Non-linear Models-VI

CS771: Introduction to Machine Learning

Purushottam Kar

Answering the Fan Mail

- None this week.
- Assignment 3 is out. Deadline Nov 14, 2359 hrs
- Discussion session this Sunday, Nov 5, 6PM, RM101
- Please submit questions to http://tinyurl.com/ml17-18ads2
- Please submit questions latest by Friday, Nov 3
- Please (re)upload your project proposals to GS by Sun, Nov 5
- Make sure all teammates are linked to the (group) submission

Recap

The Generalized Perceptron

- Simply a linear model will a wrapper thrown around it
- Makes predictions as

$$\hat{y} = f(\langle \mathbf{w}, \mathbf{x} \rangle)$$

- Given lots of data points

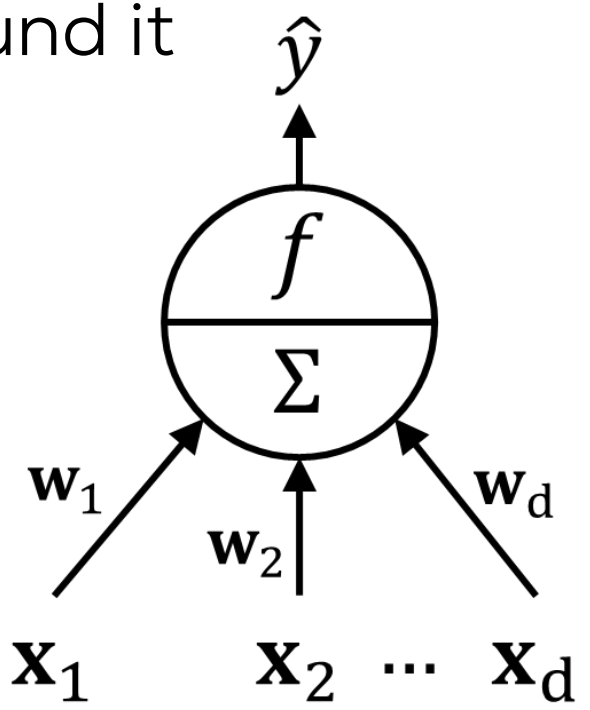
$$(\mathbf{x}^1, y^1), \ldots, (\mathbf{x}^n, y^n), \mathbf{x}^i \in \mathbb{R}^d$$

- … and a loss function

$$\ell \colon \mathbb{R} \times \mathbb{R} \to \mathbb{R}_+$$

- … training a perceptron involves finding

$$\arg\min_{\mathbf{w} \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} \ell\left(f\left(\langle \mathbf{w}, \mathbf{x}^i \rangle\right), y^i\right) =: \arg\min_{\mathbf{w} \in \mathbb{R}^d} F(\mathbf{w})$$



Gradient Descent Revisited

GRADIENT DESCENT

1. Initialize $\mathbf{w}^0$
2. For $t = 1, 2, \ldots$
   1. Obtain a descent direction $\mathbf{g}^t$
   2. Update $\mathbf{w}^{t+1} \leftarrow \mathbf{w}^t - \eta_t \cdot \mathbf{g}^t$
3. Repeat until convergence

- How to find a descent direction?
- How to choose a step length?
- How to detect convergence?
- How to avoid overfitting?

Have to be more careful than earlier since now, problems are not nicely behaved

Choosing a descent direction

$$\arg\min_{\mathbf{w}\in\mathbb{R}^d} \frac{1}{n}\sum_{i=1}^{n} \ell\left(f\left(\left\langle \mathbf{w}, \mathbf{x}^i \right\rangle\right), y^i\right) =: \arg\min_{\mathbf{w}\in\mathbb{R}^d} F(\mathbf{w})$$

- Batch gradient

$$\mathbf{g}^t = \nabla F(\mathbf{w}^t) = \frac{1}{n}\sum_{i=1}^{n} \ell'\left(f\left(\left\langle \mathbf{w}^t, \mathbf{x}^i \right\rangle\right), y^i\right) \cdot f'\left(\left\langle \mathbf{w}^t, \mathbf{x}^i \right\rangle\right) \cdot \mathbf{x}^i$$

- Mini-batch gradient: choose a mini-batch $I_1^t, I_2^t, \ldots, I_B^t \sim [n]$

$$\mathbf{g}^t = \frac{1}{B}\sum_{j=1}^{B} \ell'\left(f\left(\left\langle \mathbf{w}^t, \mathbf{x}^{I_j^t} \right\rangle\right), y^i\right) \cdot f'\left(\left\langle \mathbf{w}^t, \mathbf{x}^{I_j^t} \right\rangle\right) \cdot \mathbf{x}^{I_j^t}$$

- Newton's method

$$\mathbf{g}^t = \left(\nabla^2 F(\mathbf{w}^t)\right)^{-1} \nabla F(\mathbf{w}^t)$$

Choosing a descent direction

$$\arg\min_{\mathbf{w}\in\mathbb{R}^d} \frac{1}{n}\sum_{i=1}^{n} \ell\left(f\left(\left\langle \mathbf{w}, \mathbf{x}^i \right\rangle\right), y^i\right) =: \arg\min_{\mathbf{w}\in\mathbb{R}^d} F(\mathbf{w})$$

- Batch gradient

$$\mathbf{g}^t = \nabla F(\mathbf{w}^t) = \frac{1}{n}\sum_{i=1}^{n} \ell'\left(f\left(\left\langle \mathbf{w}^t, \mathbf{x}^i \right\rangle\right), y^i\right) \cdot f'\left(\left\langle$$

Chain rule!

Very small batch sizes usually not used for deep networks

- Mini-batch gradient: choose a mini-batch $I_1^t, I_2^t, \ldots, I_B^t \sim [n]$

$$\mathbf{g}^t = \frac{1}{B}\sum_{j=1}^{B} \ell'\left(f\left(\left\langle \mathbf{w}^t, \mathbf{x}^{I_j^t} \right\rangle\right), y^i\right) \cdot f'\left(\left\langle \mathbf{w}^t \right.\right.$$

For a NN with E edges, $\mathcal{O}(E^3)$ time per iteration!

- Newton's method

$$\mathbf{g}^t = \left(\nabla^2 F(\mathbf{w}^t)\right)^{-1} \nabla F(\mathbf{w}^t)$$

Expensive! $\mathcal{O}(d^3)$ time per iteration

How to detect convergence

- Tolerance technique
  - For a predecided tolerance value ϵ, if $F(\mathbf{w}^t) < \epsilon$, stop
- Zero-th order technique
  - If function value has not changed too much between iterations, stop!

$$|F(\mathbf{w}^{t+1}) - F(\mathbf{w}^t)| < \tau$$

- First order technique
  - If gradient is too "small" $\|\nabla F(\mathbf{w}^t)\|_2 < \delta$, stop!
- Primal dual
  - If primal and dual objective values are close, stop
  - Does not work every where – reliable for convex problems

How to decide step length?

- Choose $\eta_t \to 0$ (diminishing) and $\sum \eta_t \to \infty$ (infinite travel)
- Example $\eta_t = C/\sqrt{t}$ or $\eta_t = C/t$ for some $C > 0$
- **Line search** – super careful but expensive

$$\eta_t = \arg\min_{\eta \geq 0} F(\mathbf{w}^t - \eta \cdot \mathbf{g}^t)$$

- **Momentum methods**: don't let the procedure take zig-zag routes
  - Nesterov's Accelerated Gradient (NAG)
- **Adaptive step length**: control how much each coordinate of $\mathbf{w}$ gets updated. Slow down coordinates getting too many updates and speed up training along coordinates not getting updated
  - Adagrad, RMSProp, Adam

How to prevent overfitting?

- Add a regularization term L_2/L_1 to the objective

$$\arg\min_{\mathbf{w}\in\mathbb{R}^d} F(\mathbf{w}) + \lambda \cdot \|\mathbf{w}\|_2^2$$

- Gradient/step length calculations still easy to perform
- Constraint the weights of the network to satisfy $|\mathbf{w}_i| < r$ for all i

$$\arg\min_{\|\mathbf{w}\|_\infty < r} F(\mathbf{w})$$

- Sometimes gradient coordinates are also *clipped* this way
- Noise injection in output
  - For binary classification $y^i = 0 \rightarrow y^i = \epsilon$, $y^i = 1 \rightarrow y^i = 1 - \epsilon$
  - For regression problems, $y^i \rightarrow y^i + \epsilon^i$, where $\epsilon^i \sim \mathcal{N}(0, \sigma^2)$
  - Can be shown to be equivalent to regularization in nice cases

How to prevent overfitting?

- Early stopping – return model with best validation set performance rather than best training set performance
- Can use many of these strategies in combination
- Parameter sharing – add constraints of the form

$$\mathbf{w}_i = \mathbf{w}_j$$

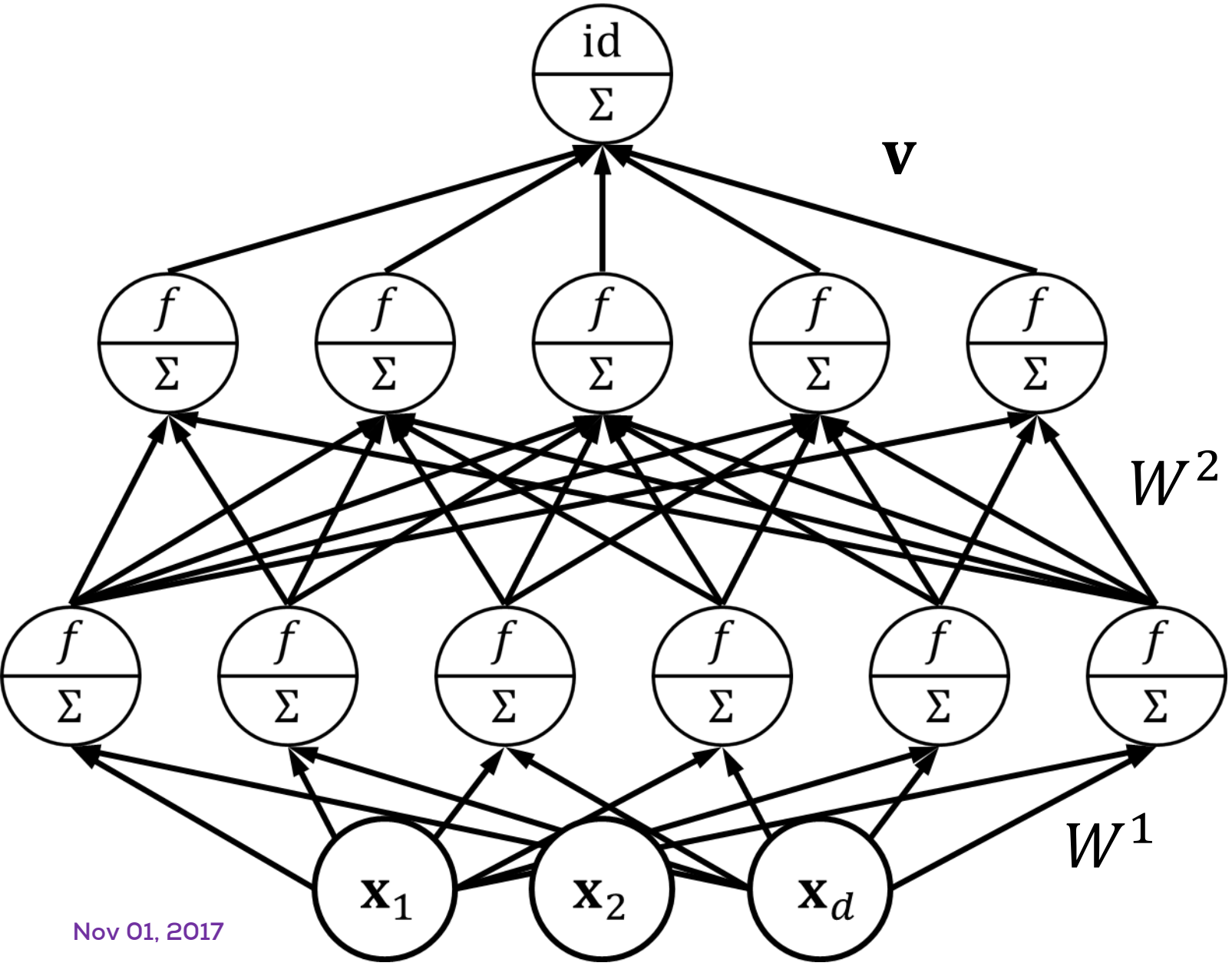
- Sparse recovery – constrain, say at least 10% weights to be zero

$$\|\mathbf{w}\|_0 \leq k \ll d$$

- Dropout – will see in a short while!

