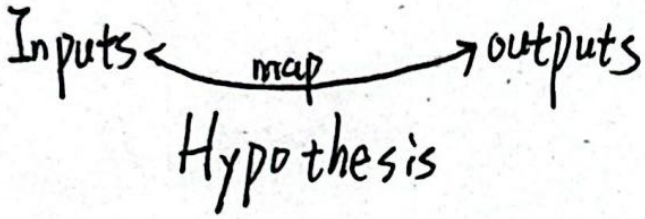
before we came into next section,

ML - consist of 3 Parts:

1. 

2. Loss function: specifies how well Hypothesis perform

3. Optimation method: minize the sum of losses

Linear to non-linear

$h_\theta(x) = \theta^T \phi(x) \quad \theta \in \mathbb{R}^{d \times k} \quad \phi: \mathbb{R}^n \to \mathbb{R}^d$

我发现他把单独的 θ^T 叫作 linear classifier

老师说出

$\theta^T \sigma(W^T x) \neq \tilde{\theta}^T x$

$W^T \in n \times d$, $\sigma: \mathbb{R}^d \to \mathbb{R}^d$, σ is a nonlinear function

How to create ϕ?

1. manual engineering i.e. W be random Gaussian, σ be cosine → random Fourier Features
2. "learn itself"

Now we want to optimize both θ and W

哈哈

原话: Cease pretending any requirement on depth

译: 不线性就是 Deep learning!

Beyond "Just not linear"

So "deep learning" just means "ML using neural network"

A neural network is a particular type of hypothesis class

multiple, parameterized, differentiable functions (aka "layers") composed together from inputs to outputs

定义:

$h_\theta(x) = W_2^T \sigma(W_1^T x)$, $\theta = \{W_1, W_2\}$

σ is e.g. sigmoid, ReLU

batch form:

$h_\theta(X) = \sigma(XW_1)W_2 \quad X \in R^{m \times n}$

Theorem:

Two-layer-network (one-hidden-layer-network)

Is Universal function Approximation

可以无限拟合任何平滑函数 $f: R \to R$

数学表述: given closed region $D \subset R$

for any $f: R \to R$ and $\epsilon > 0$

$\max_{x \in D} |f(x) - \hat{f}(x)| \leq \epsilon$, arbitrarily closely!!

课上给了感性的证明哈哈

注意名词:

layers, activations, neurons

$Z_1 = X$, $Z_{i+1} = \sigma_i(Z_i W_i)$, $i = 1, \cdots, L$, L-layer NN

~~$h_\theta(x)$~~ $Z_{L+1} = h_\theta(X)$, $Z_i \in R^{m \times n_i}$ $W_i \in R^{n_i \times n_{i+1}}$

手动链式指导：

$$\frac{\partial l_{ce}(\sigma(XW_1)W_2, y)}{\partial W_2} = \frac{\partial \cdots}{\partial \sigma(XW_1)W_2} \cdot \frac{\partial \sigma(XW_1)W_2}{\partial W_2}$$

$$= (S - I_y) \cdot \sigma(XW_1)$$

S: $m \times k$，每行是类别概率

I_y: $m \times k$，每行是 hot-encoded

embarassingly

$$\equiv [\sigma(XW_1)]^T (S - I_y)$$

$d \times m$ $\quad m \times k$

$$\frac{\partial l_{ce}(\sigma(XW_1)W_2, y)}{\partial W_1} = \frac{\partial \cdots}{\partial \sigma(XW_1)W_2} \cdot \frac{\partial \sigma(XW_1)W_2}{\partial \sigma(XW_1)} \cdot \frac{\partial \sigma(XW_1)}{\partial XW_1} \cdot \frac{\partial XW_1}{\partial W_1}$$

$$= (S - I_y) \cdot W_2 \cdot \sigma'(XW_1) \cdot X$$

$\sigma'(XW_1)$: just a scalar

notice $W_1 \in R^{n \times d}$

embarassingly

$$= X^T \left((S - I_y) \cdot W_2^T \circ \sigma'(XW_1) \right)$$

$\circ$: element-wise multiplication

哈哈，Tim Dettmers 说：

the biggest take-away from all of this is you just want to invest all your time learning automatic differentiation and don't do it anymore.

假定你对 G_i 熟悉了，$G_i = \frac{\partial \ell(Z_{L+1}, y)}{\partial Z_i}$

$$G_i = G_{i+1} \cdot \frac{\partial Z_{i+1}}{\partial Z_i}$$

$$= G_{i+1} \cdot \frac{\partial \sigma(Z_i W_i)}{\partial Z_i W_i} \cdot \frac{\partial Z_i W_i}{\partial Z_i}$$

注：$Z_{i+1} = \sigma_i(Z_i W_i)$

$$= G_{i+1} \cdot \sigma'(Z_i W_i) \cdot W_i \quad 1)$$

~~基~~

定义：$G_i \equiv \frac{\partial \ell(Z_{L+1}, y)}{\partial Z_i}$ $Z_i \in R^{m \times n_i}$

$$= \nabla_{Z_i} \ell(Z_{L+1}, y) \quad 2)$$

embarassingly, according 1) and 2)

$$G_i = [G_{i+1} \circ \sigma'(Z_i W_i)] \cdot (W_i)^T$$

另外，with respect to W_i not Z_i

$$\nabla_{W_i} \ell(Z_{L+1}, y) = G_{i+1} \cdot \frac{\partial \sigma(Z_i W_i)}{\partial W_i}$$

$$= G_{i+1} \cdot \sigma'(Z_i W_i) \cdot Z_i$$

embarassingly

$$= (Z_i)^T (G_{i+1} \circ \sigma'(Z_i W_i))$$

可引出 gradient 计算方法:

Backpropagation

I. Forward Pass

Init: $Z_1 = X$

Iter: $Z_{i+1} = \sigma(Z_i \cdot W_i)$

II. Backward Pass

Init: $G_{L+1} = \nabla_{Z_{L+1}} \ell(Z_{L+1}, y) = S - I_y$

Iter: $G_i = (G_{i+1} \circ \sigma_i'(Z_i W_i)) \cdot W_i^T \quad , i = L, \cdots, 1$

and

$\nabla_{W_i} \ell(Z_{L+1}, y) = Z_i^T (G_{i+1} \circ \sigma_i'(Z_i W_i))$

just the chain rule, huh?

