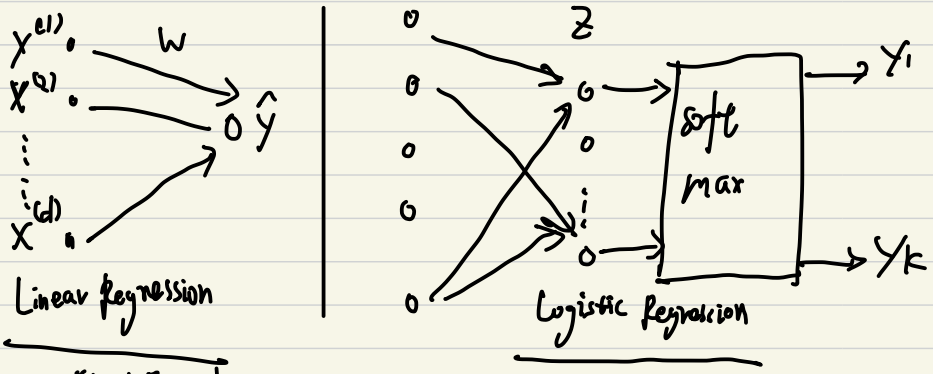
L9

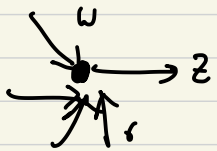


- Feed Forward
- Directed acyclic graphs

Neural Network

primitive $z = \sigma(\langle w, x \rangle + b)$

"Neuron"

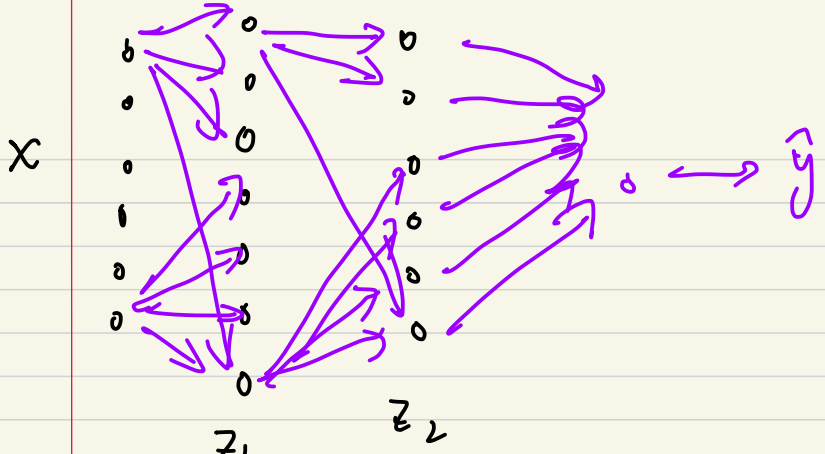


σ "activation function"

Examples: $\sigma(z) = z$ → Linear regression

$\sigma(z) = \frac{1}{1+e^{-\langle x, z \rangle}}$ → Logistic regression

$\sigma(z) = \text{sign}(z)$ → perceptron



$W^1 \quad W^2 \quad W^3$

$z_1 = \sigma^{(1)}(W^{(1)}x + b^{(1)})$

$z_2 = \sigma^{(2)}(W^{(2)}z_1 + b^{(2)})$

$\hat{y} = \sigma^{(3)}(W^{(3)}z_3 + b^{(3)})$

"3-Layer network / 2-hidden Layer network

Why chosen? — historical / biological

Universal approximation Theorem (Cybenko, 1988)

≥ 1 continuous function can be approximated by a 1-hidden Layer network of finite size (for most functions, activation.

Drawback

- Existence Theorem only
- Practical Issue
  - Representation: What network size & shape
  - Training: How to choose the network weights
  - Generalization: Does the learned generalize to unseen input?