Multilayer Perceptron

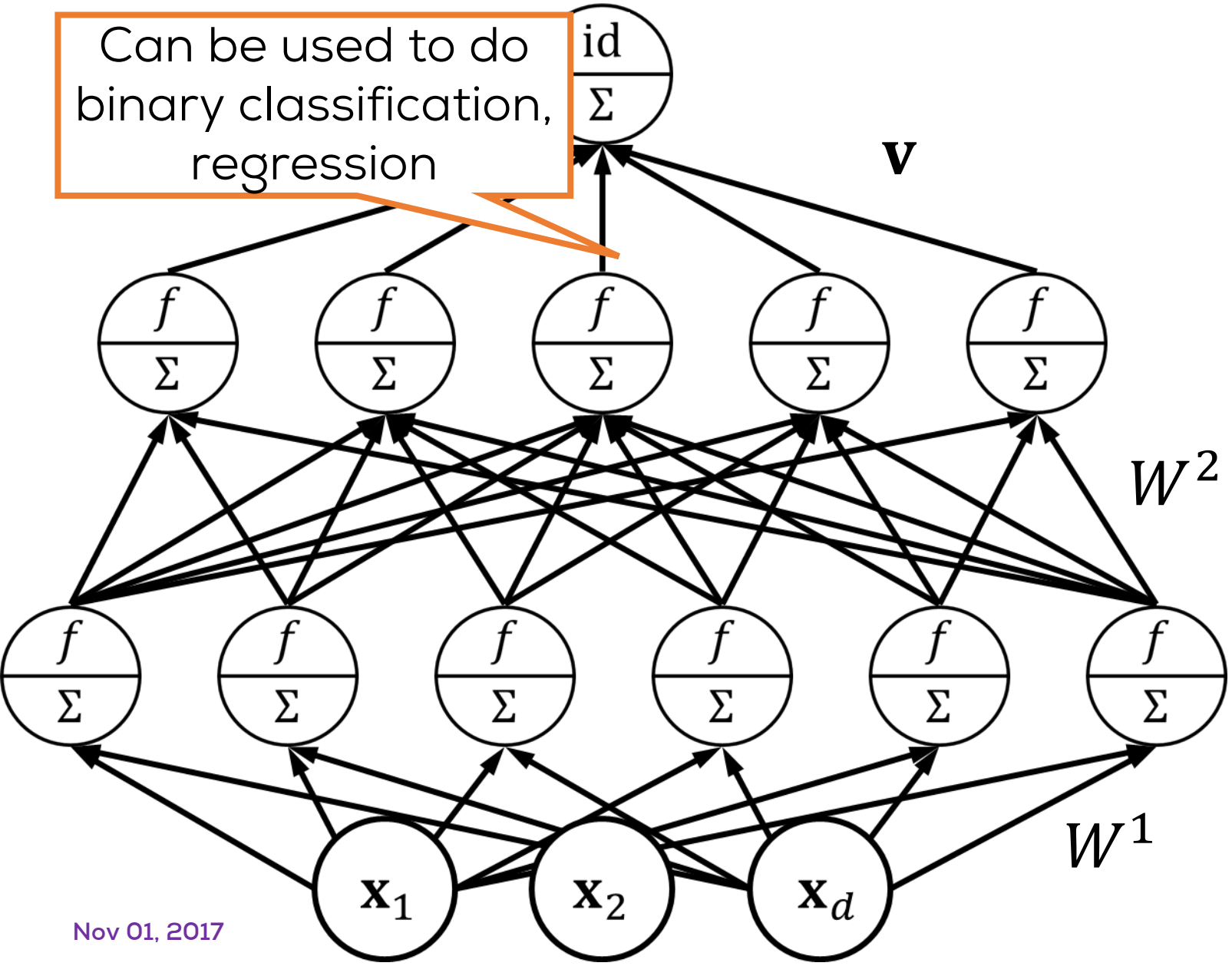
Feedforward Neural Network



- d inputs, L hidden layers
- k_l nodes in l-th layer
- $W^l \in \mathbb{R}^{k_{l-1} \times k_l}$ weights from layer $l-1$ to l
- $\mathbf{v} \in \mathbb{R}^{k_L}$ weights from layer L to output node
- $\mathbf{h}^l \in \mathbb{R}^{k_l}$ o/p by layer l
- $\hat{y}$ output of the network
- $\mathbf{h}^l = f\left(\left(W^l\right)^\top \mathbf{h}^{l-1}\right)$
- $f(\mathbf{u}) = [f(\mathbf{u}_1), \ldots, f(\mathbf{u}_k)]$
- $\hat{y} = \langle \mathbf{v}, \mathbf{h}^L \rangle$

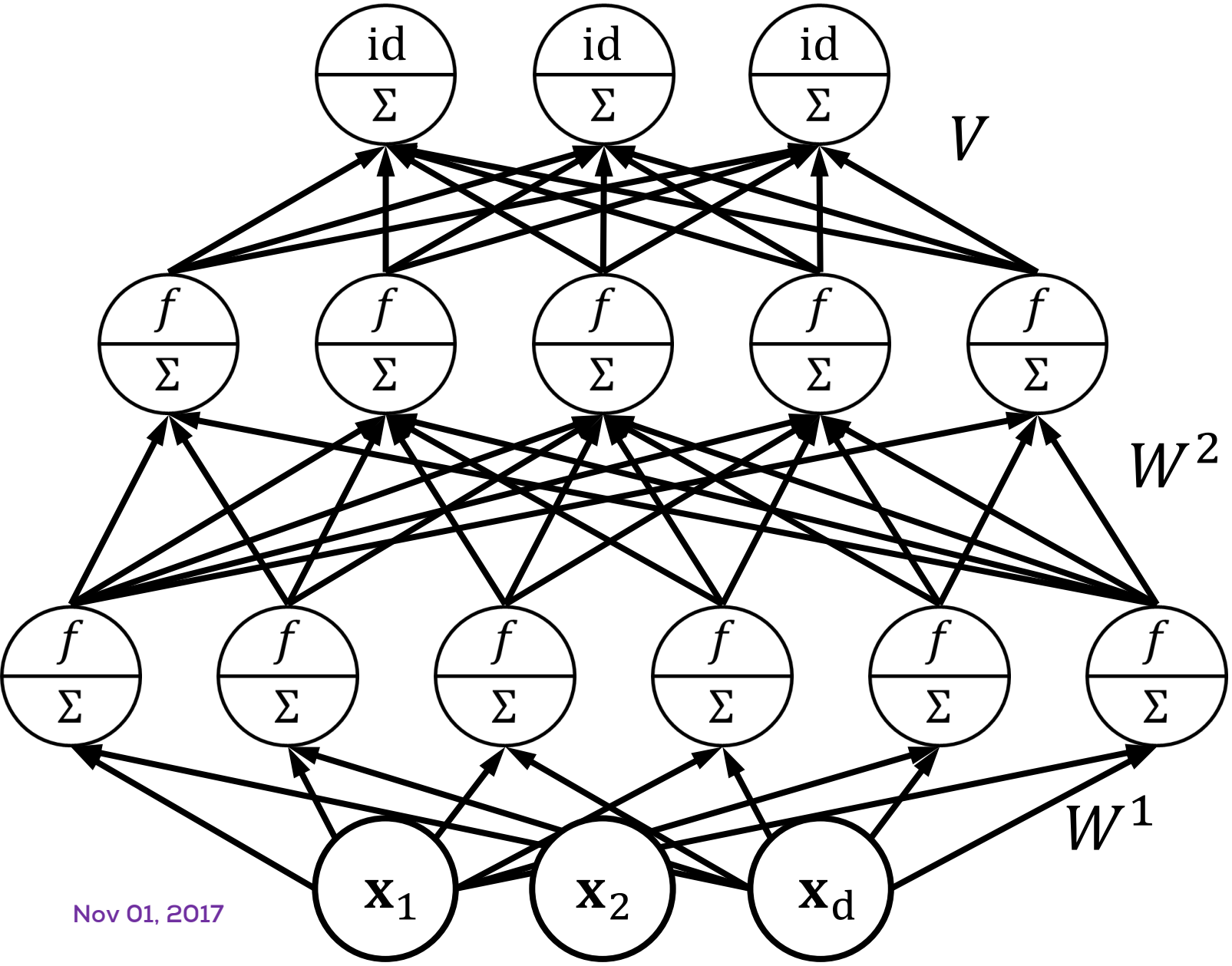
Feedforward Neural Network

Can be used to do binary classification, regression



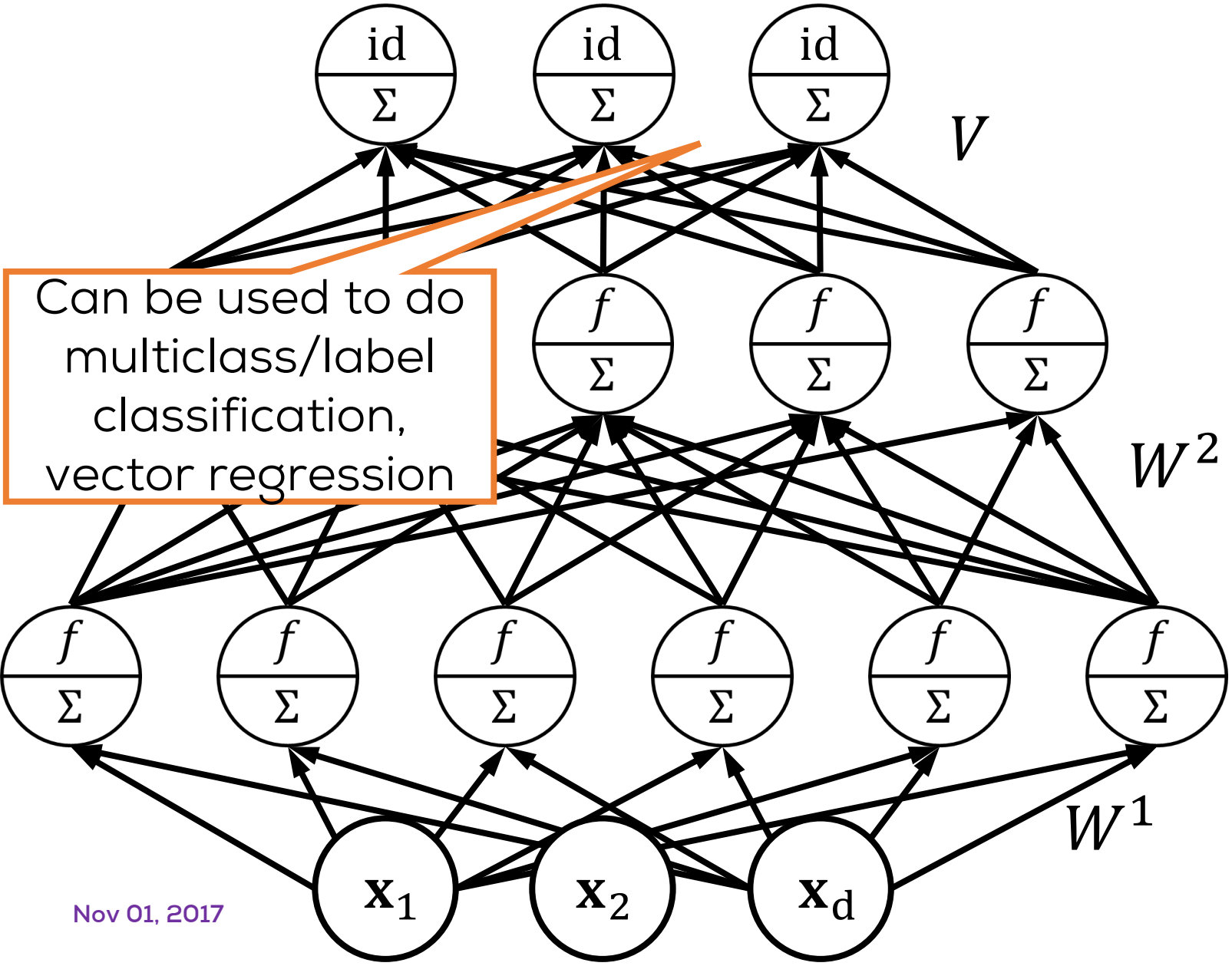
- d inputs, L hidden layers
- k_l nodes in l-th layer
- $W^l \in \mathbb{R}^{k_{l-1} \times k_l}$ weights from layer $l-1$ to l
- $\mathbf{v} \in \mathbb{R}^{k_L}$ weights from layer L to output node
- $\mathbf{h}^l \in \mathbb{R}^{k_l}$ o/p by layer l
- $\hat{y}$ output of the network
- $\mathbf{h}^l = f\left(\left(W^l\right)^\top \mathbf{h}^{l-1}\right)$
- $f(\mathbf{u}) = [f(\mathbf{u}_1), \ldots, f(\mathbf{u}_k)]$
- $\hat{y} = \langle \mathbf{v}, \mathbf{h}^L \rangle$

