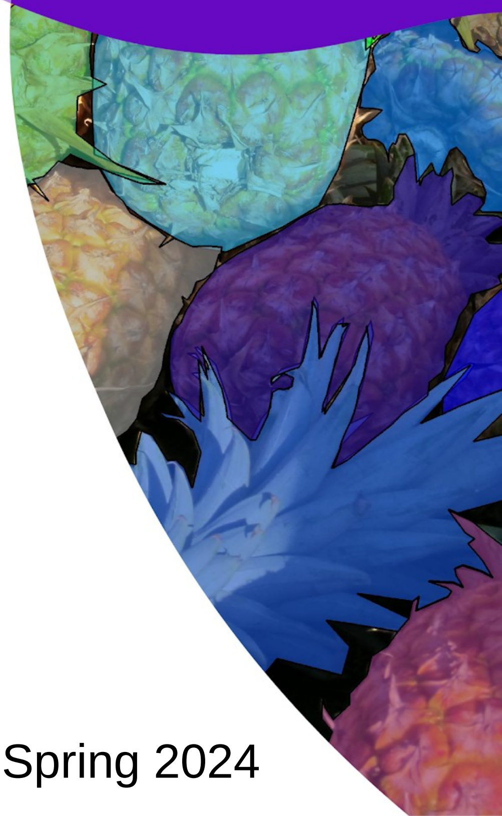
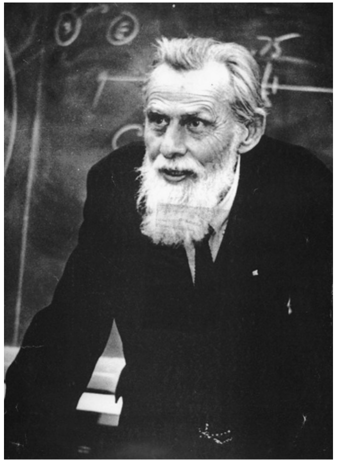
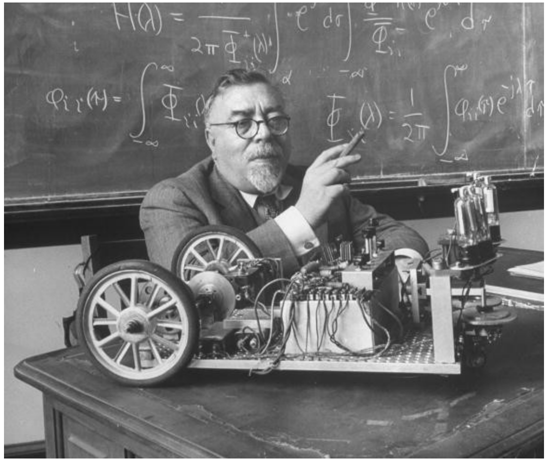
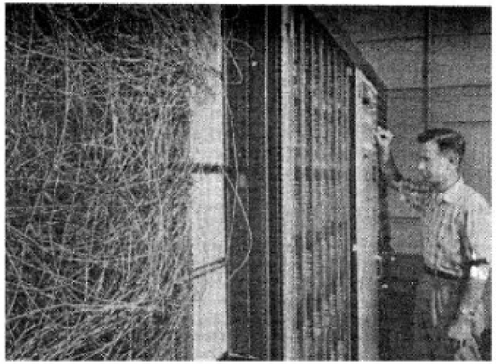
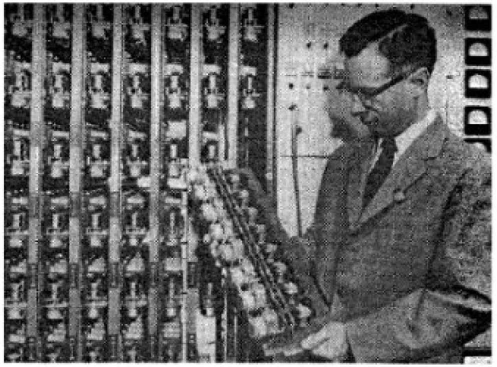
Deep Learning

Alfredo Canziani, Mengye Ren, Yann LeCun
NYU - Courant Institute & Center for Data Science



Inspiration for Deep Learning: The Brain!

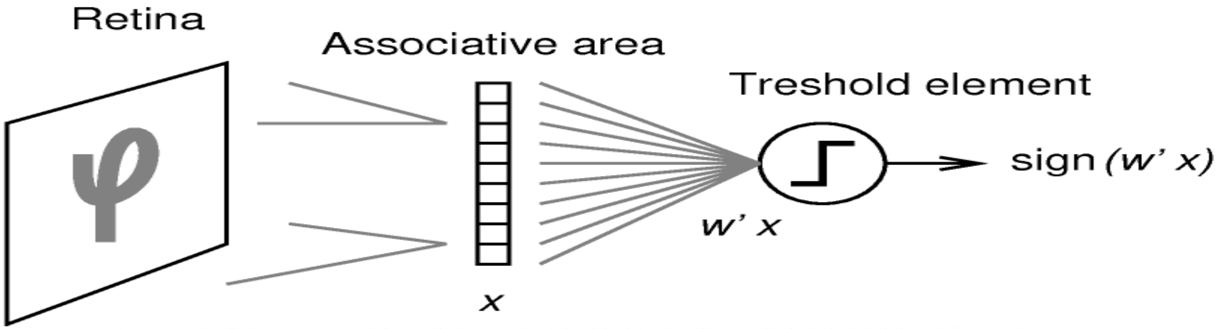
- **1943: McCulloch & Pitts, networks of binary neurons can do logic**
- **1947: Donald Hebb, Hebbian synaptic plasticity**
- **1948: Norbert Wiener, cybernetics, optimal filter, feedback, autopoïesis, self-organization.**
- **1957: Frank Rosenblatt, Perceptron**
- **1961: Bernie Widrow, Adaline**
- **1962: Hubel & Wiesel, visual cortex architecture**
- **1969: Minsky & Papert, limits of the Perceptron**