G_{i+1} → also need stored

Z_i → calculated in Forward Pass and stored

which indicates trade-off: more efficiently gradient, more memory needed

最后提了一下，所有这些 hacky 的操作，为 vector Jacobian product 有关

鉴于笔记访问的方便，我们仍将 CMU 课程作笔录

n k (labeld) m (training numbers) 记住它们!

hypothesis function $h(x) = \begin{bmatrix} h_1(x) \\ \vdots \\ h_k(x) \end{bmatrix}$ $h: R^n \to R^k$

linear form:

$$h_\theta(x) = \theta^T x$$

θ 是参数矩阵

x 是一个 column vecter

$(k \times n) \cdot (n \times 1)$

code neater notation:

$x \in R^{m \times n} = \begin{bmatrix} -x^{(1)T}- \\ \vdots \\ -x^{(m)T}- \end{bmatrix}$. 转置是因为 $x^{(i)}$ 是 column vector

$y = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(k)} \end{bmatrix}$

$$h_\theta(X) = \begin{bmatrix} -h_\theta(x^{(1)})^T- \\ \vdots \\ -h_\theta(x^{(m)})^T- \end{bmatrix}$$

$$= \begin{bmatrix} [\theta^T x^{(i)}]^T \end{bmatrix}$$

$$= [x^{(i)T} \cdot \theta] = X \cdot \theta$$

$(m \times k)$

Loss function

① simplest one: classification error

$$L_{err}(h(x), y) = \begin{cases} 0 & \text{if } \operatorname{argmax}_{index} h(x) = y \\ 1 & \end{cases}$$

不好优化, not differentiable

② Softmax or Cross-Entropy loss

转换成更"像"概率的样子:

→ make them all positive and sum to one

→ exponentiate and normalize

$$\rightarrow \frac{\exp(h_i(x))}{\sum_{j=1}^{k} \exp(h_j(x))} = p(\text{label} = i) = z_i$$

$$\Leftrightarrow z \equiv \text{normalize}(\exp(h(x))) \equiv \text{Softmax}(h(x)) \quad (k,1)$$

实际的计算并不会直接对上面的表达式,

$$l_{ce}(h(x), y) = -\log \text{Softmax}(h_y(x)) = -h_y(x) + \log \sum_{i=1}^{k} \exp(h_i(x))$$

(y在这里像是某种编号吧)

Think ML-algorithm as a method for solving associated optimization problem

~~i.e.~~ → Core:

$$\underset{\theta}{\text{Minimize}} \left\{ \text{Average}\left\{ \ell\left(h_\theta(x^{(i)}), y^{(i)}\right) \right\} \right\}$$

ℓ: loss_func; h_θ: hypothesis

Any Supervised algorithm is to solve this form problem

i.e. softmax regression:

minimize $\frac{1}{m}\sum_{i=1}^{m} l_{ce}(\theta^T x^{(i)}, y^{(i)})$, which is also minize $f(\theta$

手段：gradient descent (∇)，powers all deep learning!

Though $f(\theta)$ itself is a scalar, the gradient is $n \times k$ matrix
same as $\theta \in R^{n \times k}$

$\nabla_\theta f(\theta)$, its result belongs to $R^{n \times k}$

whose direction (saddly $R^{n \times k}$) most increases $f(\theta)$

So in iteration, we go opposite:

$$\theta := \theta - \alpha \nabla_\theta f(\theta)$$

↓ learning rate

如果 m 增长到太大，$f(\theta)$ 中 $\frac{1}{m}\sum_1^m$ 会造成计算负担

→ Stochastic GD (SGD):

sample a minibatch of data $X \in R^{B \times n}$

轻松的，对应：$\theta := \theta - \frac{\alpha}{B}\sum_1^B \nabla_\theta l(h(x^{(i)}), y^{(i)})$

xManually calculate gradient

i.e. the softmax form:

consider $\nabla_h l_{ce}(h, y)$

单独看 $\frac{\partial l_{ce}(h,y)}{\partial h_i} = \frac{\partial}{\partial h_i}(-h_y + \log \sum_{j=1}^{k} \exp h_j)$

$= -1\{i=y\} + \cdots$

检验 $= -1\{i=y\} + P(\text{label} = i)$

$\nabla_h l_{ce}(h, y) = z - e_y$

for $\nabla_\theta l_{ce}(\theta^T x, y)$, the matrix differential calculus can be cumbersome

我们粗暴的 treat it as scalar, and check numerically

$$\frac{\partial l_{ce}(\theta^T x, y)}{\partial \theta} = \frac{\partial l_{ce}(\theta^T x, y)}{\partial \theta^T x} \cdot \frac{\partial \theta^T x}{\partial \theta}$$

$$= \underset{k\times 1}{(z - e_y)} \cdot \underset{n\times 1}{x}$$

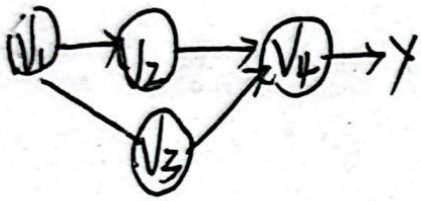
$= \text{embarrassingly} = x(z - e_y)^T$

for batch form:

$\nabla_\theta l_{ce}(X\theta, y) \overset{\text{观察得}}{=\!=} X^T(z - I_y)$

Chen's class

① forward mode automatic differentiation
DAG图啥，computational graph
② Reverse mode 非常有趣



学迷糊了.:

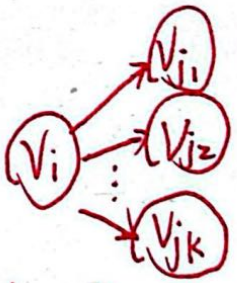
def. $\overline{V_i} = \frac{\partial Y}{\partial V_i}$

$$\frac{\partial f(V_2, V_3)}{\partial V_1} = \frac{\partial f(V_2, V_3)}{\partial V_2} \cdot \frac{\partial V_2}{\partial V_1} + \frac{\partial f(V_2, V_3)}{\partial V_3} \cdot \frac{\partial V_3}{\partial V_1}$$