Feedforward Neural Network



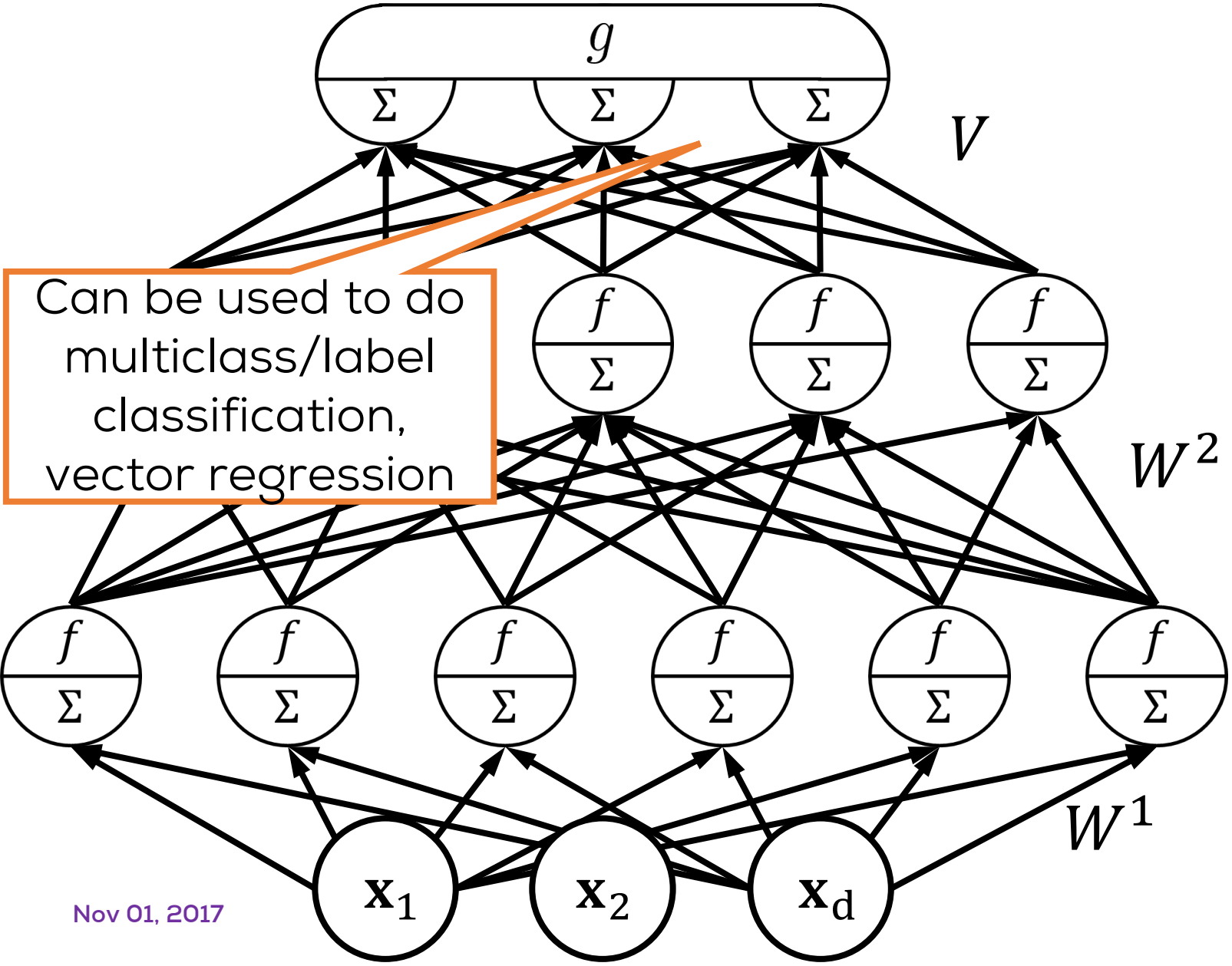
- C output nodes
- $\hat{\mathbf{y}} \in \mathbb{R}^C$ o/p of network
- $V \in \mathbb{R}^{k_L \times C}$ weights from layer L to output layer
- $\hat{\mathbf{y}} = V^\top \mathbf{h}^L$
- … or else $\hat{\mathbf{y}} = g(V^\top \mathbf{h}^L)$
- E.g. g: softmax, maxout

Feedforward Neural Network



- C output nodes
- $\hat{\mathbf{y}} \in \mathbb{R}^C$ o/p of network
- $V \in \mathbb{R}^{k_L \times C}$ weights from layer L to output layer
- $\hat{\mathbf{y}} = V^\top \mathbf{h}^L$
- … or else $\hat{\mathbf{y}} = g(V^\top \mathbf{h}^L)$
- E.g. g: softmax, maxout

Feedforward Neural Network



- C output nodes
- $\hat{\mathbf{y}} \in \mathbb{R}^C$ o/p of network
- $V \in \mathbb{R}^{k_L \times C}$ weights from layer L to output layer
- $\hat{\mathbf{y}} = V^\top \mathbf{h}^L$
- … or else $\hat{\mathbf{y}} = g(V^\top \mathbf{h}^L)$
- E.g. g: softmax, maxout

The Backpropagation Algorithm

- We have seen how to use Gradient Descent to train perceptron
- Backprop is used to perform GD on multilayer networks
- Basically chain-rule + intelligent bookkeeping
- Recall, we used bookkeeping to do coordinate descent in $\mathcal{O}(n)$ time
- Backprop is still the defacto training technique for most NN
- Indications that this may (have to) change [Hinton 2017]

The Rules that Chain

- Let $x = f(y), y = g(z), z = h(w)$ where $x, y, z, w \in \mathbb{R}$ and f, g, h: $\mathbb{R} \rightarrow \mathbb{R}$
- Chain rule of calculus

$$\frac{dx}{dw} = \frac{dx}{dy} \cdot \frac{dy}{dz} \cdot \frac{dz}{dw} = f'(y) \cdot g'(z) \cdot h'(w)$$

- What about multivariate versions?
- Let $x = f(\mathbf{y})$ where $x \in \mathbb{R}, \mathbf{y} \in \mathbb{R}^q$ and f: $\mathbb{R}^q \rightarrow \mathbb{R}$ - notion of gradient

$$\frac{dx}{d\mathbf{y}} = \nabla f(\mathbf{y}) \in \mathbb{R}^{1 \times q}$$

- If $\mathbf{x} = f(\mathbf{y})$ where $\mathbf{x} \in \mathbb{R}^p, \mathbf{y} \in \mathbb{R}^q$ and f: $\mathbb{R}^q \rightarrow \mathbb{R}^p$ - notion of Jacobian
- Think of $f(\mathbf{y}) = \left[f_1(\mathbf{y}), \ldots, f_p(\mathbf{y})\right]^\top$ as a vector of functions

$$\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}\mathbf{y}} = J^f \text{ where }, J^f_{ij} = \frac{d\mathbf{x}_i}{d\mathbf{y}_j} \text{ i. e. } J^f = \begin{bmatrix} \nabla f_1(\mathbf{y}) \\ \vdots \\ \nabla f_p(\mathbf{y}) \end{bmatrix} \in \mathbb{R}^{p \times q}$$

The Rules that Chain

- Let $x = f(y), y = g(z), z = h(w)$ where $x, y, z, w \in \mathbb{R}$ and f, g, h: $\mathbb{R} \to \mathbb{R}$
- Chain rule of calculus

$$\frac{dx}{dw} = \frac{dx}{dy} \cdot \frac{dy}{dz} \cdot \frac{dz}{dw} = f'(y) \cdot g'(z) \cdot h'($$

Notice slightly different notation where $\nabla f(\mathbf{y})$ is a row vector

- What about multivariate versions?
- Let $x = f(\mathbf{y})$ where $x \in \mathbb{R}, \mathbf{y} \in \mathbb{R}^q$ and f: $\mathbb{R}^q \to \mathbb{R}$ - notion of gradient

$$\frac{dx}{d\mathbf{y}} = \nabla f(\mathbf{y}) \in \mathbb{R}^{1 \times q}$$

- If $\mathbf{x} = f(\mathbf{y})$ where $\mathbf{x} \in \mathbb{R}^p, \mathbf{y} \in \mathbb{R}^q$ and f: $\mathbb{R}^q \to \mathbb{R}^p$ - notion of Jacobian
- Think of $f(\mathbf{y}) = \left[f_1(\mathbf{y}), \ldots, f_p(\mathbf{y})\right]^\top$ as a vector of functions

$$\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}\mathbf{y}} = J^f \text{ where }, J^f_{ij} = \frac{d\mathbf{x}_i}{d\mathbf{y}_j} \text{ i. e. } J^f = \begin{bmatrix} \nabla f_1(\mathbf{y}) \\ \vdots \\ \nabla f_p(\mathbf{y}) \end{bmatrix} \in \mathbb{R}^{p \times q}$$

The Rules that Chain

- Let $x = f(\mathbf{y}), \mathbf{y} = g(\mathbf{z})$ where $x \in \mathbb{R}, \mathbf{y} \in \mathbb{R}^q, \mathbf{z} \in \mathbb{R}^r$ and $f\colon \mathbb{R}^q \rightarrow \mathbb{R}, g\colon \mathbb{R}^r \rightarrow \mathbb{R}^q$
- The chain rule becomes a bit more involved here (proof nontrivial)

$$\frac{dx}{d\mathbf{z}_j} = \sum_{i=1}^{q} \frac{dx}{d\mathbf{y}_i} \cdot \frac{d\mathbf{y}_i}{d\mathbf{z}_j}$$

- Thus we have

$$\frac{dx}{d\mathbf{z}} = \nabla f(\mathbf{y}) \cdot J^g \in \mathbb{R}^{1 \times r}$$

Remember our slightly different notation where $\nabla f(\mathbf{y})$ is a row vector

- Similarly, if we have $\mathbf{x} = f(\mathbf{y}), \mathbf{y} = g(\mathbf{z})$ where $x \in \mathbb{R}^p, \mathbf{y} \in \mathbb{R}^q, \mathbf{z} \in \mathbb{R}^r$

$$\frac{d\mathbf{x}}{d\mathbf{z}} = J^f \cdot J^g$$

The Rules that Chain

- This actually extends very cleanly to more complex instances
- If $\mathbf{x} = f(\mathbf{y}), \mathbf{y} = g(\mathbf{z}), \mathbf{z} = h(\mathbf{w})$ where $x \in \mathbb{R}^p, \mathbf{y} \in \mathbb{R}^q, \mathbf{z} \in \mathbb{R}^r, \mathbf{w} \in \mathbb{R}^s$

$$\frac{d\mathbf{x}}{d\mathbf{w}} = J^f \cdot J^g \cdot J^h$$

- It also unifies the previous cases beautifully
- Consider a function $\mathbf{x} = f(\mathbf{y})$ where $x \in \mathbb{R}^p, \mathbf{y} \in \mathbb{R}^q$
  - If $p = q = 1$ (basic case), Jacobian is simply a 1×1 matrix (a real number)
  - If $p = 1$ and $q > 1$ (real valued function on vector space), Jacobian becomes a row vector
  - If $p > 1$ and $q = 1$ and no $\mathbf{w}$ (vector valued function on the real line), Jacobian becomes a column vector
- However, $\frac{d\mathbf{x}}{d\mathbf{w}} = J^f \cdot J^g \cdot J^h$ always applies if careful with dimensions

The Rules that Chain

- This actually extends very cleanly to more co
- If $\mathbf{x} = f(\mathbf{y}), \mathbf{y} = g(\mathbf{z}), \mathbf{z} = h(\mathbf{w})$ where $x \in \mathbb{R}$, $\mathbf{y} \in$