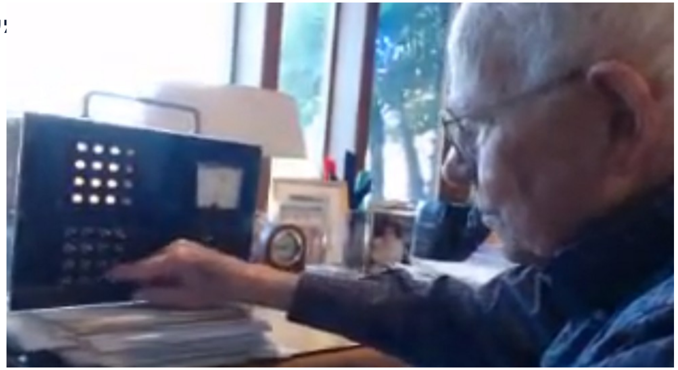







Supervised Learning goes back to the Perceptron & Adaline

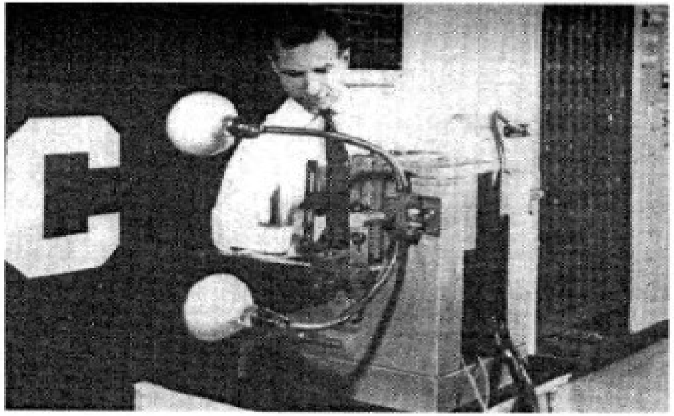
- **The McCulloch-Pitts Binary Neuron**
  - Perceptron: weights are motorized potentiometers
  - Adaline: Weights are electrochemical “memistors”

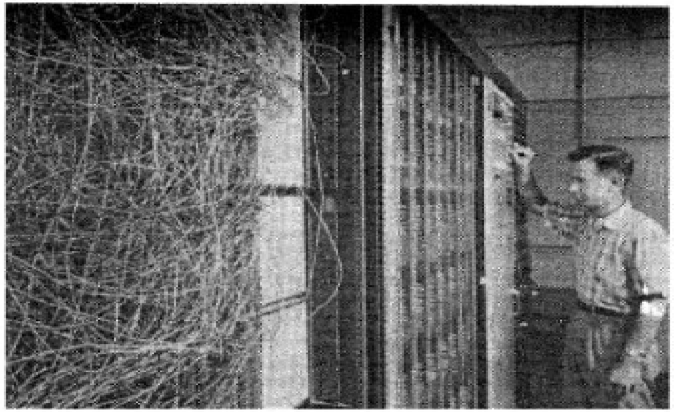
$$\tilde{y} = \text{sign}(b + \sum_{i=1}^{d} w_i x_i)$$

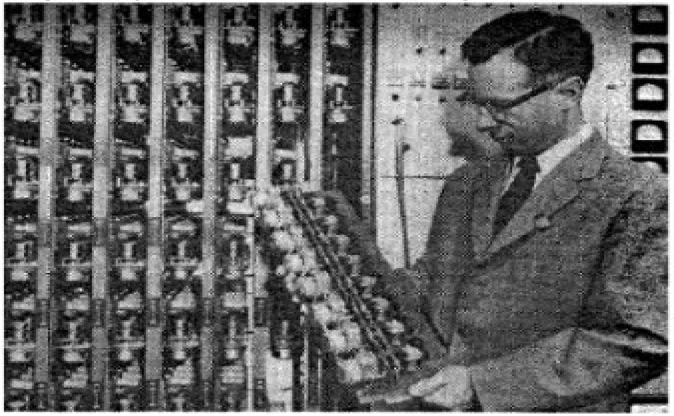




https://youtu.be/X1G2g3SiCwU







The Perceptron Learning Rule

- **Linear threshold unit / Linear classifier:**

$$\tilde{y} = \mathrm{sign}(b + \sum_{i=1}^{d} w_i x_i)$$

- **Learning by error correction:**
  - Intuition: if ỹ= y: do nothing.
  - If ỹ=-1 & y=+1: increase weights with positive input, decrease weights with negative input.
  - If ỹ=+1 & y=-1: decrease weights with positive input, increase weights with negative input.

$$w_i \leftarrow w_i + \underbrace{\eta}_{\text{Learning rate}}(\underbrace{y}_{\text{Desired output}} - \underbrace{\tilde{y}}_{\text{Predicted output}})x_i$$

- **Loss function**

$$L_{\mathrm{perc}}(x, y, w) = -(y - \tilde{y}) \sum_{j=1}^{d} w_j x_j$$

- **Gradient of the loss**

$$\frac{\partial L_{\mathrm{perc}}}{\partial w_i}(x, y, w) = -(y - \tilde{y})x_i$$

Adaline

- **Linear unit**
  - Thresholding is ignored for learning

$$\tilde{y} = b + \sum_{i=1}^{d} w_i x_i$$

- **Learning by error correction:**
  - If ỹ < y: increase weights with positive input, decrease weights with negative input.
  - If ỹ > y: decrease weights with positive input, increase weights with negative input.

$$w_i \leftarrow w_i + \eta(y - \tilde{y})x_i$$

Learning rate

Desired output

Predicted output

- **Loss function**

$$L_{\text{ada}}(x, y, w) = \frac{1}{2}(y - \tilde{y})^2 = \frac{1}{2}(y - \sum_{i=1}^{d} w_j x_j)^2$$

- **Gradient of the loss**

$$\frac{\partial L_{\text{ada}}}{\partial w_i}(x, y, w) = -(y - \tilde{y})x_i$$

Logistic Regression

- **Linear unit → tanh**

$$\tilde{y} = \tanh(b + \sum_{i=1}^{d} w_i x_i)$$

- **Learning by error correction:**
  - If ỹ < y: increase weights with positive input, decrease weights with negative input.
  - If ỹ > y: decrease weights with positive input, increase weights with negative input.

$$w_i \leftarrow w_i + \eta(y - \tilde{y})x_i$$

Learning rate

Desired output

Predicted output

- **Loss function**

$$L_{\text{logreg}}(x, y, w) = -2\log(1 + \exp(-y\sum_{j=1}^{d} w_j x_j))$$

- **Gradient of the loss**

$$\frac{\partial L_{\text{logreg}}}{\partial w_i}(x, y, w) = -(y - \tilde{y})x_i$$

Identical Learning Rules $w_i \leftarrow w_i + \eta(y - \tilde{y})x_i$

- **Perceptron**

$$\tilde{y} = \text{sign}(b + \sum_{i=1}^{d} w_i x_i) \qquad L_{\text{perc}}(x, y, w) = -(y - \tilde{y}) \sum_{j=1}^{d} w_j x_j$$

- **Adaline / LMS**

$$\tilde{y} = b + \sum_{i=1}^{d} w_i x_i \qquad L_{\text{ada}}(x, y, w) = \frac{1}{2}(y - \tilde{y})^2 = \frac{1}{2}(y - \sum_{i=1}^{d} w_j x_j)^2$$

- **Logistic Regression**

$$\tilde{y} = \tanh(b + \sum_{i=1}^{d} w_i x_i) \qquad L_{\text{logreg}}(x, y, w) = -2\log(1 + \exp(-y \sum_{j=1}^{d} w_j x_j))$$