一个 Reverse mode 的例子：$\overline{V_1} = \overline{V_2} \cdot \frac{\partial V_2}{\partial V_1} + \overline{V_3} \cdot \frac{\partial V_3}{\partial V_1}$

def. $\overline{V_{i \to j}}$ ~~$\frac{\partial V_j}{\partial V_i}$~~

$$\equiv \overline{V_j} \cdot \frac{\partial V_j}{\partial V_i}$$

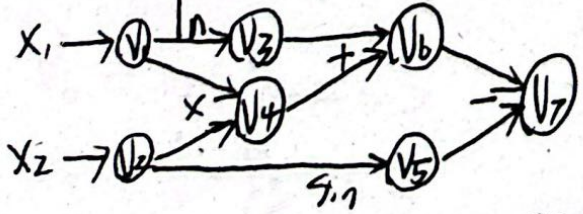


这就是对应一个 node 有 mutiple output 的情况！

有简洁的形式：

$$\overline{V_i} = \sum_{j \in Next(i)} \overline{V_{i \to j}}$$

给你画个 computation graph 示意，$y = \ln(x_1) + x_1 x_2 - \sin x_2$



each node is an operation and can represent an immediate value

添加一页：

Numerial differentiation

一般般形式：$\frac{\partial f(\theta)}{\partial \theta_i} = \lim_{\varepsilon \to 0} \frac{f(\theta + e_i\varepsilon) - f(\theta)}{\varepsilon} + O(\varepsilon)$

高精度：$\frac{\partial f(\theta)}{\partial \theta_i} = \lim_{\varepsilon \to 0} \frac{f(\theta + e_i\varepsilon) - f(\theta - e_i\varepsilon)}{2\varepsilon} + O(\varepsilon^2)$.

当然了，这只是手算

which is used for numerial checking

在前面 DAG 的示意中

我们示例 $\frac{dy}{dx_1}$ given $x_1=2$, $x_2=5$

→ 它的缺点是 input $\in R^n$，就要做 n 次 Forward AD 不效率！

Forward AD trace:

$\dot{v}_1 = 1 \quad \dot{v}_2 = 0$

$\dot{v}_3 = \frac{1}{v_1} \cdot \dot{v}_1 = \frac{1}{2}$

$\dot{v}_4 = \dot{v}_1 \cdot v_2 + v_1 \cdot \dot{v}_2$

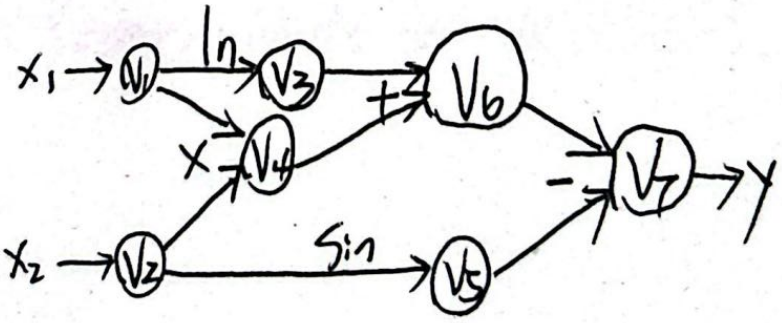
$= 1 \cdot 5 + 2 \cdot 0$

$= 5$

$\dot{v}_5 = \cos v_2 \cdot \dot{v}_2 = 0$

$\dot{v}_6 = \dot{v}_3 + \dot{v}_4 = 5.5$

$\dot{v}_7 = \dot{v}_6 - \dot{v}_5 = 5.5$



as long as i know the former input node

非常漂亮，产生了非常棒可依赖！

e.g. Reserve mode AD

还是那个图

$$\frac{\partial y}{\partial v_7} = 1$$

$$\frac{\partial y}{\partial v_6} = \frac{\partial y}{\partial v_7} \cdot \frac{\partial v_7}{\partial v_6} = 1 \cdot 1 = 1$$

$$\frac{\partial y}{\partial v_5} = \frac{\partial y}{\partial v_7} \cdot \frac{\partial v_7}{\partial v_5} = 1 \cdot (-1) = -1$$

$$\frac{\partial y}{\partial v_4} = \frac{\partial y}{\partial v_7} \cdot \frac{\partial v_7}{\partial v_6} \cdot \frac{\partial v_6}{\partial v_4} = 1 \cdot 1 \cdot 1 = 1$$

$$\frac{\partial y}{\partial v_3} = \frac{\partial y}{\partial v_6} \cdot \frac{\partial v_6}{\partial v_3} = 1 \cdot \frac{v_4}{v_3} 1 = 1$$

$$\frac{\partial y}{\partial v_2} = \frac{\partial y}{\partial v_5} \cdot \frac{\partial v_5}{\partial v_2} + \frac{\partial y}{\partial v_4} \cdot \frac{\partial v_4}{\partial v_2} = \cdots$$

$$\frac{\partial y}{\partial v_1} = \frac{\partial y}{\partial v_3} \cdot \frac{\partial v_3}{\partial v_1} + \frac{\partial y}{\partial v_4} \cdot \frac{\partial v_4}{\partial v_1} = 1 \cdot \frac{1}{v_1} + 1 \cdot v_2 = 5.5$$

[illegible].

$$= \begin{pmatrix} \frac{\partial y}{\partial x_2} \\ \frac{\partial y}{\partial x_1} \end{pmatrix} = \nabla f$$，所以 Reverse Mode 做一次就够了。

Backprop Vs Reverse mode AD

↓

by extending computational graph

好处：
① the compation process is another computional graph
可以做梯度的梯度 for free
可拓展性很棒!!!
② output is a computional graph
不需要 run forward
which make process asymmetric

最后提了一下 Reverse mode AD on data structures
"tuple value"？没怎么听懂

我把伪代码写一遍，可能会清晰些：

```
def grad(out):
    node_tobe_grad = {最后一个点}
    for i in reverse_todo_order(out):  ——→ 这个逆序实现经典问题
        V̄i = Σj V̄i→j = sum(node_tobe_grad[i])  ——→ 先把当前 V̄i 算出来
        for k ∈ input(i):
            V̄k→i = V̄i · ∂Vi/∂Vk
            ~~upper~~
            node_tobe_grad[k].append(V̄k→i)  →把 i 的前驱结点算出来，加入维护队列
```