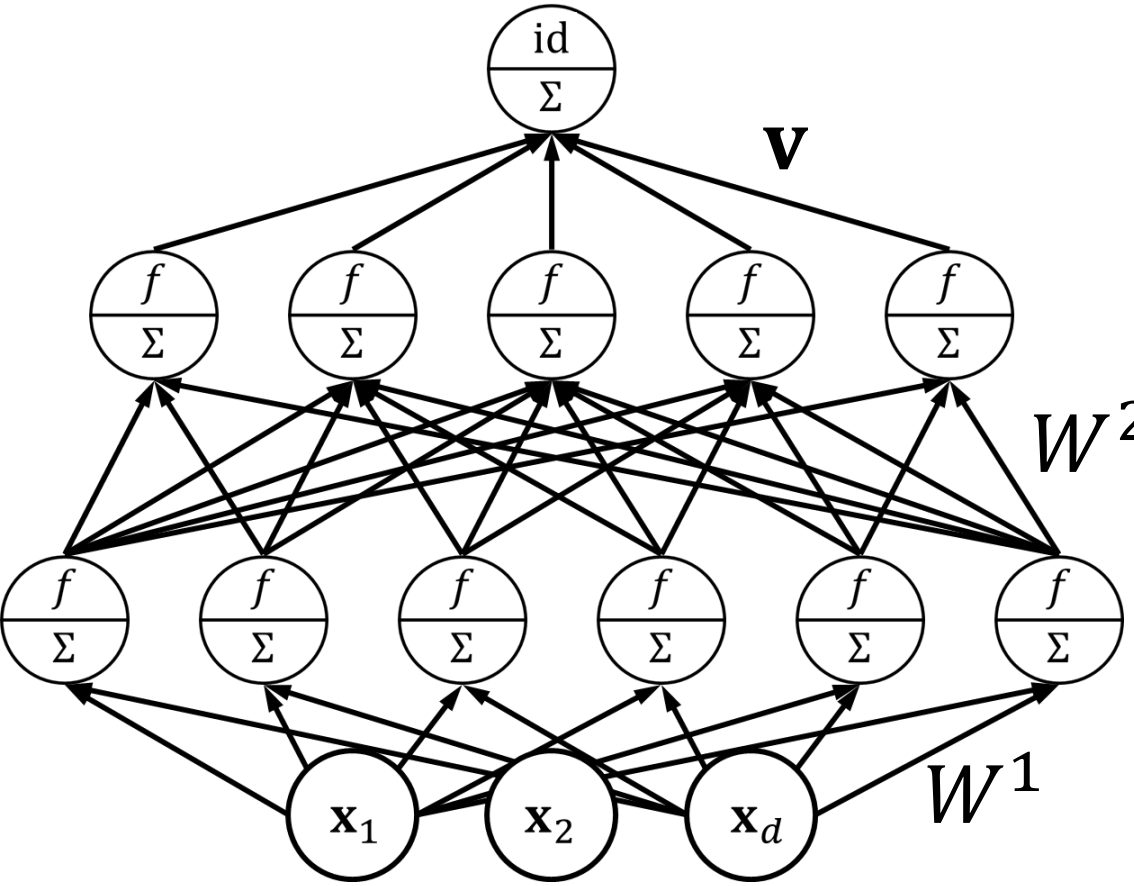
$$\frac{d\mathbf{x}}{d\mathbf{w}} = J^f \cdot J^g \cdot J^h$$

Mimics earlier result

$$\frac{dx}{dw} = f'(y) \cdot g'(z) \cdot h'(w)$$

- It also unifies the previous cases beautifully
- Consider a function $\mathbf{x} = f(\mathbf{y})$ where $x \in \mathbb{R}^p, \mathbf{y} \in \mathbb{R}^q$
  - If $p = q = 1$ (basic case), Jacobian is simply a 1×1 matrix (a real number)
  - If $p = 1$ and $q > 1$ (real valued function on vector space), Jacobian becomes a row vector
  - If $p > 1$ and $q = 1$ and no $\mathbf{w}$ (vector valued function on the real line), Jacobian becomes a column vector
- However, $\frac{d\mathbf{x}}{d\mathbf{w}} = J^f \cdot J^g \cdot J^h$ always applies if careful with dimensions

Back to Backpropagation



- $\hat{y} = \langle \mathbf{v}, \mathbf{h}^2 \rangle$ is the output of the NN
- We wish to train using GD on

$$F(\mathbf{v}, W^2, W^1) = \sum_{i=1}^{n} \ell(\hat{y}^i, y^i) = \sum_{i=1}^{n} \ell^i$$

- Need only describe how to calculate

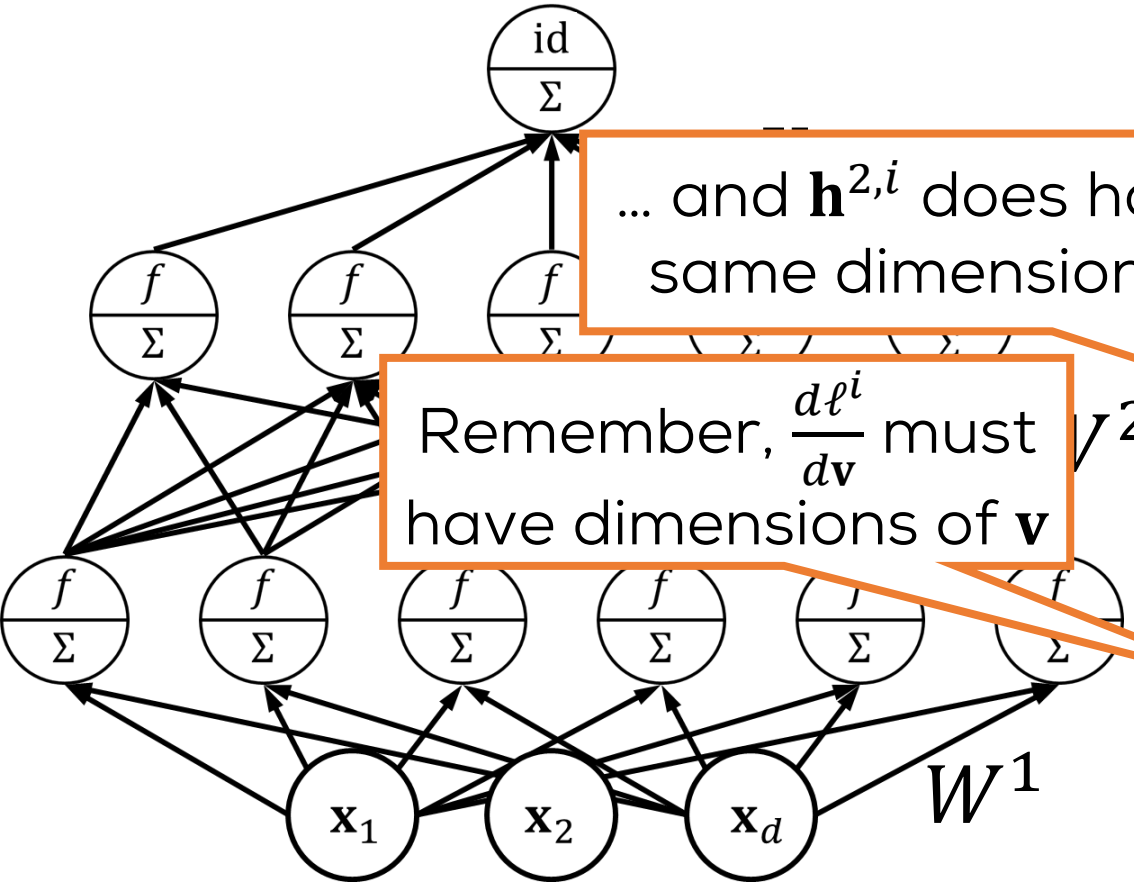
$$\frac{d\ell^i}{d\mathbf{v}}, \frac{d\ell^i}{dW^2}, \frac{d\ell^i}{dW^1}$$

- First is easy

$$\frac{d\ell^i}{d\mathbf{v}} = \frac{d\ell^i}{d\hat{y}^i} \cdot \frac{d\hat{y}^i}{d\mathbf{v}} = \ell'(\hat{y}^i, y^i) \cdot \mathbf{h}^{2,i}$$

- $\mathbf{h}^{2,i}$ is the o/p of the 2nd hidden layer on the i-th data point

Back to Backpropagation



… and $\mathbf{h}^{2,i}$ does have the same dimensions as $\mathbf{v}$

Remember, $\frac{d\ell^i}{d\mathbf{v}}$ must have dimensions of $\mathbf{v}$

- $\hat{y} = \langle \mathbf{v}, \mathbf{h}^2 \rangle$ is the output of the NN
- We wish to train using GD on

$$W^2, W^1) = \sum_{i=1}^{n} \ell(\hat{y}^i, y^i) = \sum_{i=1}^{n} \ell^i$$

- … only describe how to calculate

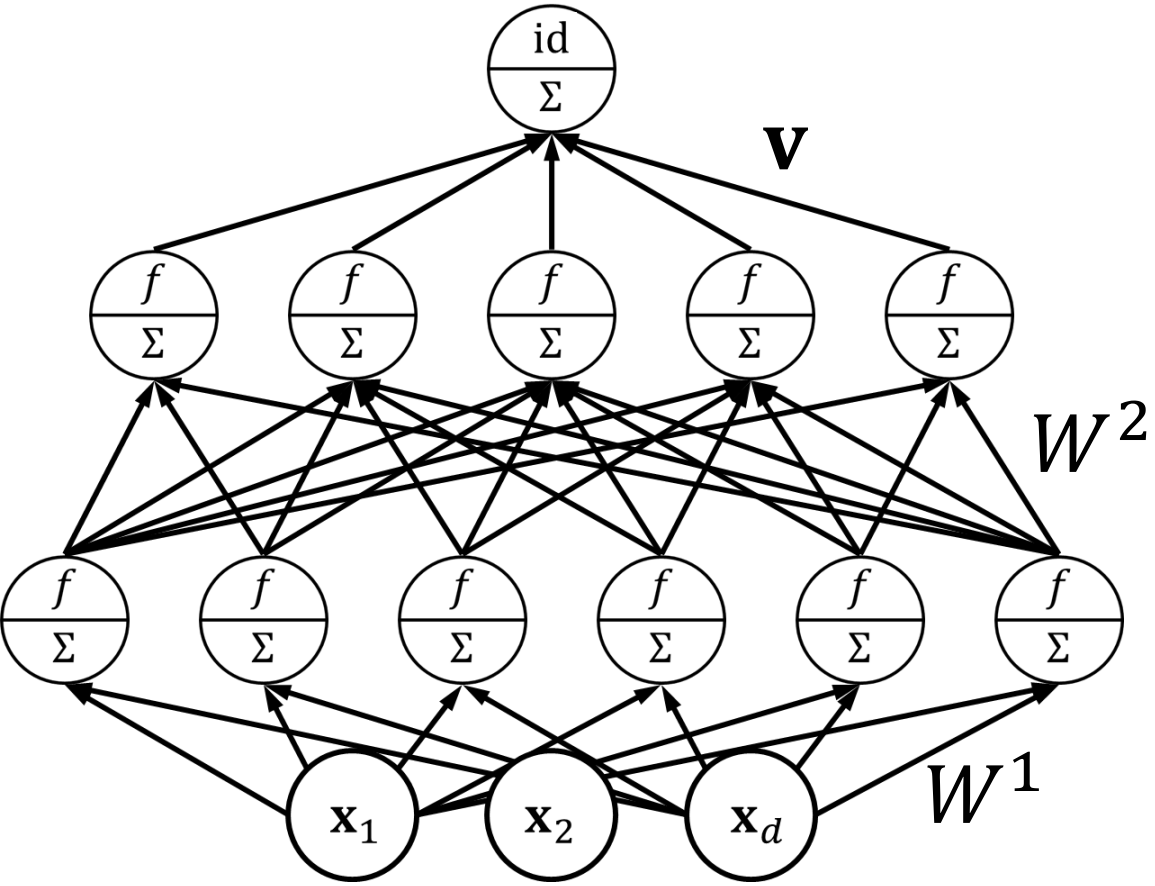
$$\frac{d\ell^i}{d\mathbf{v}}, \frac{d\ell^i}{dW^2}, \frac{d\ell^i}{dW^1}$$

First is easy

$$\frac{d\ell^i}{d\mathbf{v}} = \frac{d\ell^i}{d\hat{y}^i} \cdot \frac{d\hat{y}^i}{d\mathbf{v}} = \ell'(\hat{y}^i, y^i) \cdot \mathbf{h}^{2,i}$$

- $\mathbf{h}^{2,i}$ is the o/p of the 2nd hidden layer on the i-th data point

Back to Backpropagation



- Second is more challenging

$$\frac{d\ell^i}{dW^2} = \frac{d\ell^i}{d\hat{y}^i} \cdot \frac{d\hat{y}^i}{dW^2} = \ell'(\hat{y}^i, y^i) \cdot \frac{d\hat{y}^i}{dW^2}$$

$$= \ell'(\hat{y}^i, y^i) \cdot \frac{d\hat{y}^i}{d\mathbf{h}^{2,i}} \cdot \frac{d\mathbf{h}^{2,i}}{dW^2} = \ell'(\hat{y}^i, y^i) \cdot \mathbf{v} \cdot \frac{d\mathbf{h}^{2,i}}{dW^2}$$

- Use the fact that $\mathbf{h}^{2,i} = f\left((W^2)^\top \mathbf{h}^{1,i}\right)$
- Why did we not consider $\frac{d\hat{y}^i}{d\mathbf{v}}$ above?