Neural network architecture

Design criteria : 1) Weights

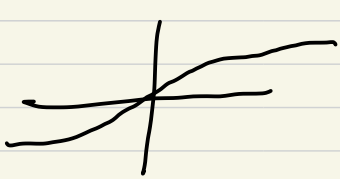
2) Activation function

Activation Functions:

- $\sigma(z) = z$ (this is linear) $\Rightarrow$ [Not too useful for multi-layer networks]

$$y = W^{(3)} W^{(2)} W^{(1)} X = \bar{W} X$$

- $\sigma(z) = \frac{1}{1+e^{-z}}$ sigmoid
- $\sigma(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$ (tanh)



- $\sigma(z) = ReLU(z)$
  $= \max(z, 0)$

- $\sigma(z)$ $sgn(z)$

- $\sigma(z)$ $HT(z)$

Weights:

- Dense Layers ⟶ weights are arbitrary
- Convolution Layers ⟶ weights implement convolution
- pooling Layers ⟶ Downsampling size of output
- Batch normalization ⟶ Rescaling
- Recurrent layers ⟶ Feed back
- Attention layers ⟶ NLP (etc.)

Q: How do I mix & match?

A: No correct answer

Thumb Rule: Just use a good exsiting architecture

Training

Given architecture, how do I train a network?

* Define loss function (+ Optional Regularization)
* Use some variant of gradient descent (GD/SGD/...)

$\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$

- $L(W) = \frac{1}{n}\sum_{i=1}^{n} l(y_i, f_W(x_i)) + \lambda R(W)$
- $W_{t+1} \leftarrow W_t - \alpha^t \nabla L(W)$ — Hard / Complex to compute

Toy example

$x \longrightarrow \overset{\text{sigmoid}}{\sigma} \longrightarrow y$

$z = wx + b$

$f(z) = \sigma(z)$

$L(w,b) = \frac{1}{2}(y - f(x))^2 + \lambda w^2$

$= \frac{1}{2}(y - \sigma(wx+b))^2 + \lambda w^2$

$\nabla L(w,b) = \begin{bmatrix} \frac{\partial L}{\partial w} \\ \frac{\partial L}{\partial b} \end{bmatrix}$

$\frac{\partial L}{\partial w} = (\sigma(wx+b) - y) \cdot \sigma'(wx+b) \cdot x + 2\lambda w$

$$\frac{\partial l}{\partial b} = (\sigma(wx+b) - y)\,\sigma'(wx+b)$$

代入特定值

efficiency

Back propagation / backprop.

- exploit the structure of the network

solve

$$z = wx + b$$

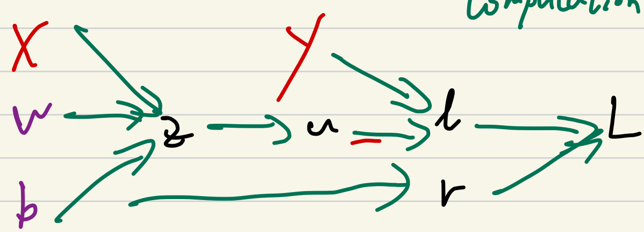
$$u = \sigma(z)$$

$$l = \frac{1}{2}(u-y)^2$$

$$r = w^2$$

$$L = l + \lambda r$$

Computation graph for a single neuron.



FORWARD (graph) (value)

For node $i = 1, 2, \ldots N$

Compute v_i as a function of parent of i (前面的)

$i = 1, 2, \ldots,$

$N \leftarrow$ number of nodes

BACKWARD (graph): (gradient)

For node $i = N-1, \ldots 1$:

Compute $\displaystyle \frac{\partial L}{\partial v_i} = \sum_{j \in \text{children}(i)} \frac{\partial L}{\partial v_j} \cdot \frac{\partial v_j}{\partial v_i}$

$\frac{\partial L}{\partial v} := \partial_v L$ (表达形式)

$\partial_L L := 1$

$\partial_l L := 1$

$\partial_r L = \lambda$

chain rule

$\frac{\partial L}{\partial u} := \partial_l L \cdot \partial_u l = 1 \cdot (u - y)$

$\partial_x L = \partial_u L \cdot \partial_z u$

$= \partial_u L \cdot \sigma'(z)$

$\partial_w L = \partial_x L \, \partial_w z + \partial_r L \, \partial_w r$

$\partial_b L = \partial_z L \cdot \partial_b z$

$= \partial_z L$

没有重复计算 不像会计那样 一步到位 会重复代值

- Computationally efficient
- Moduler 一个一个模块，很标准，可以套用
- Scalable 适用于更大规