数学直觉上，每轮到一个 i 时，它的 j^{th} output(i)，$\overline{V_{i\to j}}$ 是全部被计算完毕的

它的那个 R-mode Computational Graph

就是照上面的流程画出来的，

不过是用了 stored memory，幸好我很菜。

Fully connected networks

来：$b \in \mathbb{R}^{n_{i+1}}$

$b.\text{reshape}((1, n_{i+1})).\text{broadcast\_to}((m, n_{i+1}))$

newton's method:

directly going to the optimized point.

momentum:

$u_{t+1} = \beta u_t + (1-\beta) \nabla_\theta f(\theta_t)$，$\beta$ is momentum averaging parameter

$\theta_{t+1} = \theta_t - \alpha u_{t+1}$

所有过去的导数都会起作用.

momentum 变种：

一开始 warm-up 可能很慢

"Unbiasing momentum"

$$\theta_{t+1} = \theta_t - \frac{\alpha u_{t+1}}{1-\beta^{t+1}}$$

第一步跨一些，可能收敛快一些

Nesterov Momentum:

$$U_{t+1} = \beta U_t + (1-\beta) \nabla_\theta f(\theta_t - \alpha U_t)$$

↓ 做前瞻预测

Adam: good optimizer in practice

$$U_{t+1} = \beta_1 U_t + (1-\beta_1) \nabla_\theta f(\theta_t)$$

$$V_{t+1} = \beta_2 V_t + (1-\beta_2) (\nabla_\theta f(\theta_t))^2$$

$$\theta_{t+1} = \theta_t - \frac{\alpha U_{t+1}}{\sqrt{V_{t+1}} + \varepsilon}$$

↘ 每次步长都差不多

几条 tips:

1. Initialization matters
2. Weights doesn't move that much

最后是有趣的 varience 跟随输入现象:

independent random variables: $x \sim N(0,1)$, $w \sim N(0, \frac{1}{n})$

$Var(w^T x) = 1$

Kaiming normal initialization: $W_i \sim N(0, \frac{2}{n} I)$ 没听懂.

declarative programming. 先宣称计算，再跑。

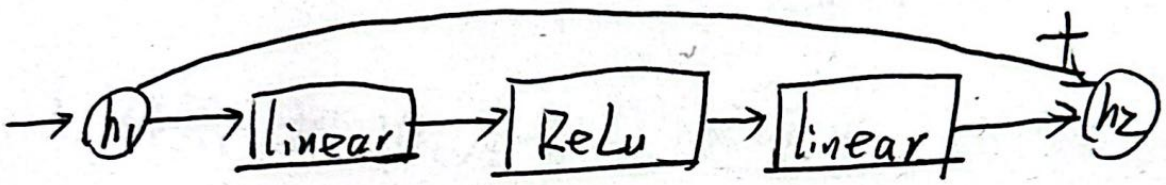
↳ tensorflow.

pytorch: imperative

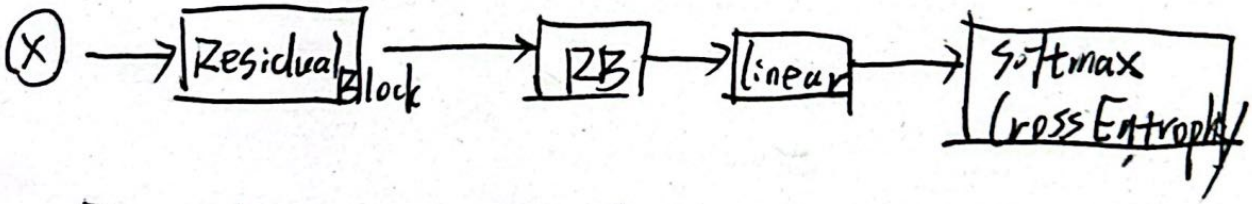
立刻计算

High level modular library components.

0. Residual Block:



1. Muti_layer Residual Net



. Deep learning is modular in nature

nn.module就是把上面2个东西读在一起

things to consider:

1. tensor in tensor out
2. get list of trainable parameters
3. way to initilize the parameters

