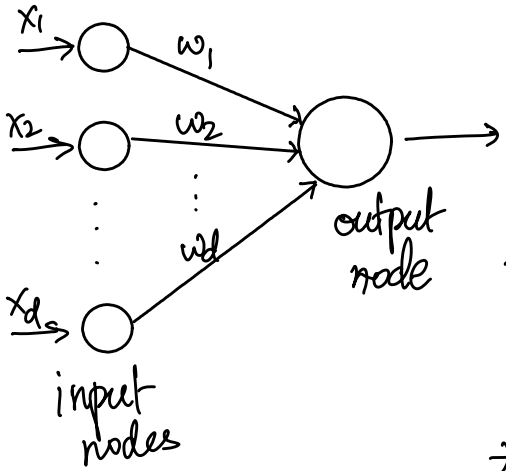
MATH 565: Lecture 24 (04/09/2026)

Today:
* Regression as a CG
* NNs as DACGs
* Gradients in CGs

Linear Regression as a Computational Graph



Output function $\hat{y} = f(\bar{x}) = f(x_1, \ldots, x_d) = \sum_{i=1}^{d} w_i x_i$

$= \bar{w}^T \bar{x}$

* For a sample $(\bar{x}_i, y_i)$, take input $\bar{x}_i$, and evaluate $\hat{y}_i$ ← prediction
* compare $\hat{y}_i$ with actual value y_i.
* loss function computes $(\hat{y}_i - y_i)^2$
* adjust w_i's to reduce (minimize) loss

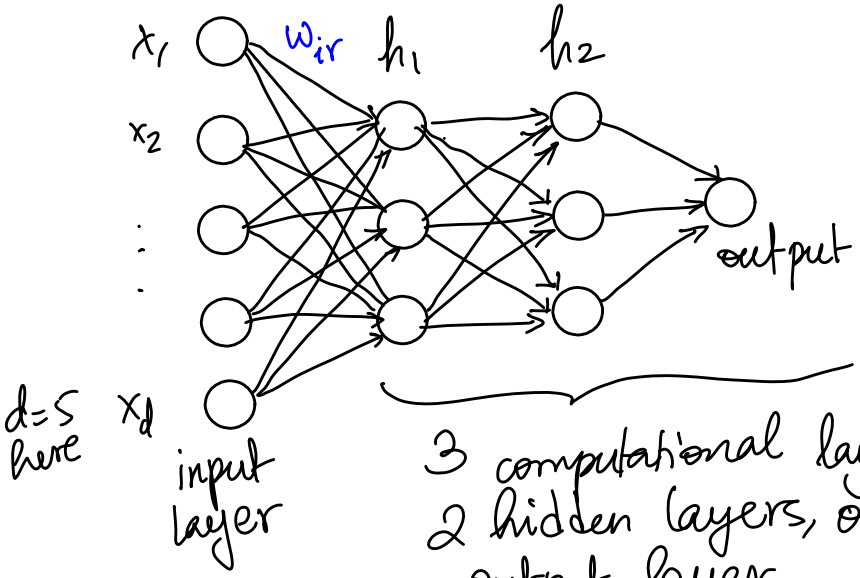
If we do this one sample (x_i, y_i) at a time, this is equivalent to stochastic gradient descent (SGD)!

We can change the output function to model logistic regression, SVM, etc.

Recall: kernel trick in dual of SVM — allows us to model nonlinear kernels. But neural networks (NNs) as DACGs are a lot more powerful, as they allow one to model highly general and nonlinear functions.

Neural Networks (NNs)

* Nodes are arranged in layers
* nodes in layer i are each connected to all nodes in layer $i+1$ (and no others).



h_i : hidden layer i

h_i has p_i nodes

$$h_{ir} = \Phi\left(\sum_{i=1}^{d} w_{ir} x_i\right), \quad r = 1, \ldots, p_1$$

(p_1: # nodes in first layer)

Φ : activation function

In linear regression, $\Phi = I$ (identity). But typical NNs use nonlinear activation functions.

$$\Phi(v) = \frac{1}{1+e^{-v}}$$ sigmoid function

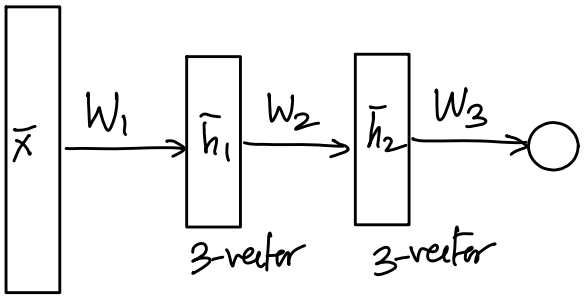
$$\Phi(v) = \frac{e^{2v}-1}{e^{2v}+1}$$ tanh function — hyperbolic tangent

$$\Phi(v) = \max\{v, 0\}$$ ReLU (Rectified linear unit)

$$\Phi(v) = \max\{\min\{v, 1\}, -1\}$$ hard tanh function — hyperbolic tangent

These are all nonlinear functions!