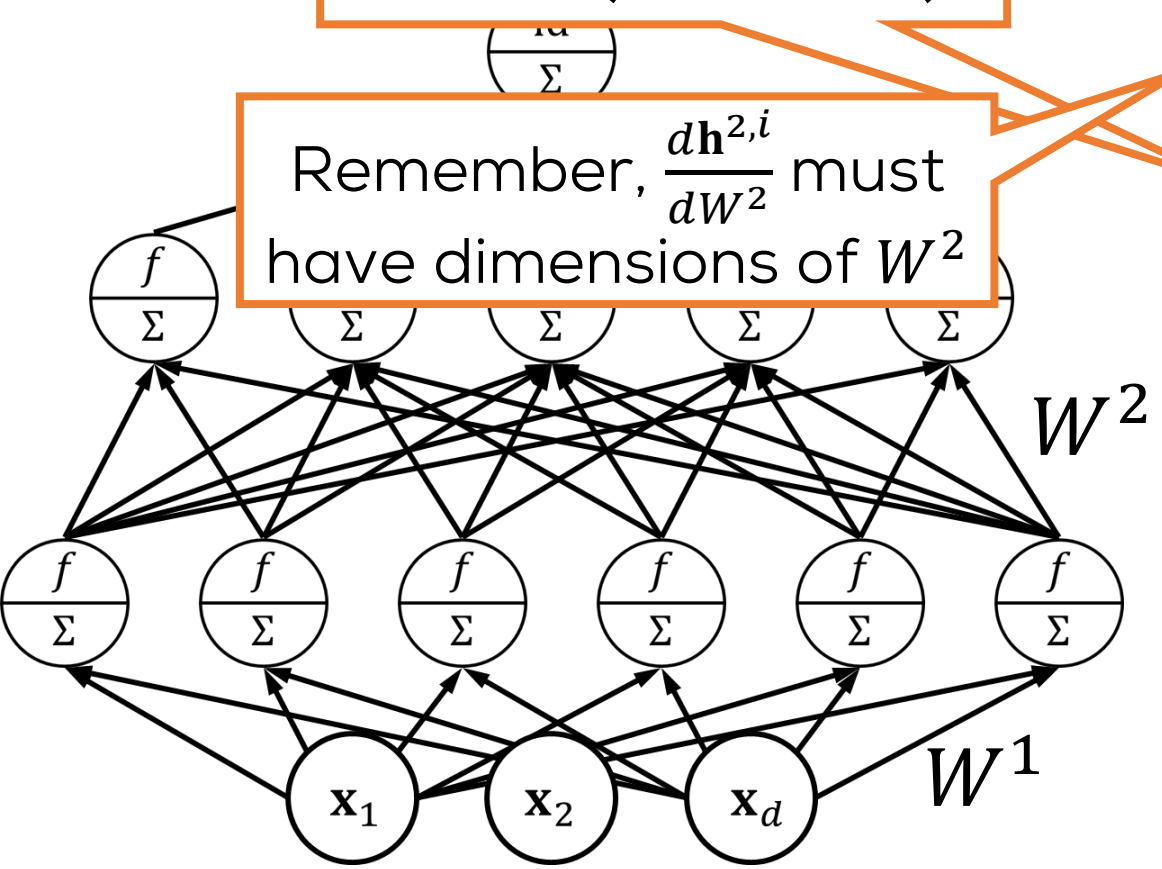
$$\frac{d\hat{y}^i}{dW^2} = \frac{d\hat{y}^i}{d\mathbf{h}^{2,i}} \cdot \frac{d\mathbf{h}^{2,i}}{dW^2} + \frac{d\hat{y}^i}{d\mathbf{v}} \cdot \frac{d\mathbf{v}}{dW^2}$$

- We could have but $\frac{d\mathbf{v}}{dW^2} = \mathbf{0}$ since $\mathbf{v}$ is not a function of W^2

Back to Backpropagation

Careful, $\frac{d\mathbf{h}^{2,i}}{dW^2}$ is a 3-D matrix (3D tensor)

Remember, $\frac{d\mathbf{h}^{2,i}}{dW^2}$ must have dimensions of W^2



- Second is more challenging

$$\frac{d\ell^i}{dW^2} = \frac{d\ell^i}{d\hat{y}^i} \cdot \frac{d\hat{y}^i}{dW^2} = \ell'(\hat{y}^i, y^i) \cdot \frac{d\hat{y}^i}{dW^2}$$

$$= \ell'(\hat{y}^i, y^i) \cdot \frac{d\hat{y}^i}{d\mathbf{h}^{2,i}} \cdot \frac{d\mathbf{h}^{2,i}}{dW^2} = \ell'(\hat{y}^i, y^i) \cdot \mathbf{v} \cdot \frac{d\mathbf{h}^{2,i}}{dW^2}$$

- Use the fact that $\mathbf{h}^{2,i} = f\left((W^2)^\top \mathbf{h}^{1,i}\right)$
- Why did we not consider $\frac{d\hat{y}^i}{d\mathbf{v}}$ above?

$$\frac{d\hat{y}^i}{dW^2} = \frac{d\hat{y}^i}{d\mathbf{h}^{2,i}} \cdot \frac{d\mathbf{h}^{2,i}}{dW^2} + \frac{d\hat{y}^i}{d\mathbf{v}} \cdot \frac{d\mathbf{v}}{dW^2}$$

- We could have but $\frac{d\mathbf{v}}{dW^2} = \mathbf{0}$ since $\mathbf{v}$ is not a function of W^2

Bookkeeping in Backprop

- Lower derivatives get more and more costly

$$\frac{d\ell^i}{dW^1} = \frac{d\ell^i}{d\hat{y}^i} \cdot \frac{d\hat{y}^i}{d\mathbf{h}^{2,i}} \cdot \frac{d\mathbf{h}^{2,i}}{d\mathbf{h}^{1,i}} \cdot \frac{d\mathbf{h}^{1,i}}{dW^1}$$

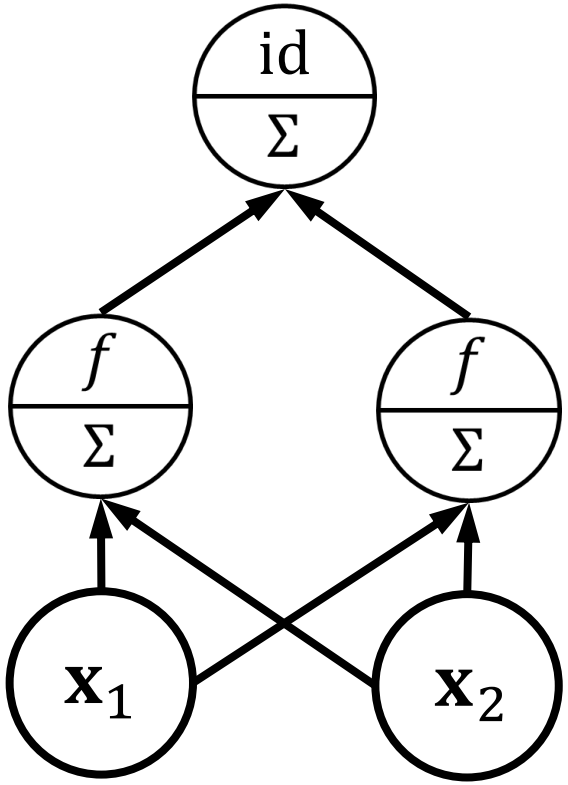
- We have ignored the terms $\frac{\mathrm{d}\ell^i}{dy^i}, \frac{d\hat{y}^i}{d\mathbf{v}}, \frac{d\mathbf{h}^{2,i}}{dW^2}, \frac{d\mathbf{h}^{1,i}}{d\mathbf{x}^i}$ ($\mathbf{x}^i$ is the input layer) since $\frac{dy^i}{dW^1} = \frac{d\mathbf{v}}{dW^1} = \frac{dW^2}{dW^1} = \frac{d\mathbf{x}^i}{dW^1} = \mathbf{0}$ ($\mathbf{x}^i, y^i$ are constants)
- Note that quantities like $\frac{d\ell^i}{d\hat{y}^i}, \frac{d\hat{y}^i}{d\mathbf{h}^{2,i}}$ get used again and again
- Better to compute them once and store them – bookkeeping
- Basically what frameworks like Theano, TensorFlow, Torch do
- Which is why they take up so much memory as well!

Dropout

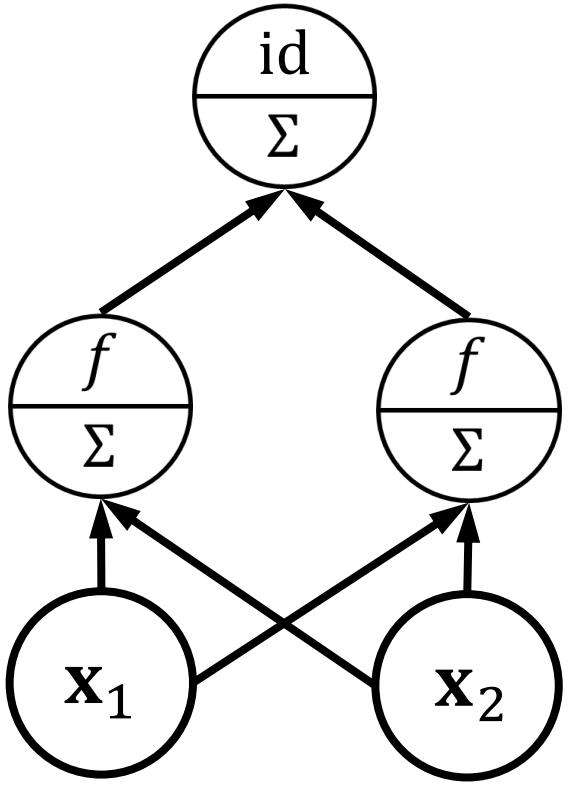
Dropout

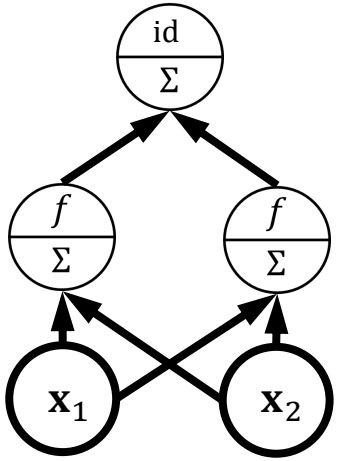
- An effort to make networks more resilient to noise, node failure
- Side effect is slightly faster training and regularization
- Basic idea is to train sparse networks (lots of edges missing)
- The lead author of the Dropout paper is an IITK alumni ☺
- During training, before applying mini-batch gradient descent
  - Randomly sample a subset of input nodes (choose each with prob 20%)
  - Randomly sample a subset of hidden nodes (choose each w/p 50%)
  - Remove those nodes, and corresponding edges from the network
  - Apply mini-batch gradient descent-backprop to the remaining network
  - Keep using NAG, AdaGrad etc as usual
- Forces nodes to learn to work in absence of other nodes – robust!