当然了 Softmax Cross Entropy 是 Tensor in, scalar out

Optimizer：就是 SGD. Momentum 啊那些

又说 regularization，l2 regularization，完全没有听懂。

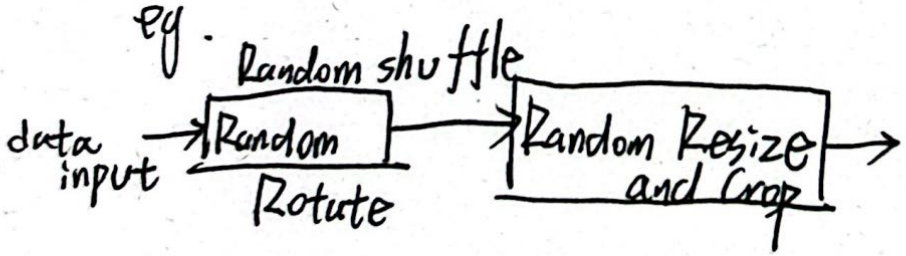
l1 正则化：$loss\_func += \sum |w_i|$

会使很多不重要的 w_i 直接变 0

l2 正则化：$loss\_func += \sum w_i^2$

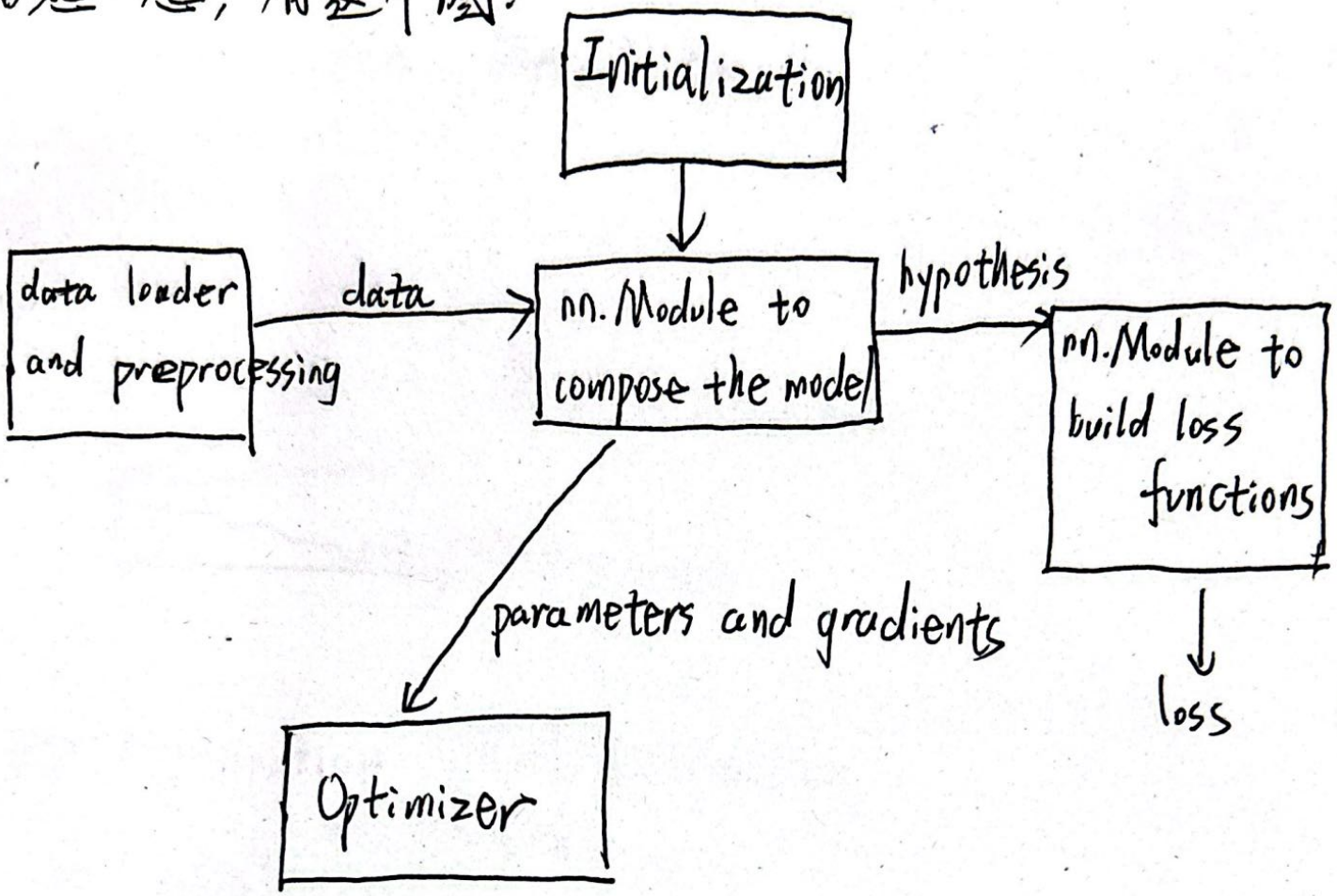
作用：把 w_i 尽量拉回零点，w_i 过大往往意味着过拟合。

Data load and augmentation



对训练准确也有影响。

合起一起，有这个图：


Initialization
data loader and preprocessing
data
nn.Module to compose the model
hypothesis
nn.Module to build loss functions
parameters and gradients
loss
Optimizer

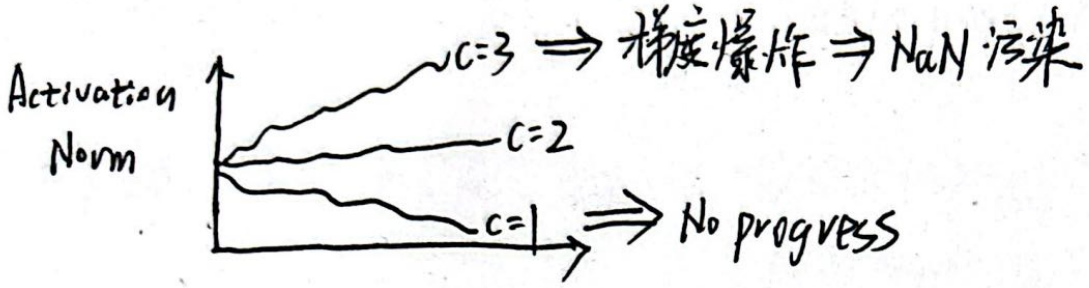
tricks:

Normalization Regularization

一、

现象:

Init: $W_i \sim N(0, \frac{c}{n})$



Layer normalization: 按行来 normalize.

$$\hat{z}_{i+1} = b_i(W_i^T z_i + b_i)$$

↗ 行向量哈哈

$$z_{i+1} = \frac{\hat{z}_{i+1} - E[\hat{z}_{i+1}]}{\sqrt{Var(\hat{z}_{i+1}) + \varepsilon}}$$

↳ 极小, 避免 div 0

Batch normalization: 按列来 norm化 (归一?? 不知道)

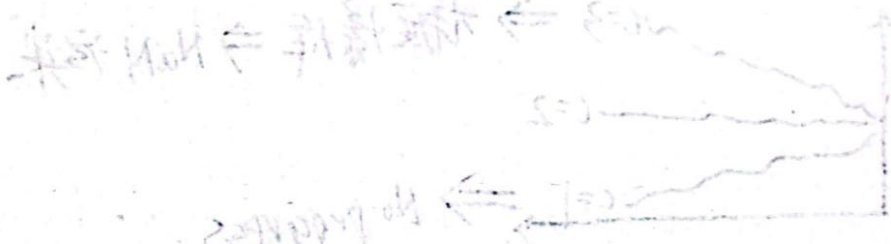
odd的地方是让不同行的样本产生依赖.

二、

~~我加的~~ DNN是 over-parametered的, 因为它的表现力太强啦! 很容易过拟合, 导致泛化能力不足.

Regularization: limiting the complexity of function class.

限制表达力, 预防过拟合.



a whole lot of designing DLSys is to Implicitly regularizing the complexity of functions so we can generalize better.