More History

- 1970s: Statistical pattern recognition (Duda & Hart 1973)
- 1979: Kunihiko Fukushima, Neocognitron
- 1982: Hopfield Networks
- 1983: Hinton & Sejnowski, Boltzmann Machines
- 1985/1986: Practical Backpropagation for neural net training
- 1989: Convolutional Networks
- 1991: Bottou & Gallinari, module-based automatic differentiation
- 1995: Hochreiter & Schmidhuber, LSTM recurrent net.
- 1996: structured prediction with neural nets, graph transformer nets
- …..
- 2003: Yoshua Bengio, neural language model
- 2006: Layer-wise unsupervised pre-training of deep networks
- 2010: Collobert & Weston, self-supervised neural nets in NLP

Hopfield Net, Boltzmann Machine

- **Recurrent networks with symmetric weights.**
  - Dynamics settles in a local minimum of an energy function

$$E(y) = -\sum_{ij} y_i w_{ij} y_j \qquad\qquad y_i \leftarrow \tanh(\sum_j w_i j x_j)$$

$$w_{ij} \leftarrow w_{ij} + \eta y_i y_j$$

$$w_{ij} \leftarrow w_{ij} + \eta (y_i y_j - \tilde{y}_i \tilde{y}_j)$$

More History

- 2012: AlexNet / convnet on GPU / object classification
- 2015: I. Sutskever, neural machine translation with multilayer LSTM
- 2015: Weston, Chopra, Bordes: Memory Networks
- 2016: Bahdanau, Cho, Bengio: GRU, attention mechanism
- 2016: Kaiming He: ResNet
- 2017: He, Gkioxari, Dollar, Girshick: Mask R-CNN
- 2017: Waswani et al.: Transformer architecture
- 2017: graph NN / geometric DL (Bruna, Bronstein, Welling, ...)
- 2018: Devlin et al.: BERT / denoising AE pretraining for NLP
- 2019: Conneau et al. XLM-Roberta
- 2020: Transformers in vision, e.g. ViT, DETR by Carion et al.
- 2020: Self-supervised learning with joint-embedding architectures
- 2021+: too many to mention!

Parameterized Model

- **Parameterized model**

$$\bar{y} = G(x, w)$$

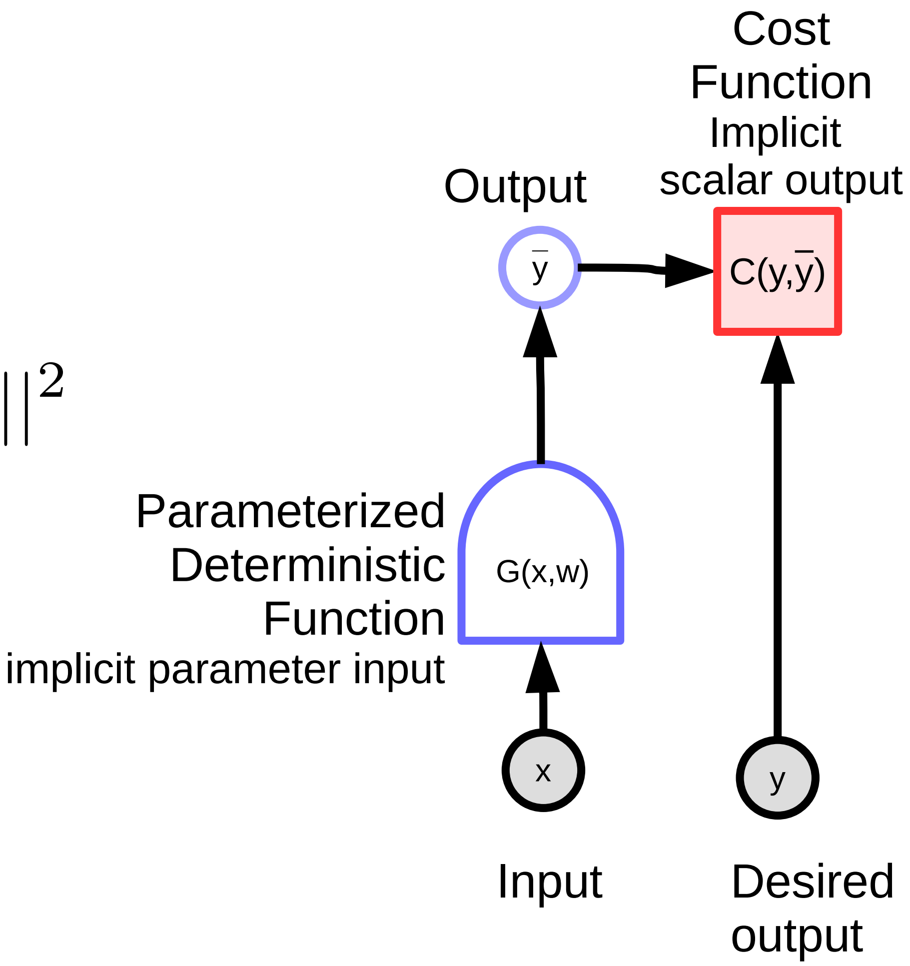
- Example: linear regression

$$\bar{y} = \sum_i w_i x_i \quad C(y, \bar{y}) = ||y - \bar{y}||^2$$

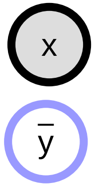
- Example: Nearest neighbor:

$$\bar{y} = \text{argmin}_k ||x - w_{k,.}||^2$$

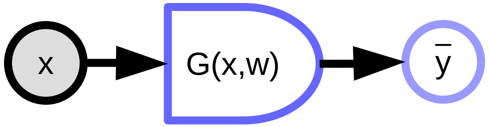
- Computing function G may involve complicated algorithms, e.g. optimization, search, ...



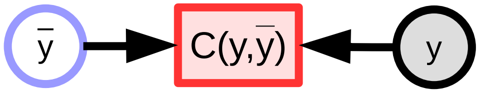
Block diagram notations for computation graphs



- **Variables (tensor, scalar, continuous, discrete...)**
  - Observed: input, desired output…
  - Computed variable: outputs of deterministic functions



- **Deterministic function**
  - Multiple inputs and outputs (tensors, scalars,….)
  - Implicit parameter variable (here: w)



- **Scalar-valued function (implicit output)**
  - Single scalar output (implicit)
  - used mostly for cost functions

Loss function, average loss.

