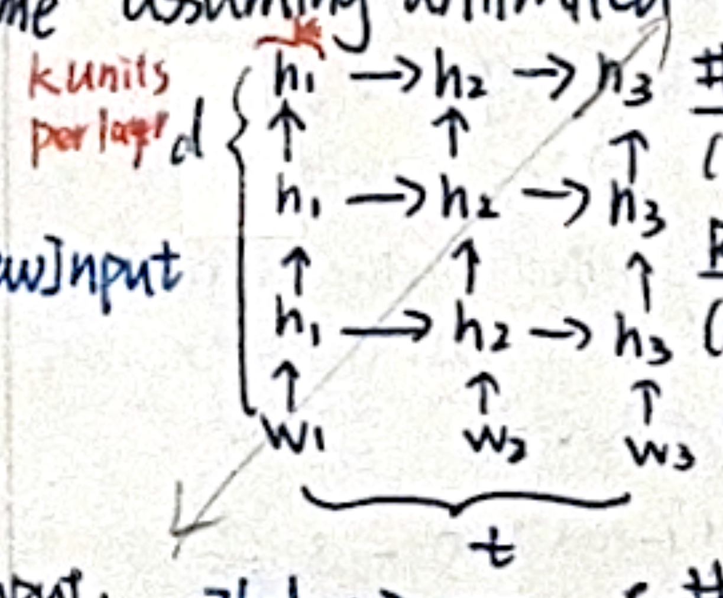
Flops # of add-multiply operations for the forward and backward passes, determined by the number of connections

Memory **training**: determined by the number of activations, since they need to be stored for back prop

Test time: determined by # of parameters

Efficient due to weight sharing across time steps

Sequential complexity: # of sequential operations required to accomplish some task assuming unlimited parallelism.



# of flops: $2k^2dt$ connections

RNN train mem: kdt (grow to rem long term info)

Generation time

• activation to store kd

• # parameters k^2d

• # of sequential op to perfom 1 gradient update $2(d+t)$ (BP+FP)

(we compute activation & weight in parallel)

- # of sequential op to generate data dt

(must compute output for time t before compute any activation for time t+1)

Sequential complexity is bad for long seq

The hidden state must store all critical info about the input sequence, redundantly across many time steps

Convolutional Autogressive Model

① causal convolution: each prediction only depends on earlier inputs in the seq.

Implemented by masking or setting zero weights on connections to future inputs

k: # of features per layer w: kernal width

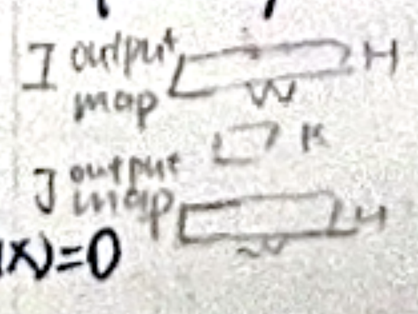
	CNN	RNN
connections	wk^2dt	k^2dt
para #	wk^2d	k^2d
activation	kdt	kdt
Seq Com (train)	d^*	$d+t$
SeqCom (generat)	dt	dt

$\frac{\partial(XW)}{\partial W} = X^T$

kernal size: x

Padding $(x-1)/2$

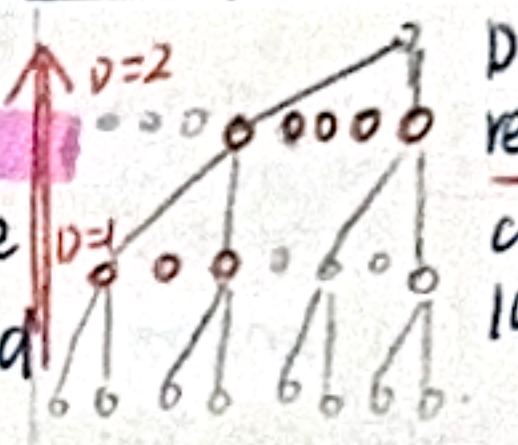
* no temporal dependencies



	CNN	Fully connected
# output unit:	IWH	IWH
# weights:	K^2IJ	W^2H^2IJ
# connection:	K^2WHIJ	W^2H^2IJ