比如，每次训练，其实 weight 变化不大。

l_2 Regularization (or weight decay)

minize $\frac{1}{m}\sum_{1}^{m} \ell(h(x), y) + \frac{\lambda}{2}\sum_{i=1}^{L} \|W_i\|_F^2$

梯度下降：

$W_{i+1} = W_i - \alpha \nabla_{W_i} \ell(\cdots) - \alpha\lambda W_i = (1-\alpha\lambda) W_i - \alpha \nabla_{W_i} \ell(\cdots)$

每次都 shrink $(1-\alpha\lambda)$

Dropout - another common regularization strategy

$$\hat{z}_{i+1} = \sigma_i(W_i^T z_i + b_i)$$

$$(z_{i+1})_j = \begin{cases} \text{itself\_row} / (1-p) & \text{with prob } 1-p \\ 0 & \text{with prob } p \end{cases}$$

除以是为了保持平均数不变

dropout as a Stochastic approximation

使得DNN不依赖特定的行(神经元)，被迫学习更鲁棒的特性

参考一篇：

表面象是 Batch Norm reducing internal covariant shift，训练加速了，前者真的是后者的原因吗？

并不！

事实是 BatchNorm makes the optimization landscape smoother!

Convontional Operator

定义：略

一个例子，in image processing.

① Gaussian blur：变糊了

② "Grad"：勾画庞廓

$$\left(z * \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}\right)^2 + \left(z * \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}\right)^2$$

在传统信息处理是 pre-fixed

而卷积核确是可学习的！

Mutichannel-convolution

卷积核 $W_{ij} \in R^{C_{out} \cdot C_{in} \cdot k \cdot k}$

↓

k is kernel size.

（进一步简化：把 Cout. Cin 也分组）

practical Convolution tricks

1. for kernel size k, pad input with (k-1)/2 zeros in all side (when k is odd)

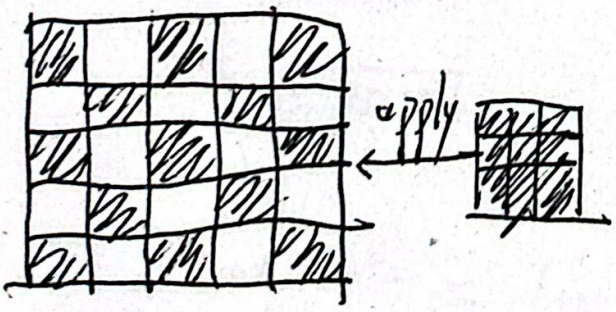
2. Strided Convolution

窗口当长每次移动>1，卷积出来更小。

3. Max pooling

从卷积核中选个最大的作数。

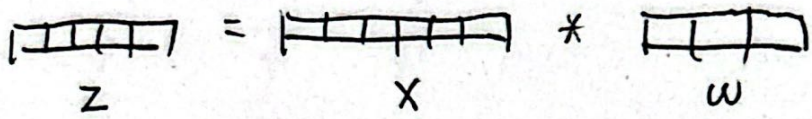
4. Dilations



Differentiating Convolution

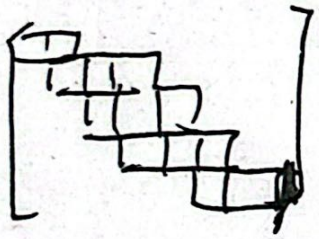
缺点(工程上): store all intermediate product before add them up

一维 toy 示例



* 回忆 $Z = W^T \cdot X$

W:



W^T 居然形式上一模一样!

除了 $|W_1|W_2|W_3|$

变成 $|W_3|W_2|W_1|$

有: $\nabla \cdot \frac{\partial Conv(X,W)}{\partial X} = Conv(\nabla, flip(W))$

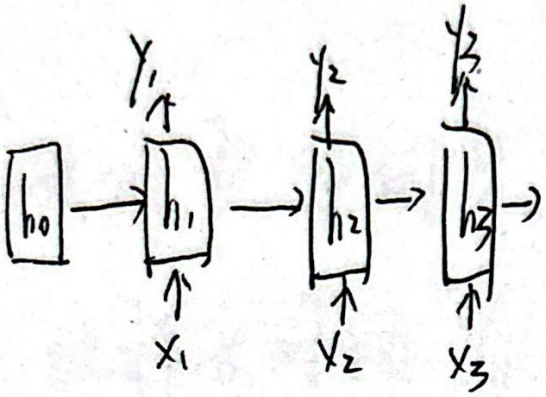
如果他们过来: $Z = \begin{bmatrix} 0 & x_1 & x_2 \\ x_1 & x_2 & x_3 \\ \cdots & \cdots & \cdots \\ x_4 & x_5 & 0 \end{bmatrix} [W]$

↓

这种叫 "im2col" 阵

Sequence Modeling Recurrent NN

RNN:



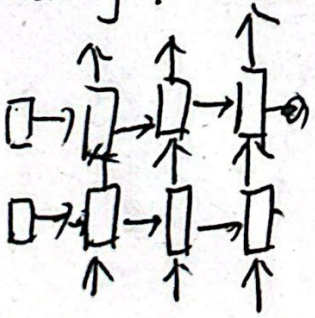
$$h_t = f(W_{hh} \cdot h_{t-1} + W_{hx} \cdot X_t + b_n) \qquad y_t = g(W_{hy} \cdot h_t)$$

会涉及 BPTT back progation through time

先把时间跑一段再做微分 loss.

(怎么递归微分呀?)

Stacking RNNs



就是 h 层加深了.

↓

hidden unit

这个有些缺点：

variance 大：梯度爆炸 norm 猛增。

variance 太小：Vanishing activation

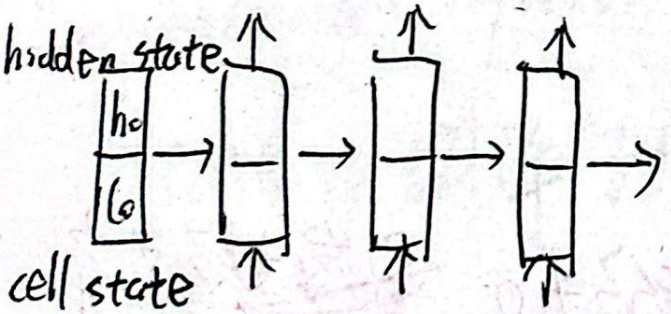
我们想要的是 "long stable range" dependencies

这样长时间 X_i 们都起作用。

ReLu can grow unboundly. why not bounded activation

实际上不起作用。

Long short Term Memory



(也就是另一个 cell state)