- **Simple per-sample loss function**

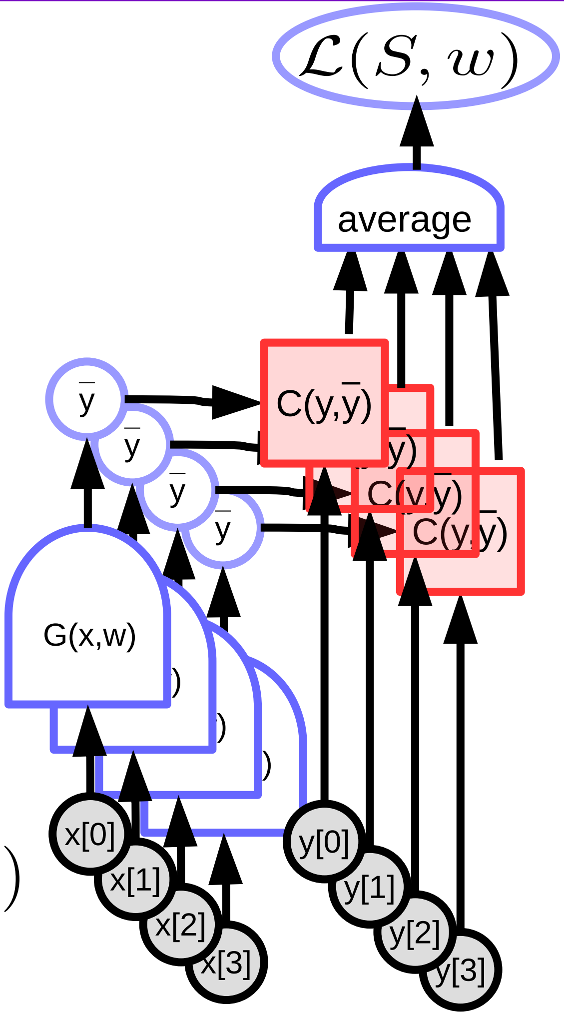
$$L(x, y, w) = C(y, G(x, w))$$

- **A set of samples**

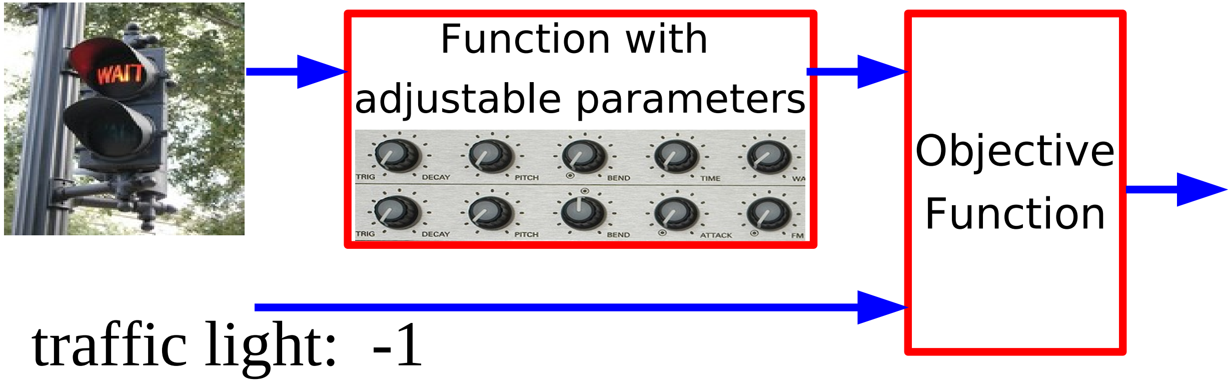
$$S = \{(x[p], y[p]) \;/\; p = 0 \dots P-1\}$$

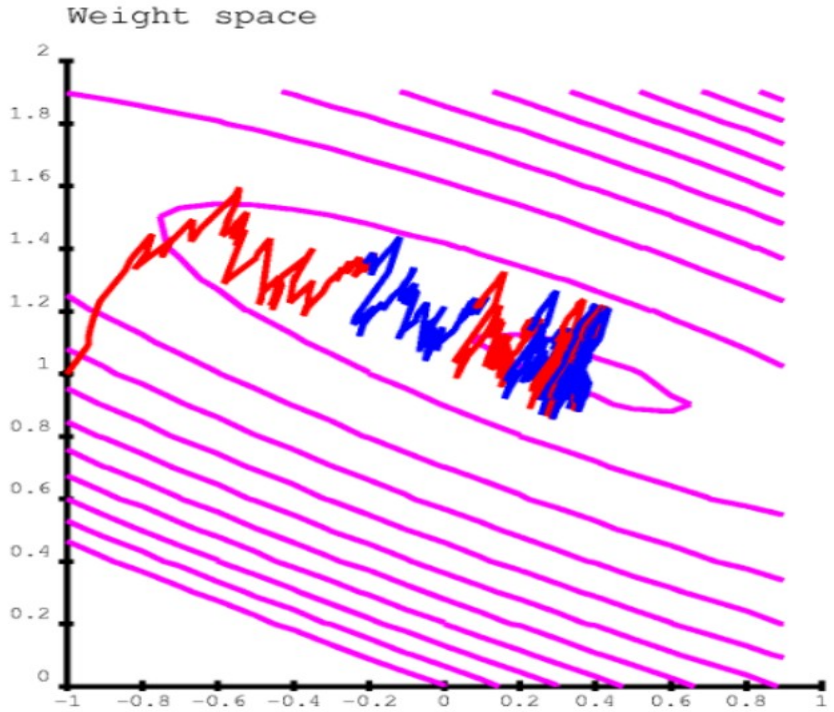
- **Average loss over the set**

$$\mathcal{L}(S, w) = \frac{1}{P} \sum_{(x,y)} L(x, y, w) = \frac{1}{P} \sum_{p=0}^{P-1} L(x[p], y[p], w)$$



Supervised Machine Learning = Function Optimization





- **It's like walking in the mountains in a fog and following the direction of steepest descent to reach the village in the valley**
- **But each sample gives us a noisy estimate of the direction. So our path is a bit random.**

$$w \leftarrow w - \eta \frac{\partial L(x[p], y[p], w)}{\partial w}$$

Gradient Descent

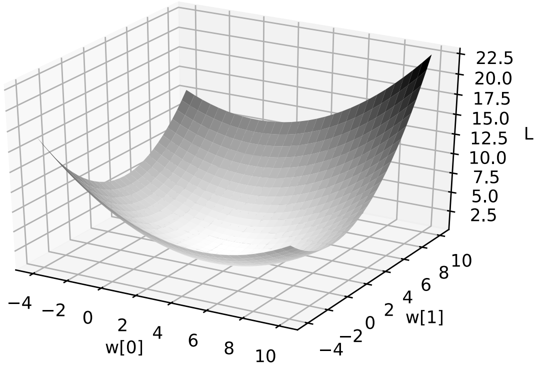
- **Full (batch) gradient**

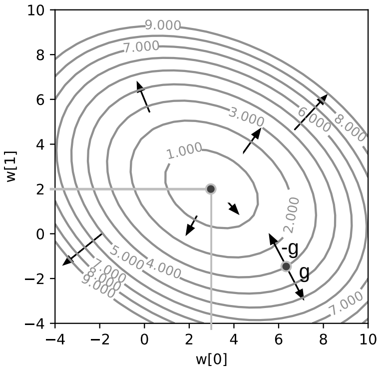
$$w \leftarrow w - \eta \frac{\partial \mathcal{L}(S, w)}{\partial w}$$