We can also view the NN as having vector architecture:



$\bar{x}$: 5-vector, W_2: 3×3, W_3: 1×3

W_1: 3×5 matrix

$p_1 \times d$, in general

W_{r+1}: $p_{r+1} \times p_r$ matrix.

$$\bar{h}_1 = \bar{\Phi}(W_1 \bar{x})$$
$$\bar{h}_{r+1} = \bar{\Phi}(W_{r+1} \bar{h}_r)$$

If there are k hidden layers,

$$\bar{y} = \bar{\Phi}(W_{k+1} \bar{h}_k)$$
$$= \bar{\Phi}_{k+1}(W_{k+1}(\bar{\Phi}_k(W_k(\bar{\Phi}_{k-1} \cdots))))$$

Let's suppose $\bar{\Phi} = I$, back in our example,

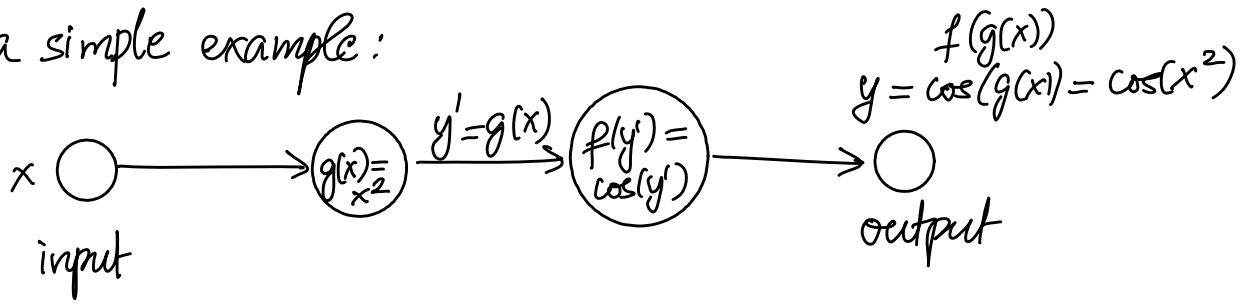
$$\bar{h}_1 = W_1 \bar{x}, \quad \bar{h}_2 = W_2 \bar{h}_1 = W_2 W_1 \bar{x}$$
$$\bar{y} = W_3 \bar{h}_2 = W_3 W_2 W_1 \bar{x} = \tilde{W} \bar{x}$$ for 1×5 matrix $\tilde{W}$

Any two hidden layers correspond to a single hidden layer. More generally, we get a single output layer! Of course, gradient computations are straightforward here.

But we cannot combine and simplify nonlinear $\bar{\Phi}$ activations!

Challenge of Gradients in CGs

Consider a simple example:



Here, the composition of the functions has a simple structure — we can get the closed form expression of the loss function in terms of input variable x. So, we can then take the gradient of the loss function directly.

But now, consider $f(x) = g(x) = \tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$.

$$\Rightarrow f(g(x)) = \frac{e^{2\left(\frac{e^{2x}-1}{e^{2x}+1}\right)} - 1}{e^{2\left(\frac{e^{2x}-1}{e^{2x}+1}\right)} + 1}$$

Too complicated! We may not want to compute closed form function(s) of output nodes in terms of inputs!

We have to work with the CG locally, i.e., in parts, more intelligently!

Broad Framework for Gradient Computations

Before talking about the details of gradient computations in CGs, we describe the general approach for training using CGs.

Approach for training CGs:

1. * Use input data $(\bar{x}_i)$ to fix values at input nodes.
   * Repeatedly find nodes for which all incoming nodes have already been computed, and compute them. We can do this by progressing forward in layers from the input nodes.
   * compute output $(\hat{y}_i)$ values at output nodes, and evaluate loss function J using $(\hat{y}_i - y_i)$

   → called the forward phase.