$$\begin{bmatrix} i_t \\ f_t \\ g_t \\ o_t \end{bmatrix} = \begin{pmatrix} \text{Sigmod} \\ \text{Sigmod} \\ \tanh \\ \text{Sigmoid} \end{pmatrix} \left(W_{hh} \cdot h_{t-1} + W_{hx} \cdot X_t + b_n \right)$$

$$\begin{bmatrix} i \\ f \\ g \\ o \end{bmatrix}$$

$$c_t = c_{t-1} \circ f_t + i_t \circ g_t$$

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

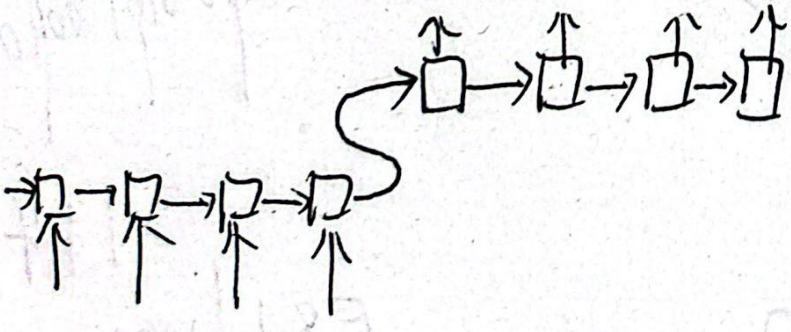
为什么他们能 help with vanishing?

keypoint:

1. scaling down c_{t-1}. and add a term to it.

Sequence-to-Sequence Model

可以看作翻译了!



they leverage RNNs in Module!!!

Bidirectional RNNs

微分很难

like translation, we don't only need the current-before things. 这个整个都可以用，像翻译

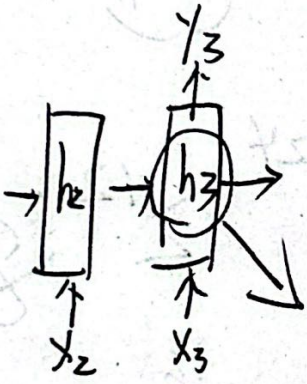
Transformers and Attention

two approaches to time series modeling

1.1 RNN's "latent state" approach

隐藏层

eg.



包含了所有前面的信息.

rnns can capture infinite history

pros: good for long time series prediction

cons: long "compute path" leads to hard to learn

§1.2 "direct prediction" approach

just need a function that can accept different-sized inputs

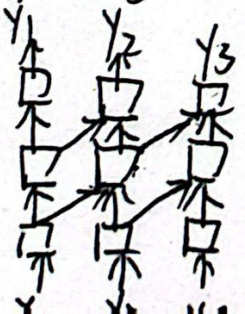
Pros: map from past to current with shorter path

Cons: 每次都要把所有 X_i 输一遍，对比之下 RNN 只需要输入 $X_{current}$

The difference between §1.1, §1.2 就是 RNNs 和 Transformers 的区别

当然了 "direct prediction" 也有很多 architecture, 实现

eg. TCNs



"→" 是卷积，所以这是 CNNs

*名词：receptive field

显然，卷积层越深，CNNs receptive field 才能大一点

p.s. 为了让卷积层更深，可以增加 kernel size

~~显~~显然卷积层的深度跟 receptive field 大小是呈正比的

p.s. 池化 pool 是真女合伙。hhaa

2. Self-attention and Transformer

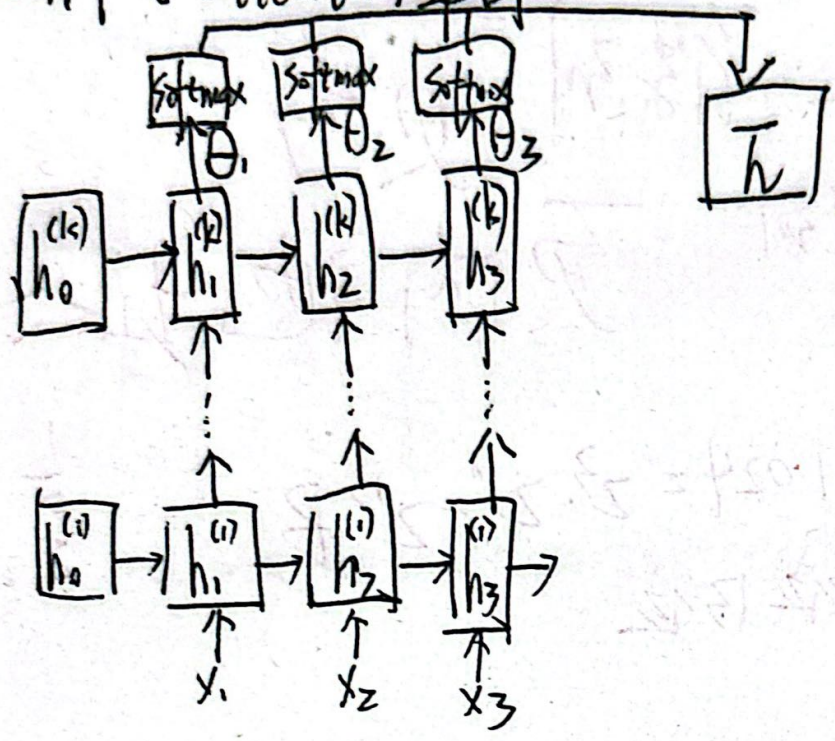
打开一看：

"Attention" in deep learning

前言：

no matter how good these fancy mechanisms skit, the ultimate state will last less depend on origin the Influence of first points fade — the exploding or vanishing gradien

RNN中的 attention 举例



§2.1 Self-attention

HaHaHa, Detters says 他完全没看出

K.Q.V 分别叫 keys, queries, values

可通俗理解

$$\text{SelfAttention} = \text{softmax}\left(\frac{KQ^T}{\sqrt{d}}\right)V$$

where $K, Q, V \in R^{T \times d}$

$$softmax(\frac{KQ^T}{\sqrt{d}})V$$

理解：

$K \cdot Q^T$ 效果 $\begin{bmatrix} k_i^T \cdot q_j \end{bmatrix}$

a way to measure the Similarity between k_i and q_j

$$\underbrace{softmax(\frac{KQ^T}{\sqrt{d}})}_{T \times T} \underbrace{V}_{T \times d}$$

注意到里没有，K在前，Q在后却能置，

它们的顺 k_i, q_j 执行向量乘的时候是完整的

唯独 V 在后面没有能置，相乘的拆开时间说 V_i，按列来处理

Properties of self-attention

1. permutations 无关性 (K.Q.V时序改变，输出不变)

do same permutation ops to K.Q.V

⇒ output act the same permutation

2. allows influence of K.V.Q over ALL TIMEs

without increase parameter Count!