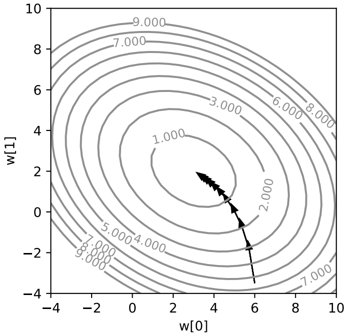
- **Stochastic Gradient (SGD)**
  - Pick a p in 0...P-1, then update w:

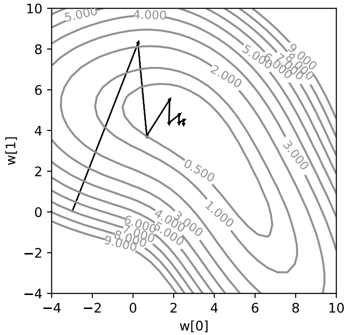
$$w \leftarrow w - \eta \frac{\partial L(x[p], y[p], w)}{\partial w}$$

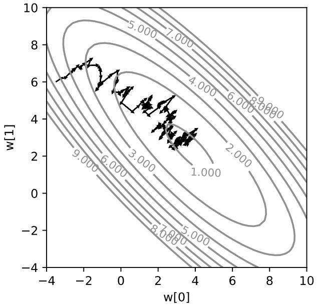
- **SGD exploits the redundancy in the samples**
  - It goes faster than full gradient in most cases
  - In practice, we use mini-batches for parallelization.





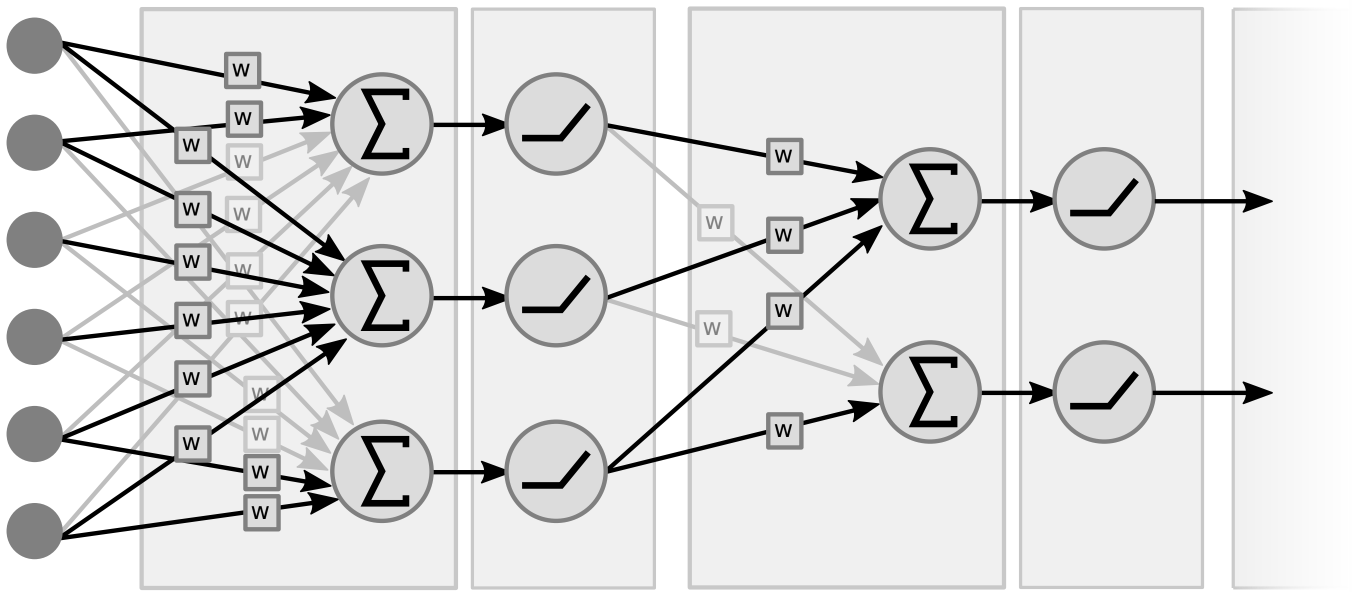






Traditional Neural Net

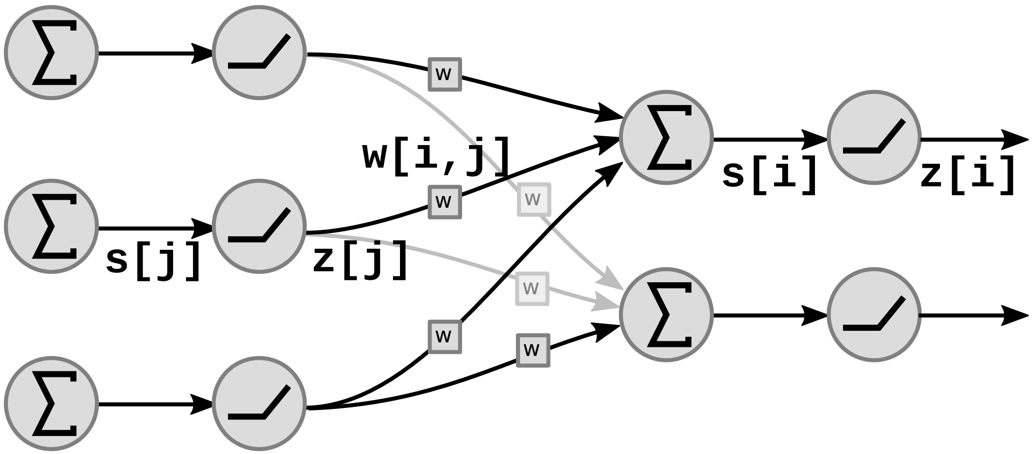
- **Stacked linear and non-linear functional blocks**
  - Weighted sums, matrix-vector product
  - Point-wise non-linearities (e.g. ReLu, tanh, ….)