Dropout at Work

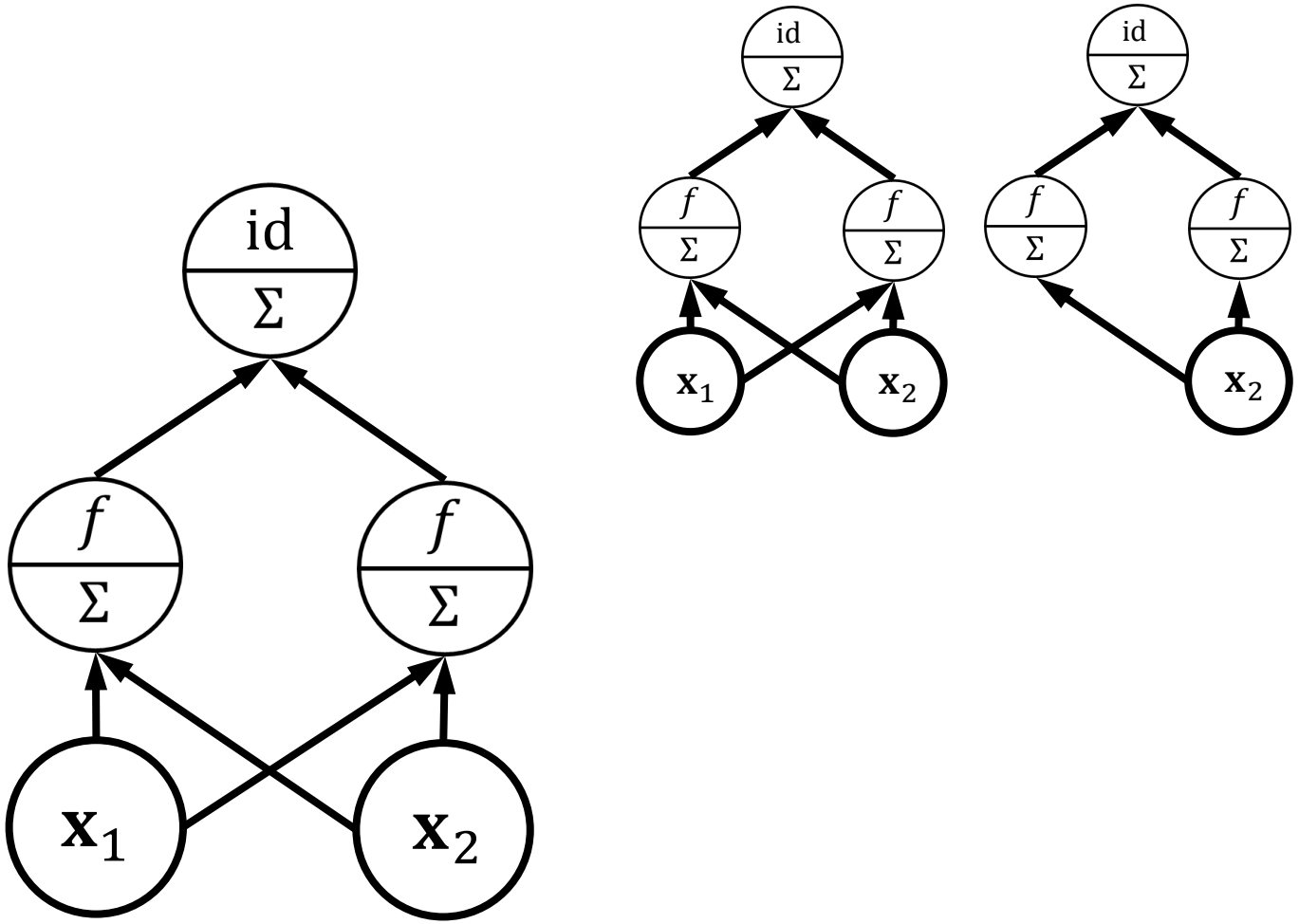


Dropout at Work

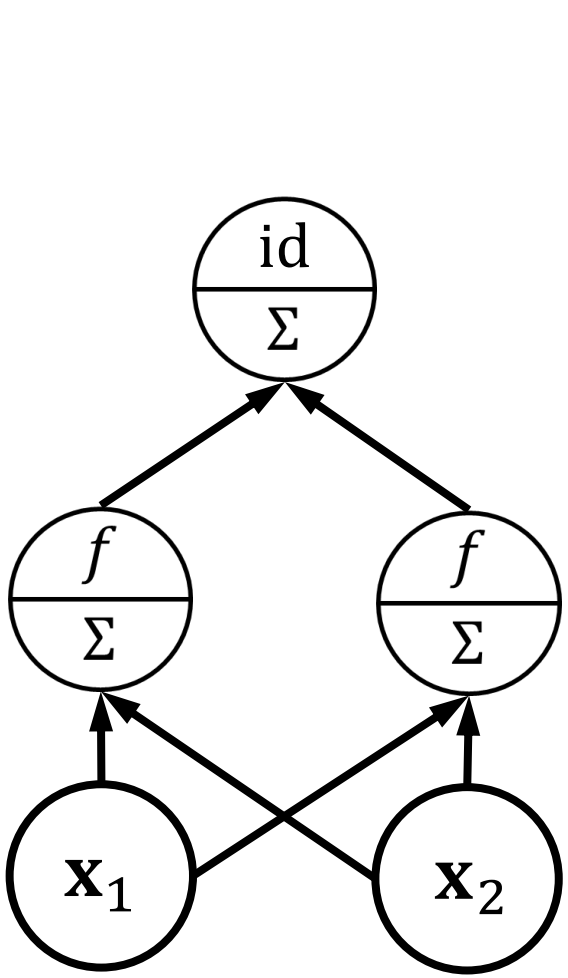


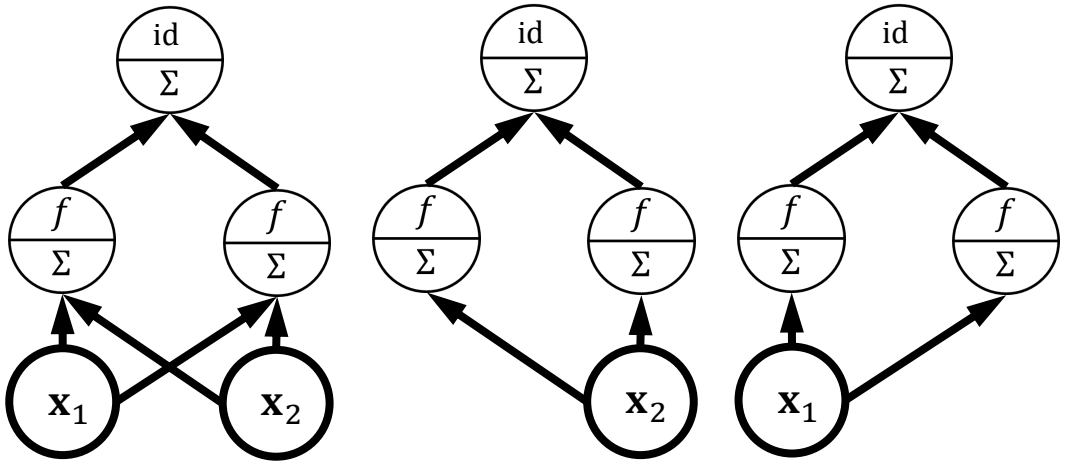


Dropout at Work

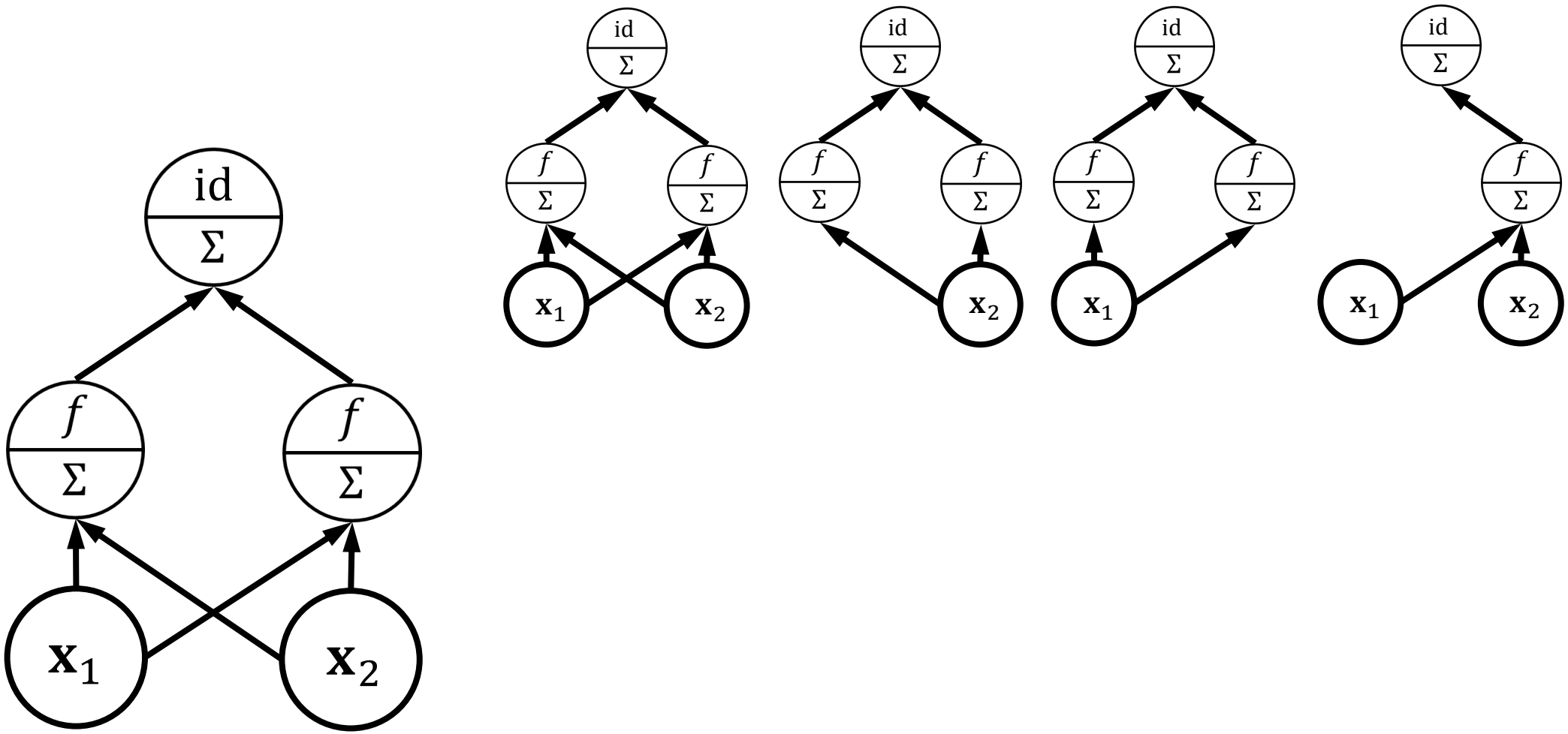


Dropout at Work

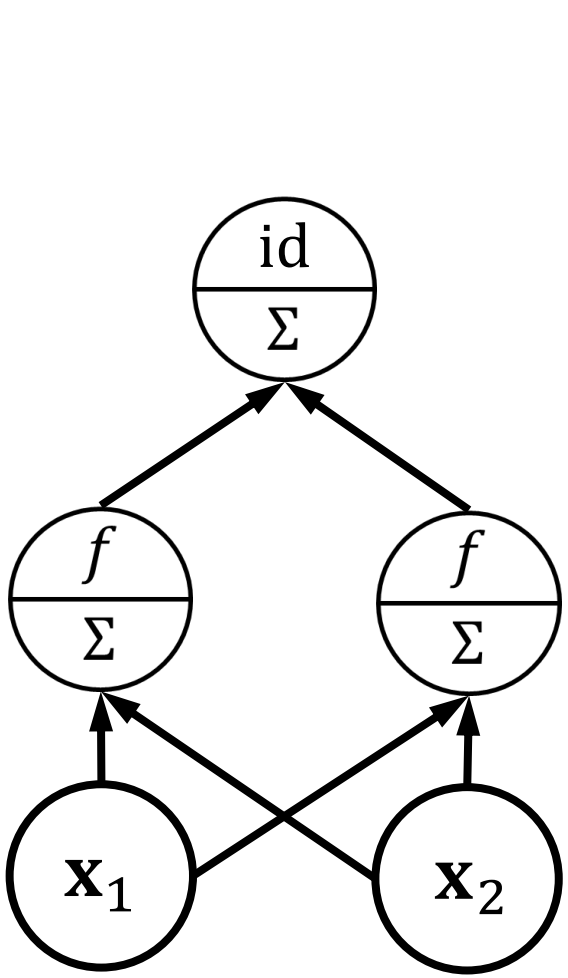


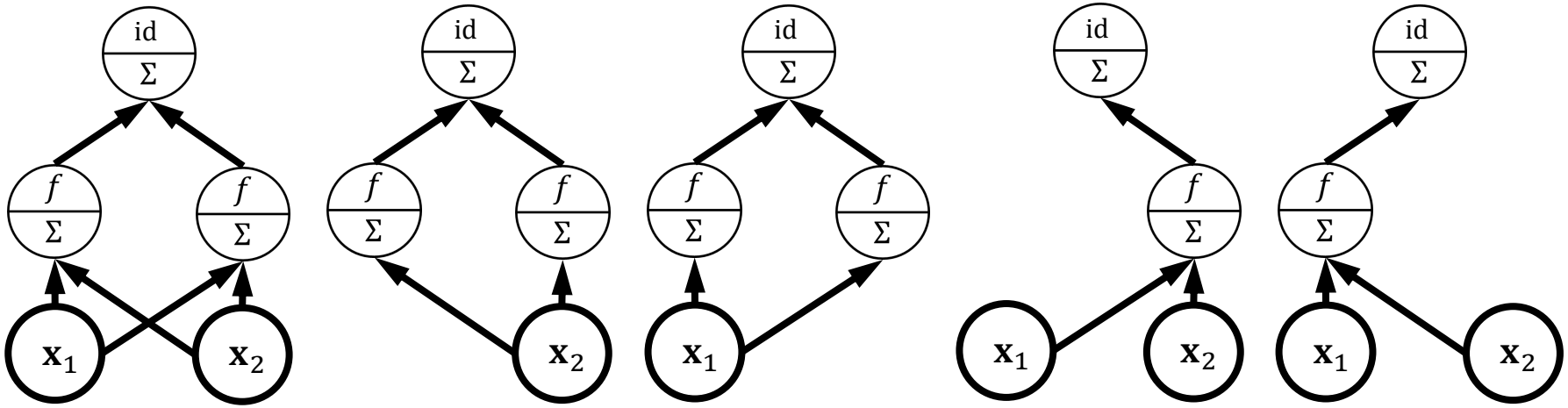


Dropout at Work

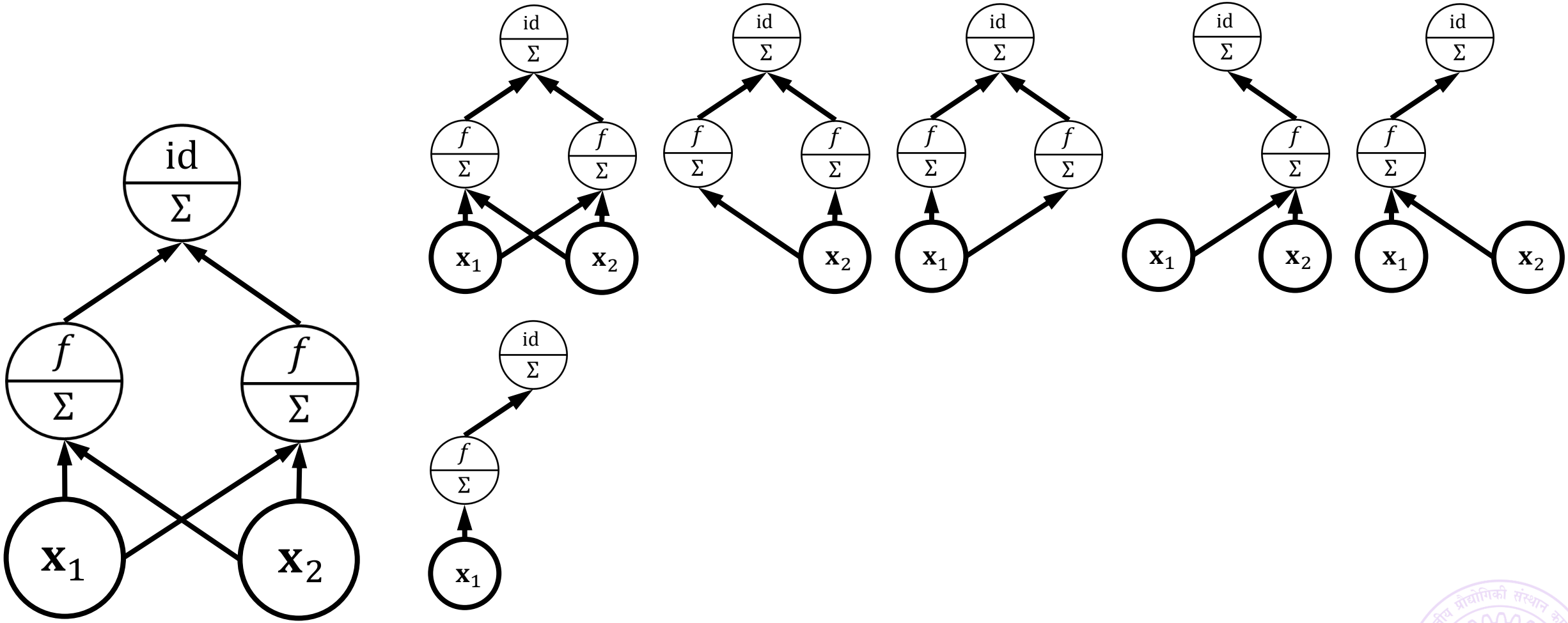


Dropout at Work

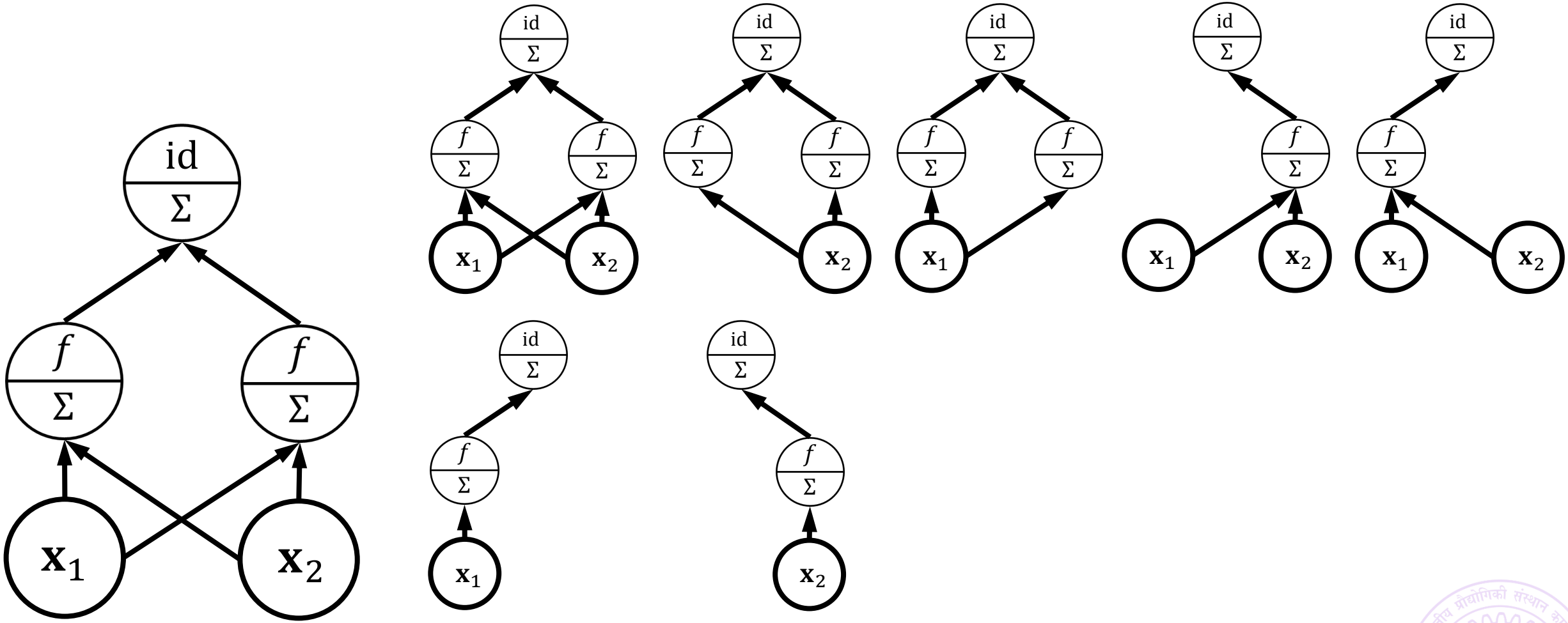




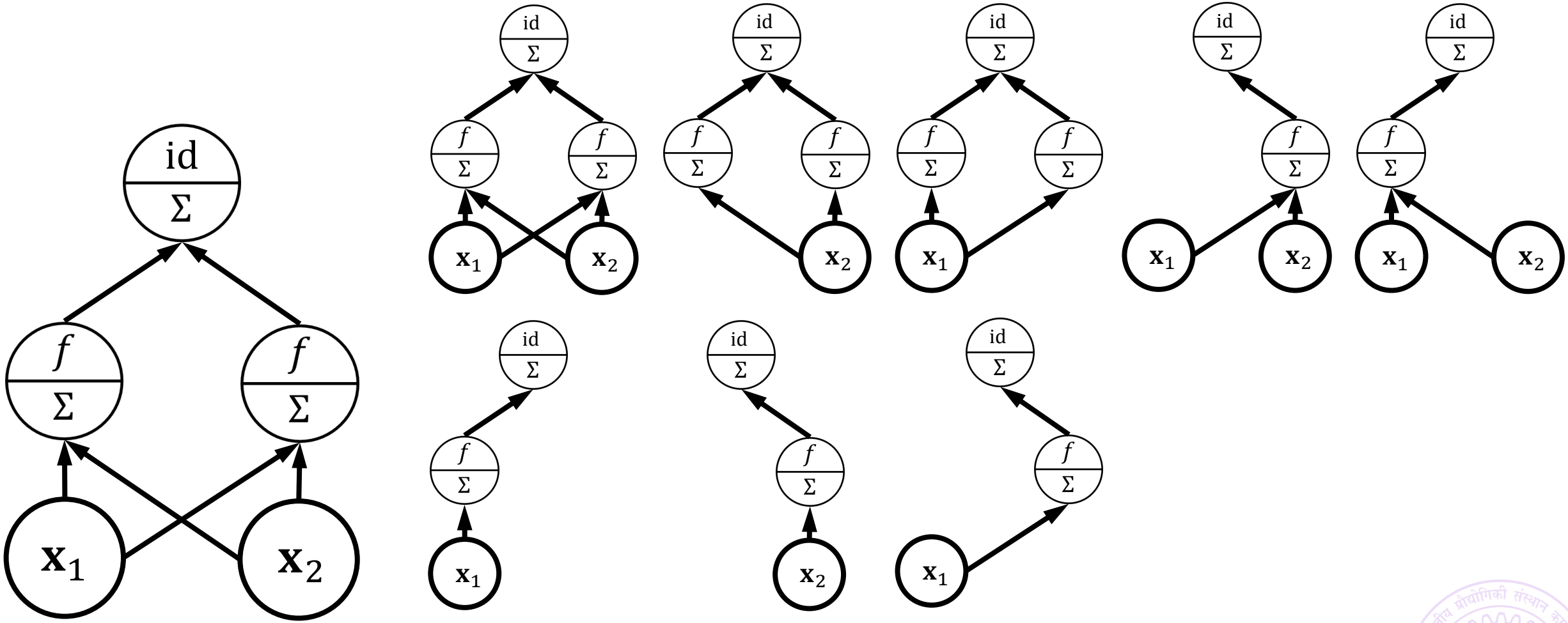
Dropout at Work



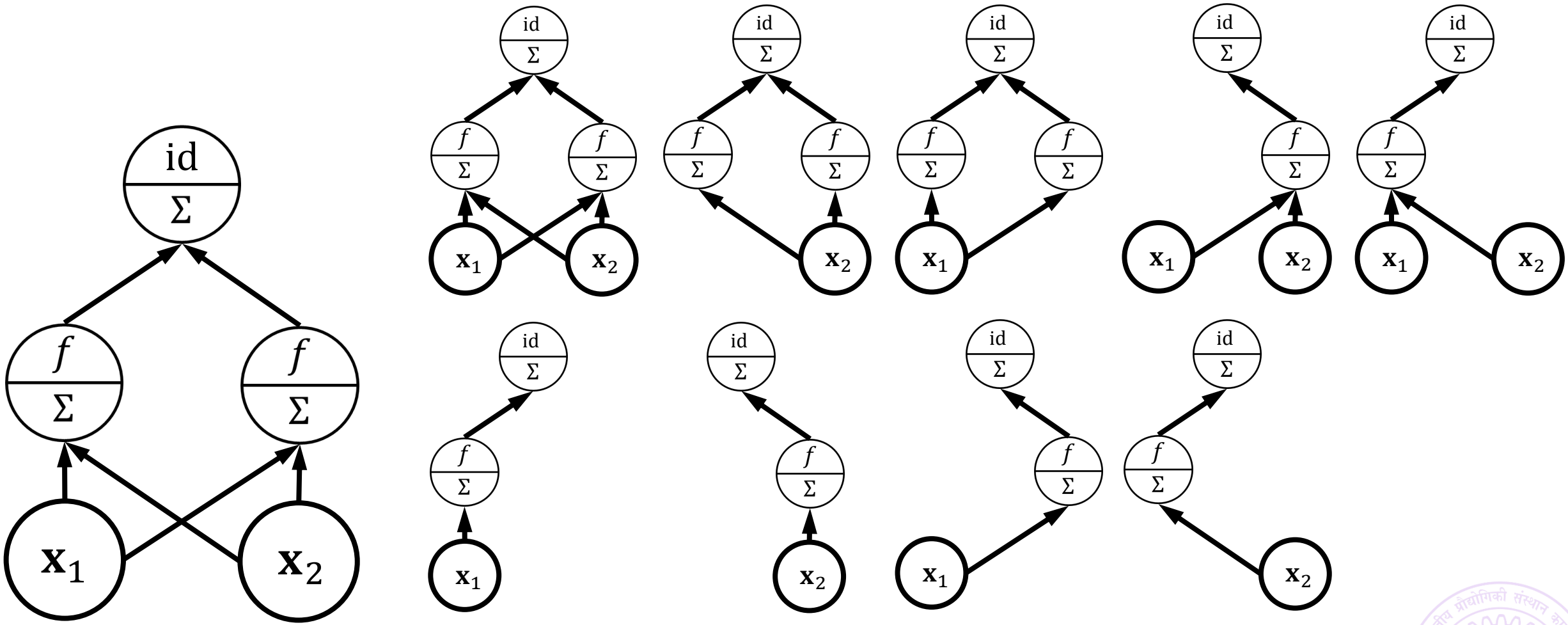
Dropout at Work



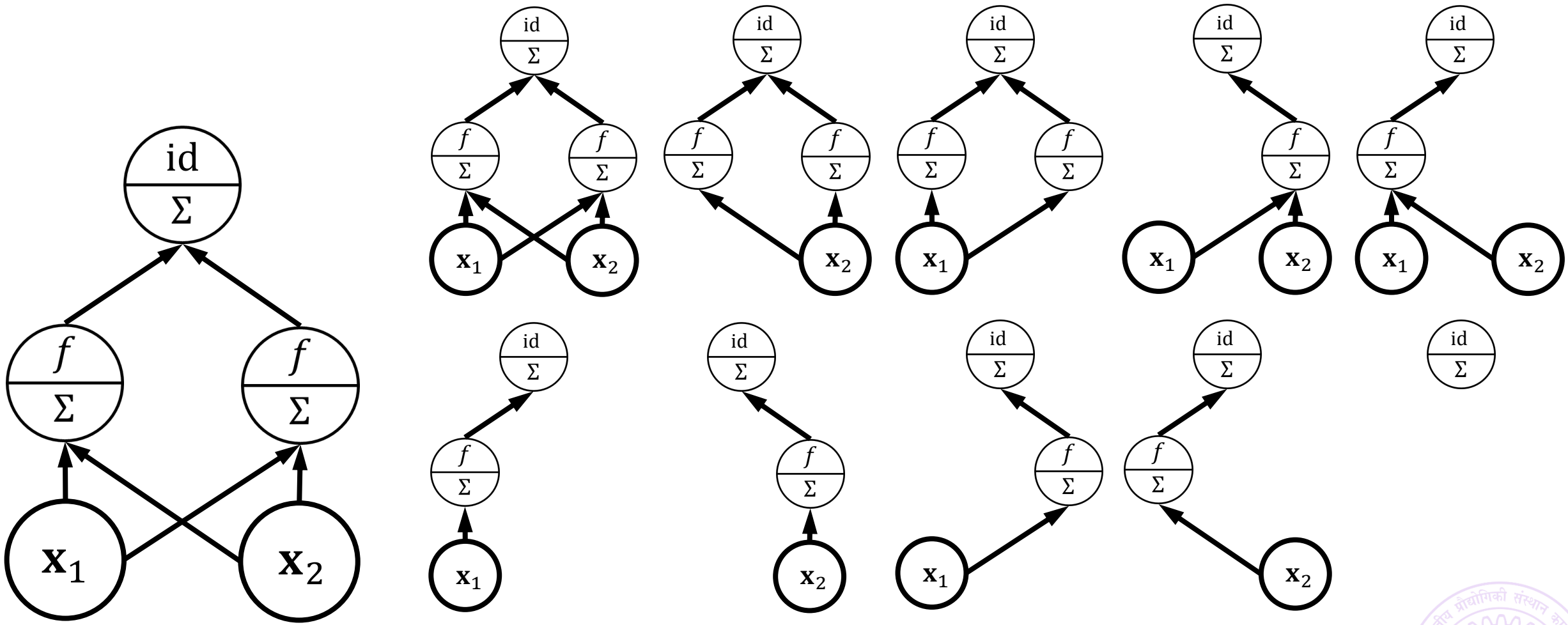
Dropout at Work