2. Compute gradient ∇ of loss function J w.r.t. edge weights using <u>back propagation</u>
   ≡ backward phase

3. Update the w_j's ($\bar{w}$ or $W_{p_{r+1} \times p_r}$) in the direction of $-\nabla J$.

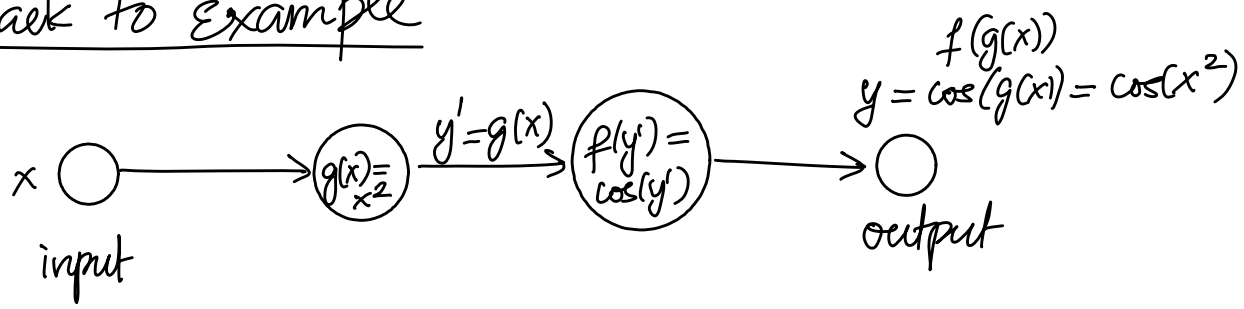
We cycle through all n sample points $(\bar{x}_i, y_i)$. A single cycle through all sample points is an <u>epoch</u>.

Computing Node-to-Node Derivative

Back propagation using chain rule for differentiation.

The idea was known to optimizers back in 1960s, but was reinvented/reintroduced by ML researches in 1980s.

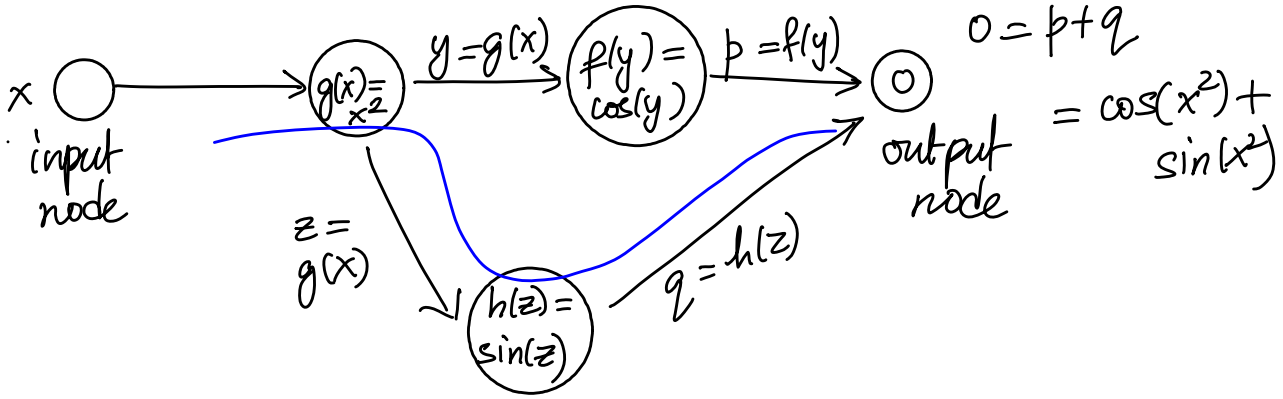
Back to example



$$f(g(x)) = \cos(g(x)) = \cos(x^2)$$

$$\frac{\partial f(g(x))}{\partial x} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial x} = -\sin(g(x)) \cdot 2x = -2x \sin(x^2).$$

A slight generalization:



$$\frac{\partial o}{\partial x} = \frac{\partial o}{\partial p} \cdot \frac{\partial p}{\partial y} \cdot \frac{\partial y}{\partial x} + \frac{\partial o}{\partial q} \cdot \frac{\partial q}{\partial z} \cdot \frac{\partial z}{\partial x}$$

$y = x^2$
$z = x^2$

$$= 1 \cdot -\sin(y) \cdot 2x + 1 \cdot \cos(z) \cdot 2x$$

$$= -2x \sin(x^2) + 2x \cos(x^2)$$