Traditional Neural Net

- **Stacked linear and non-linear functional blocks**

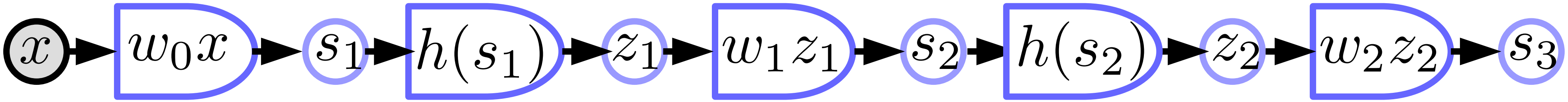
$$s[i] = \sum_{j \in \mathrm{UP}(i)} w[i,j] \cdot z[j] \qquad\qquad z[i] = f(s[i])$$



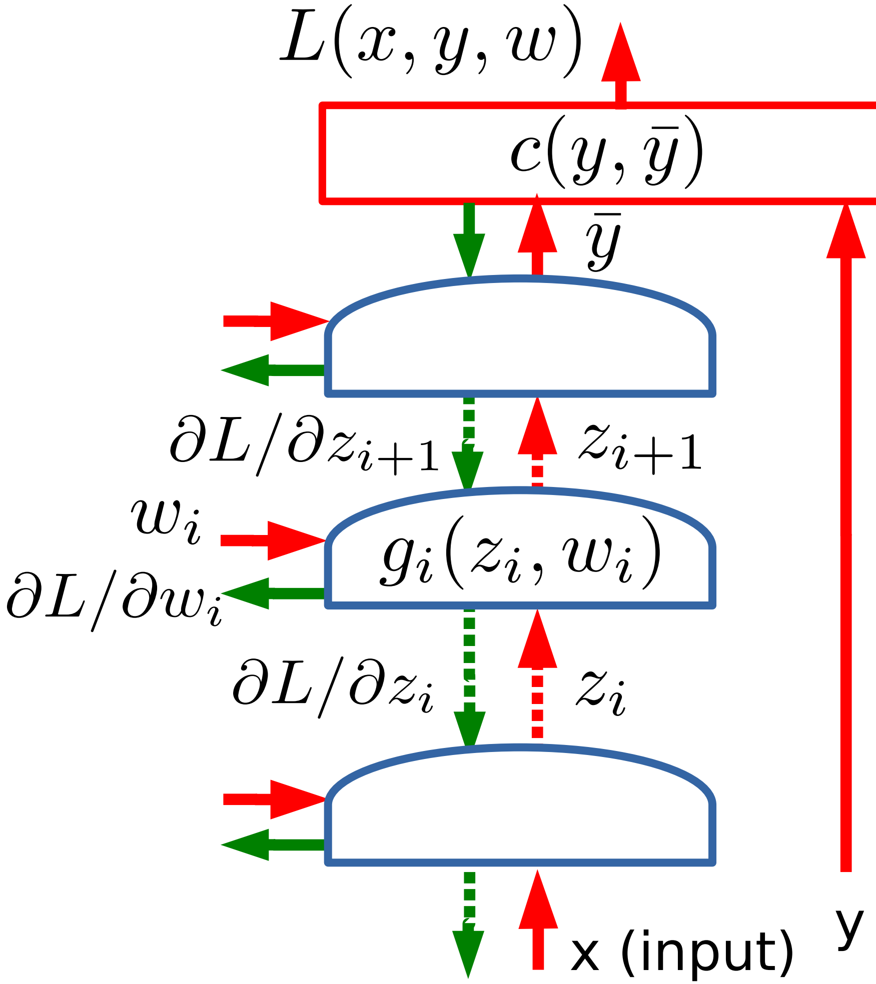
Block Diagram of a Traditional Neural Net

- **linear blocks** $$s_{k+1} = w_k z_k$$

- **Non-linear blocks** $$z_k = h(s_k)$$



The main trick of Deep Learning: Gradient Back-propagation



- **A practical Application of Chain Rule**
- **Backprop for the state gradients:**

$$\frac{\partial L}{\partial z_i} = \frac{\partial L}{\partial z_{i+1}} \frac{\partial z_{i+1}}{\partial z_i} = \frac{\partial L}{\partial z_{i+1}} \frac{\partial g_i(z_i, w_i)}{\partial z_i}$$

- **Backprop for the weight gradients:**

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial z_{i+1}} \frac{\partial z_{i+1}}{\partial w_i} = \frac{\partial L}{\partial z_{i+1}} \frac{\partial g_i(z_i, w_i)}{\partial w_i}$$

Backprop through a functional module

- **Using chain rule for vector functions**

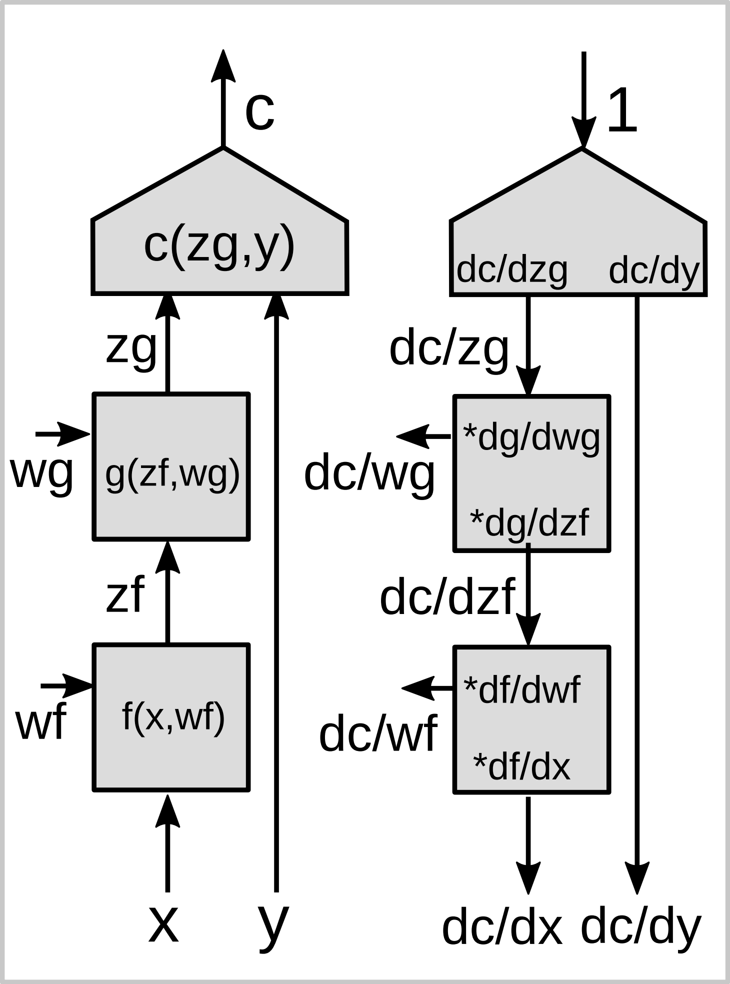
$$z_g : [d_g \times 1] \quad z_f : [d_f \times 1]$$

$$\frac{\partial c}{\partial z_f} = \frac{\partial c}{\boxed{\partial z_g}} \frac{\boxed{\partial z_g}}{\partial z_f}$$

$$[1 \times d_f] = [1 \times d_g] * [d_g \times d_f]$$

- **Jacobian matrix**
  - Partial derivative of i-th output w.r.t. j-th input

$$\left(\frac{\partial z_g}{\partial z_f}\right)_{ij} = \frac{(\partial z_g)_i}{(\partial z_f)_j}$$



Chain Rule

- **Chain rule**

$$g(f(x))' = g'(f(x))f'(x)$$

$$\frac{\partial c}{\partial z_f} = \frac{\partial c}{\boxed{\partial z_g}} \frac{\boxed{\partial z_g}}{\partial z_f}$$