1个 filter → 1个 output map k^2

② Dilated convolution



Dramatically increase CNN's receptive field by using dialated convolutions with progressively larger dilation rates in succesive layers

causal limitation: • only uses past info, limit ability to consider future context • Early time steps have less context available, potentially resulting in lower quality outputs at sequence beginings

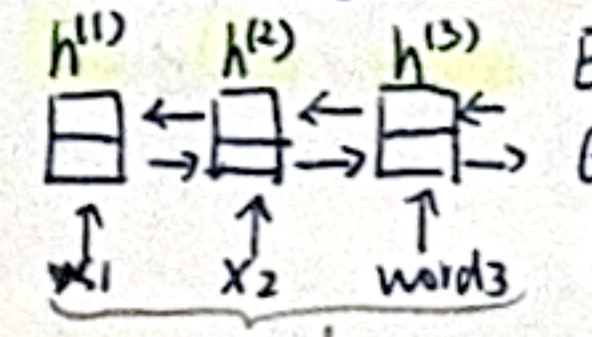
Dilated limitation • can create checkerboard artifacts if dilation factors aren't carefully designed • May miss fine grained local patterns when dilation rate high

Attention based RNN

不仅 compression. 部分

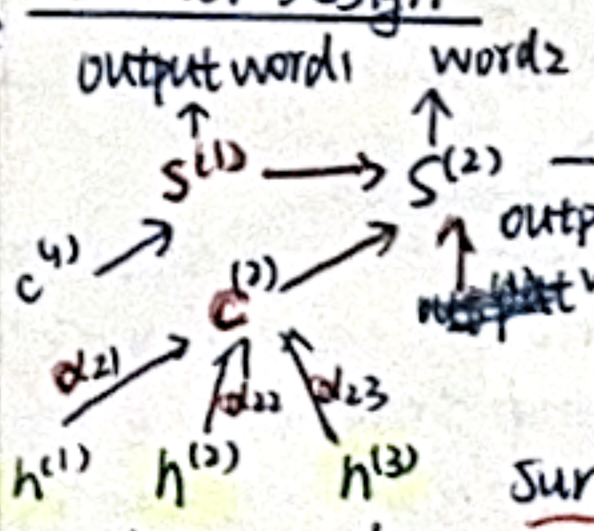
teach the model to selatively foucs on rel part

Encoder Design bidrectional RNN → context



Each hidden unit is a Gated Recurrent Unit (update & reset gates)

Decoder Design



• predict one word at a time ~~use out~~

- use output as input

• receive a context vector at each time step, a weighted sum of the encoder annotation

• Allows decoder to focus on different parts of the input sentence when generate each

context $c^{(i)} = \sum_j \alpha_{i,j} h^{(j)}$ ← Encoder Annotation

attention weights $a_{ij} = \frac{\exp(e_{i,j})}{\sum_{j'} \exp(e_{ij'})}$ relevant ↑ a ↑

$e_{ij'} = a(s^{(i-1)}, h^{(j)})$ ← Decoder's state

↑ alignment model

Decoder 比较 和 h

$a(s_{i-1}, h_j) = V_a^T \tanh(W_a s_{i-1} + U_a h_j)$

Additive attention

Content based addressing: annotation function a depends on annotation vector $\vec{h}^{(i)}$, 不是位置

Focus on semantic not position

Info flow in standard Encoder Decoder: the max path length is proportional to # of time steps.

Info flow in Attention Based RNN: path length const between the encoder inputs and decoder hidden

	Traing Compl	Train mem	Test comp	Test mem	seq op	max path
RNN	k^2dt	kdt	k^2dt	kd	t	t
atten RNN	k^2dt^2	kdt^2	k^2dt^2	kdt	t	1