(想想TCNs)

K.V.Q不是Parameter

Nice!
- Self-attention is just an ops to mix all-time
- with no favoritism (because permutationally invarient)
  - △ △
- without increasing parameter count

3. Cons: computation cost is ~~$O(T^2+Td)$~~

时间复杂度

$O(dT^2+T^2+T^2+T\times d\times T)=O(T^2d)$

Transformer Block

$Z^{(i)} \rightarrow Z^{(i+1)}$

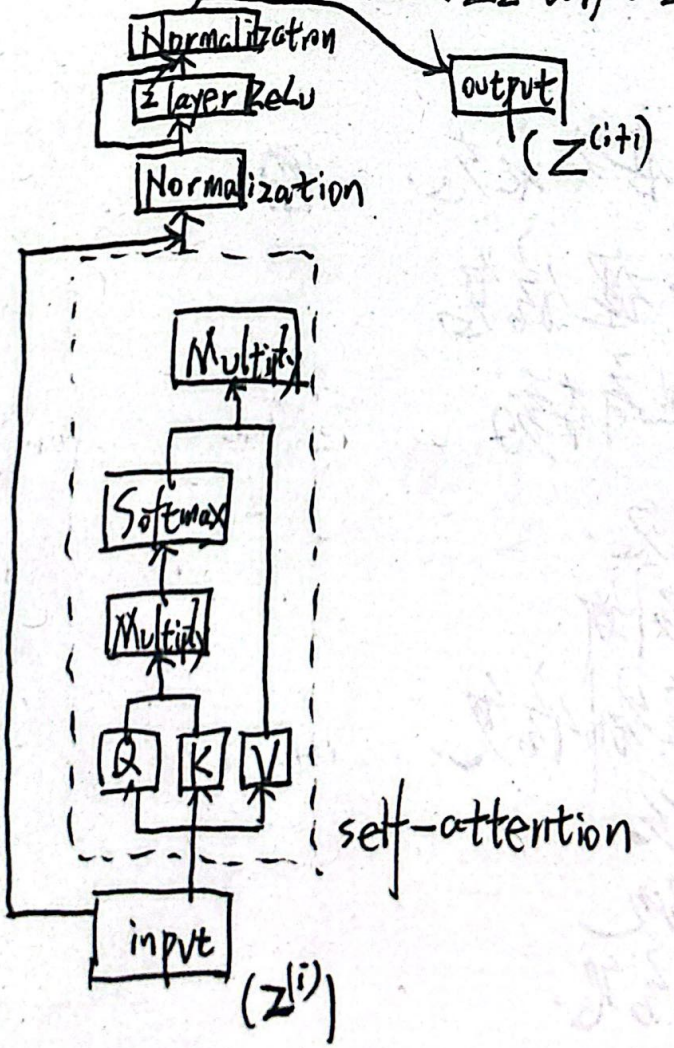
$$\tilde{Z}_1 = \text{Self-attention}(Z^{(i)} \cdot W_K,\ Z^{(i)} \cdot W_Q,\ Z^{(i)} \cdot W_Z)$$

$$= \text{Softmax}\left(\frac{Z^{(i)} \cdot W_K \cdot W_Q^T \cdot Z^{(i)T}}{\sqrt{d}}\right) Z^{(i)} \cdot W_Z$$

$$\tilde{Z}_2 = \text{LayerNorm}(Z^{(i)} + \tilde{Z}_1)$$

$$Z^{(i+1)} = \text{LayerNorm}(\tilde{Z}_2 + \text{ReLu}(\tilde{Z}_2 \cdot W_1) \cdot W_2)$$

画图如下：



Transformer:

Pros: • full receptive field within a single layer

• don't add parameter_count

Cons: shuffle 不管用

output depends all inputs

没法做 time series

解决办法:

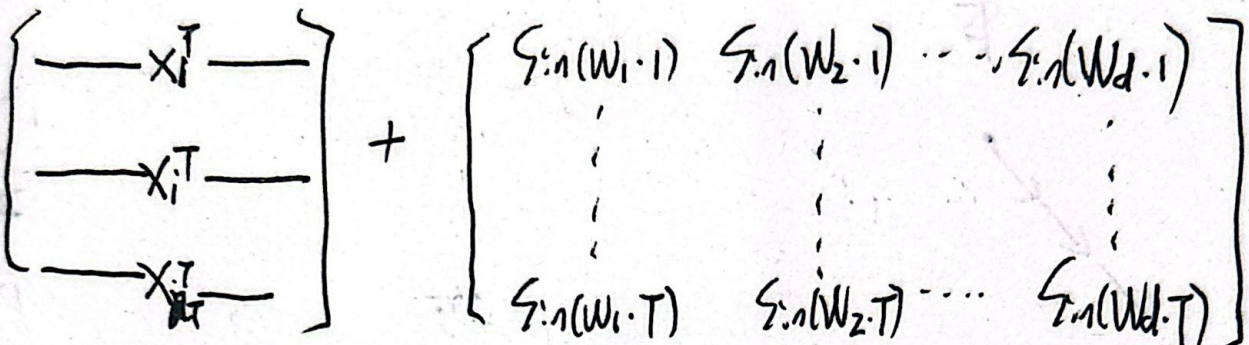
Masked self-attention

$Softmax(\frac{KQ^T}{\sqrt{d}} - M) \cdot V$, $M = \begin{bmatrix} 0 & \infty \\ & 0 \end{bmatrix}$

会变生

$\begin{bmatrix} 正常 & 0 \\ & \end{bmatrix} \cdot [V]$

解决 order Invariance 问题



↓

positional encoding

至于为什么张这样，没讲

最后介绍了 Transformers beyond time series

Transformer, a powerful paradigm used everywhere!