Backprop through a non-linear function

- **Chain rule:**

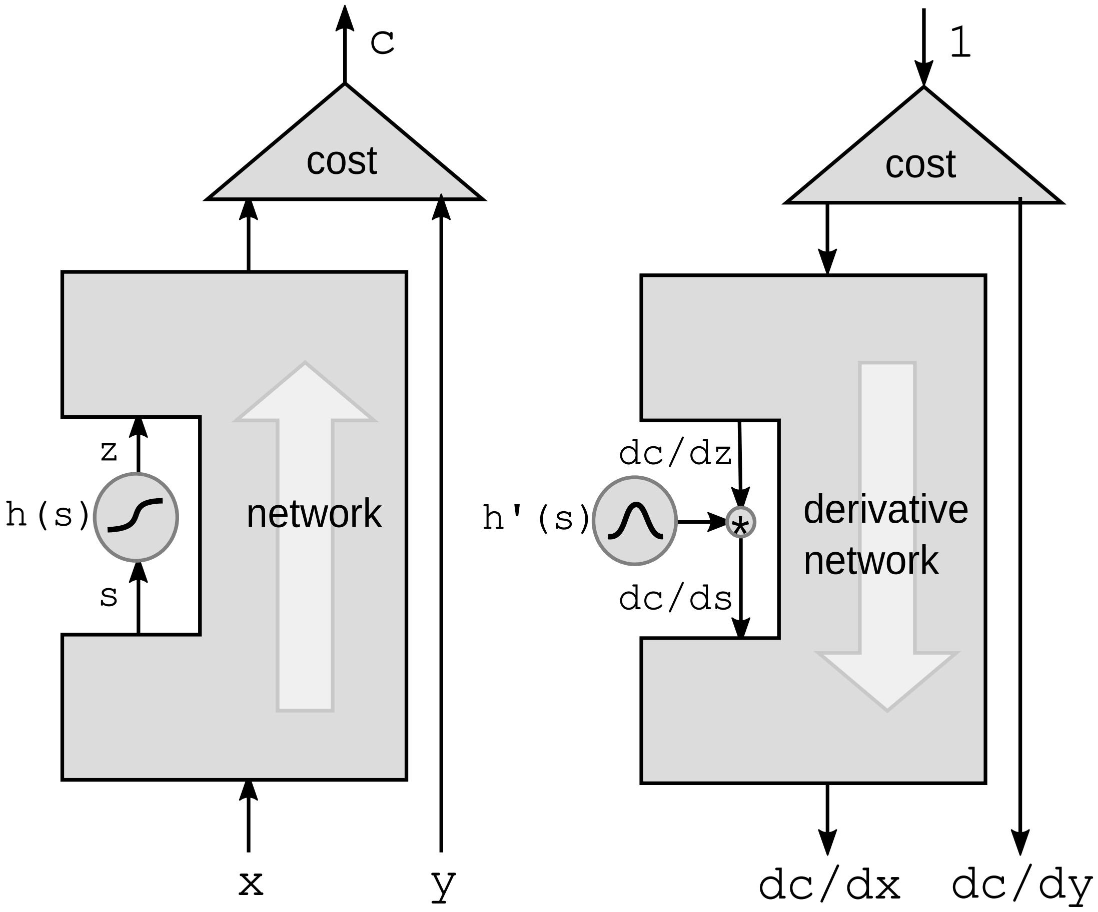
  g(h(s))’ = g’(h(s)).h’(s)

  dc/ds = dc/dz*dz/ds

  dc/ds = dc/dz*h’(s)

- **Perturbations:**
  - Perturbing s by ds will perturb z by: dz=ds*h’(s)
  - This will perturb c by

    dc = dz*dc/dz = ds*h’(s)*dc/dz
  - Hence: dc/ds = dc/dz*h’(s)



Backprop through a weighted sum

- **Perturbations:**
  - Perturbing z by dz will perturb s[0],s[1],s[2] by ds[0]=w[0]*dz, ds[1]=w[1]*dz, ds[2]=w[2]*dz
  - This will perturb c by

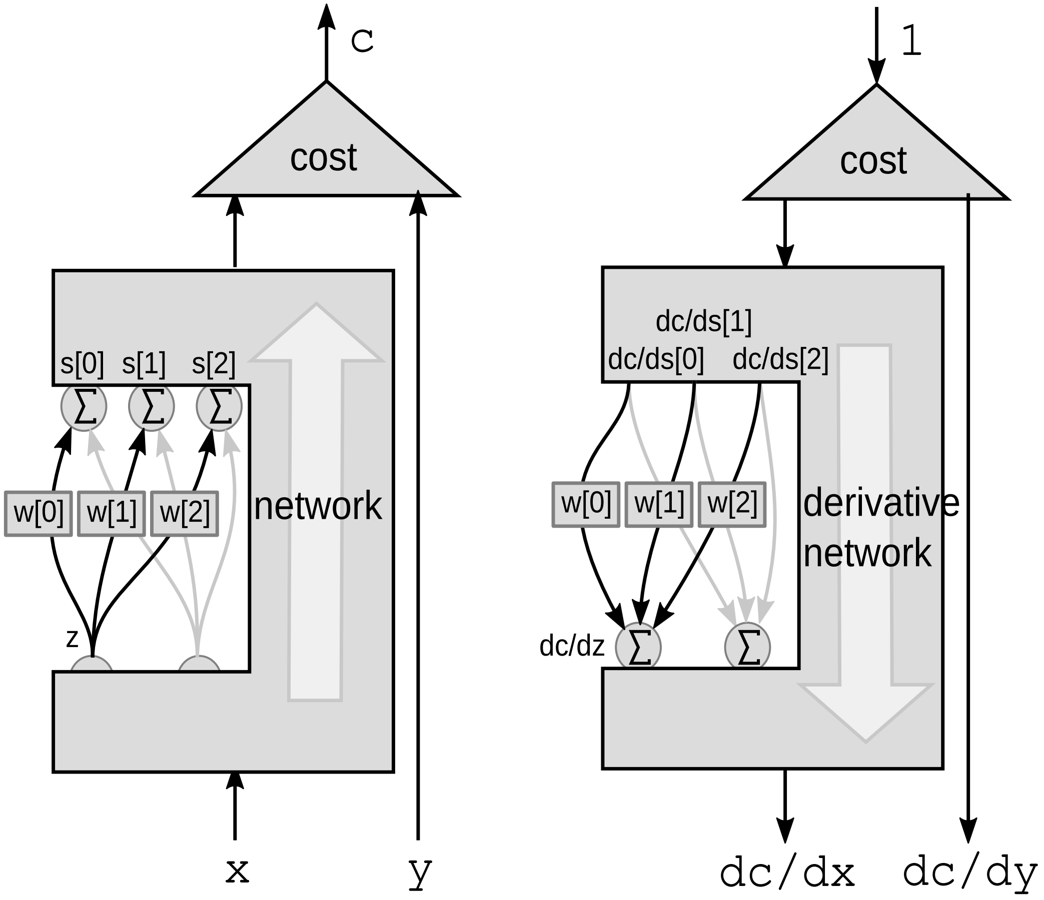
    dc = ds[0]*dc/ds[0]+

    ds[1]*dc/ds[1]+

    ds[2]*dc/ds[2]
  - Hence: dc/dz = dc/ds[0]*w[0]+

    dc/ds[1]*w[1]+

    dc/ds[2]*w[2]+



Backprop through a functional module

- **Using chain rule for vector functions**

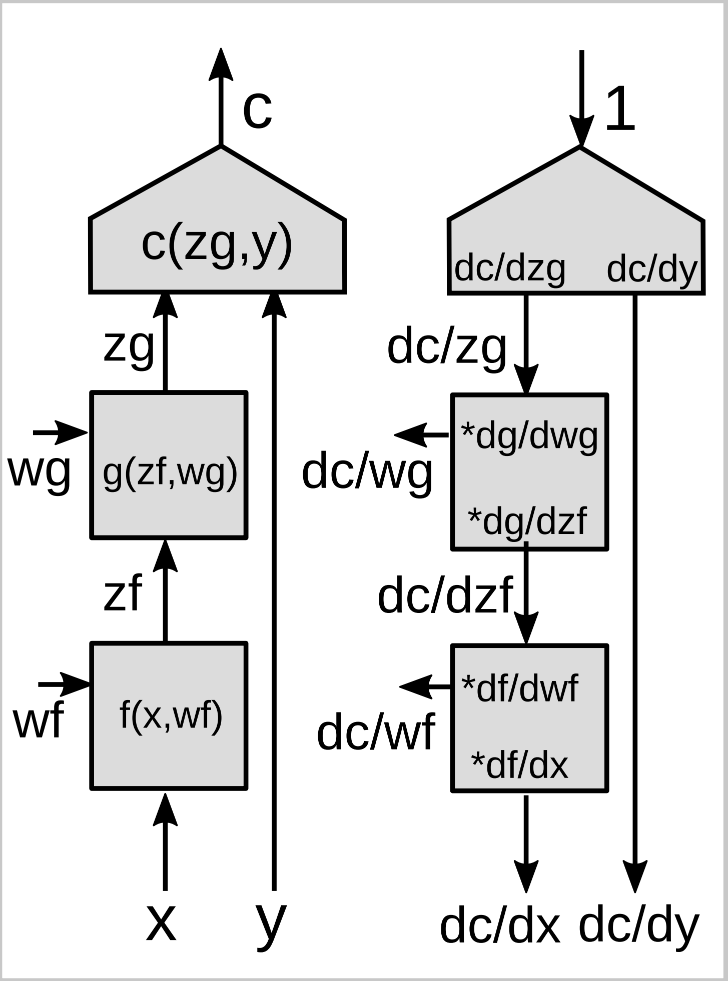
$$z_g : [d_g \times 1] \quad z_f : [d_f \times 1]$$

$$\frac{\partial c}{\partial z_f} = \frac{\partial c}{\boxed{\partial z_g}} \frac{\boxed{\partial z_g}}{\partial z_f}$$

$$[1 \times d_f] = [1 \times d_g] * [d_g \times d_f]$$

- **Jacobian matrix**
  - Partial derivative of i-th output w.r.t. j-th input

$$\left(\frac{\partial z_g}{\partial z_f}\right)_{ij} = \frac{(\partial z_g)_i}{(\partial z_f)_j}$$



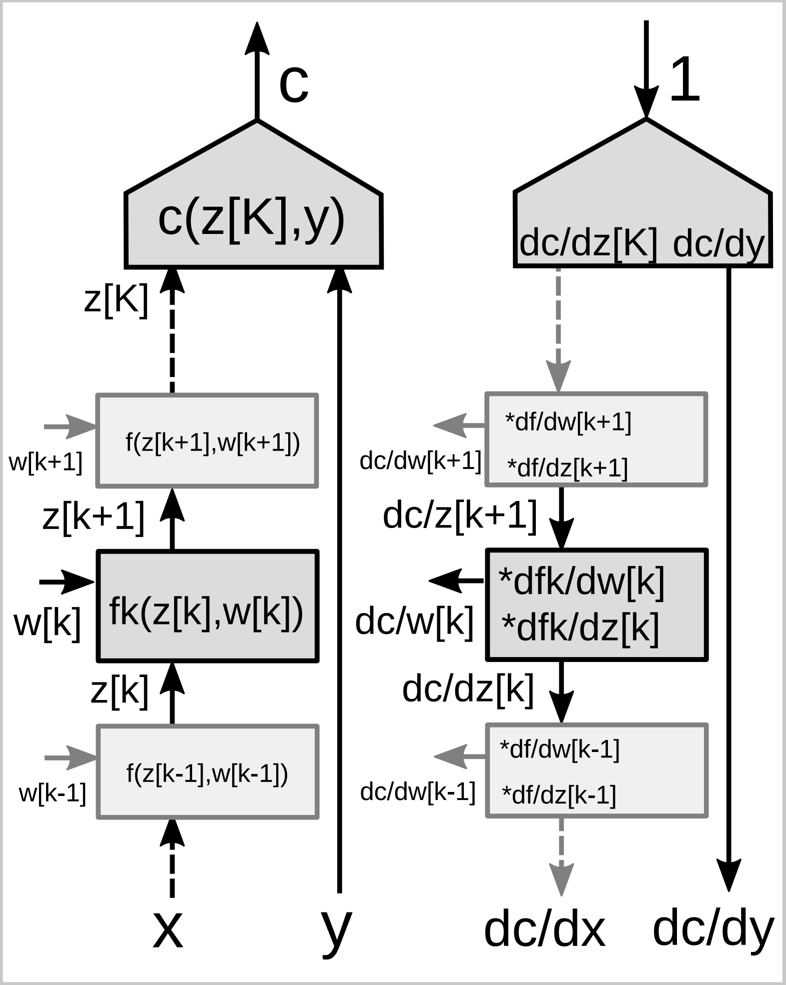
Backprop through a multi-stage graph

- **Using chain rule for vector functions**

$$\frac{\partial c}{\partial z_k} = \frac{\partial c}{\partial z_{k+1}} \frac{\partial z_{k+1}}{\partial z_k} = \frac{\partial c}{\partial z_{k+1}} \frac{\partial f_k(z_k, w_k)}{\partial z_k}$$

$$\frac{\partial c}{\partial w_k} = \frac{\partial c}{\partial z_{k+1}} \frac{\partial z_{k+1}}{\partial w_k} = \frac{\partial c}{\partial z_{k+1}} \frac{\partial f_k(z_k, w_k)}{\partial w_k}$$

- **Two Jacobian matrices for the module:**
  - One with respect to z[k]
  - One with respect to w[k]



- **End of Lecture 1**