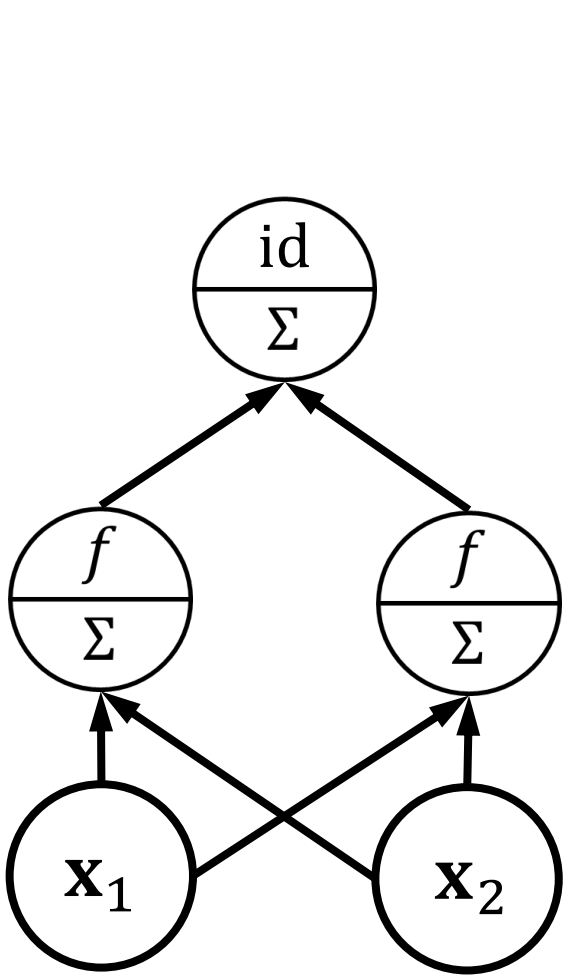
Dropout at Work



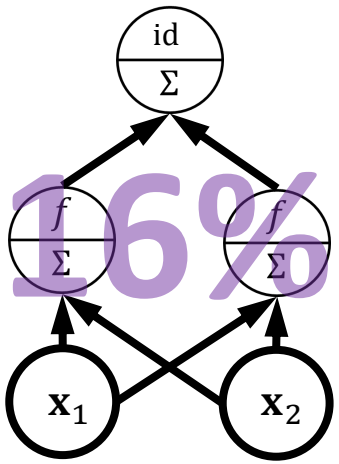
Dropout at Work



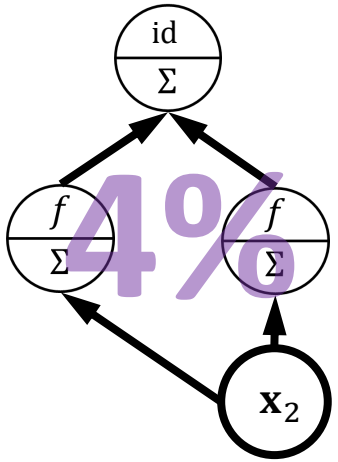
Dropout at Work



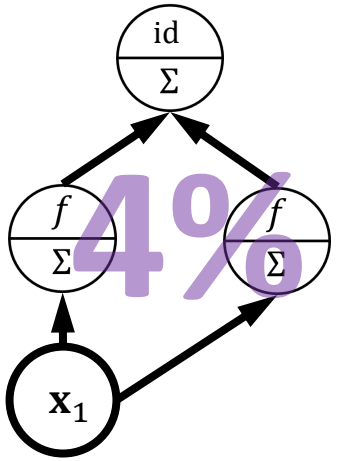
16%



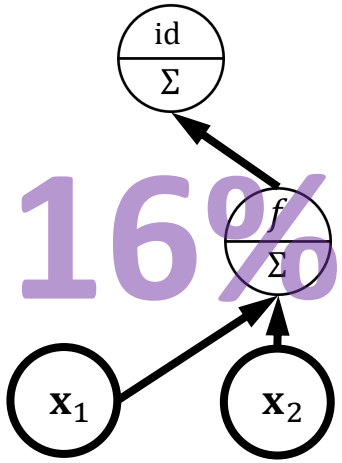
4%



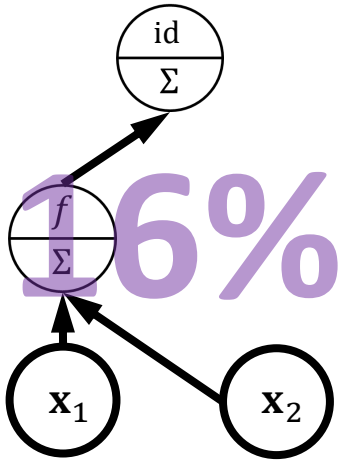
4%



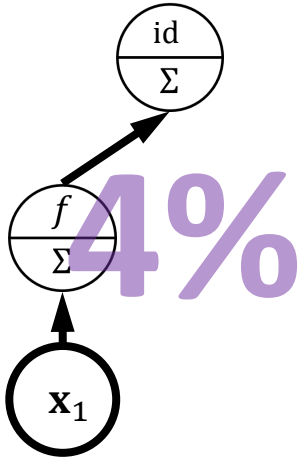
16%



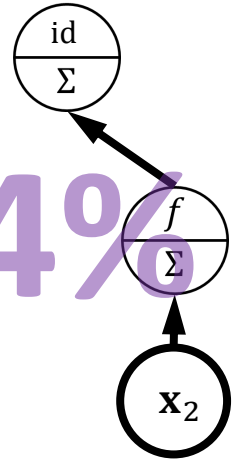
16%



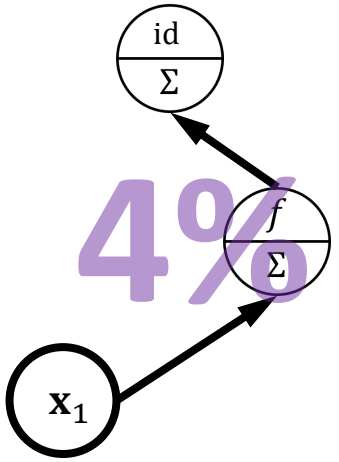
4%



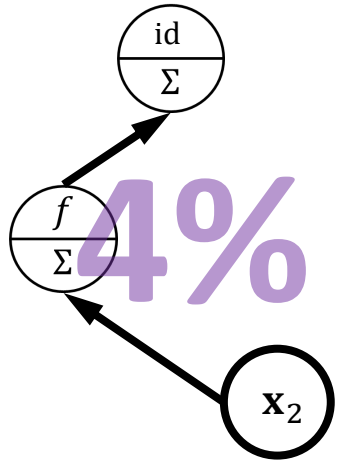
4%



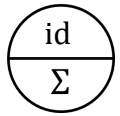
4%



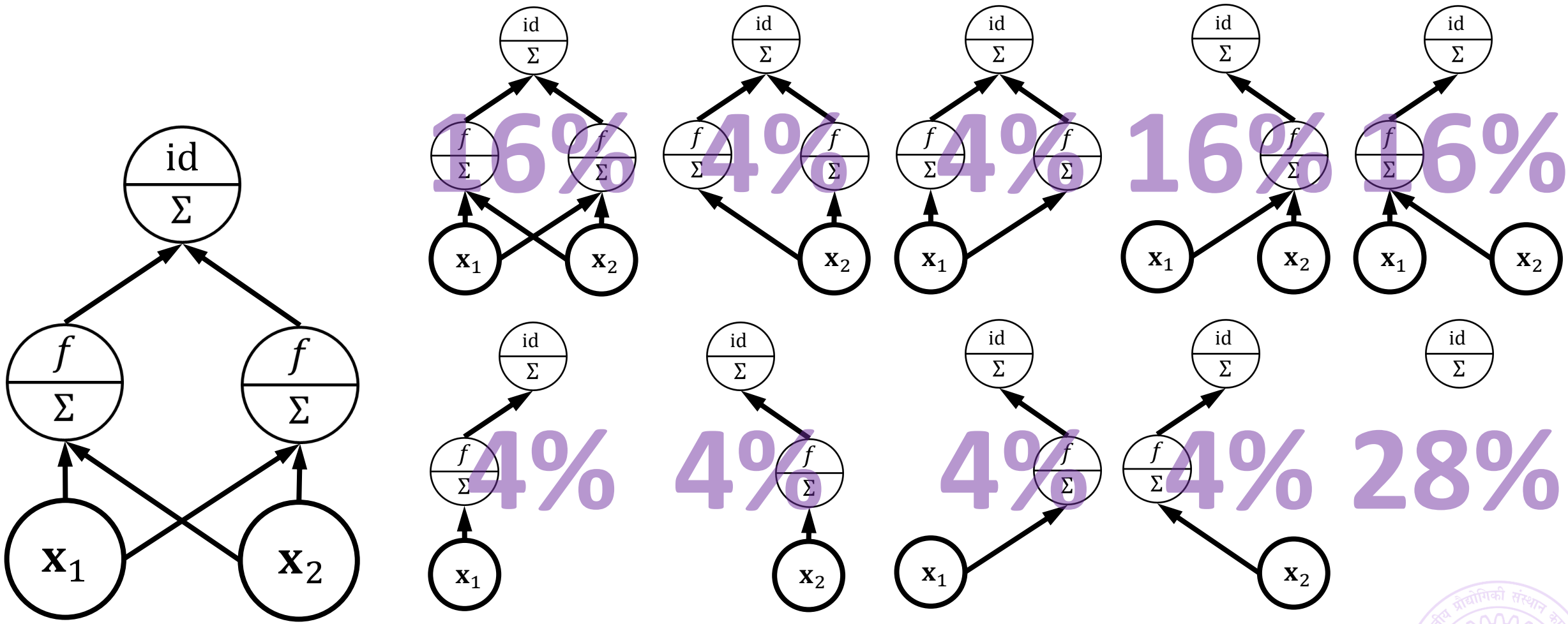
4%



28%



Dropout at Work



28% networks have no input nodes or no path connecting at least one input node to the output node (cannot apply GD to them)

Dropout at Test time

- Can do dropout at test time too
- What is desirable is to achieve the average effect of all the sparse subnetworks in the previous slide
- The above becomes intractable (kind of like Bayesian inference)
- Approximations used
  - Sample several networks using same sampling techniques, predict using each and take the average prediction
  - Simply scale the output of each node in the NN with the probability it would have been missing
  - See Deep Learning book [**GBC**] for a more detailed discussion

Data Modelling with NNs

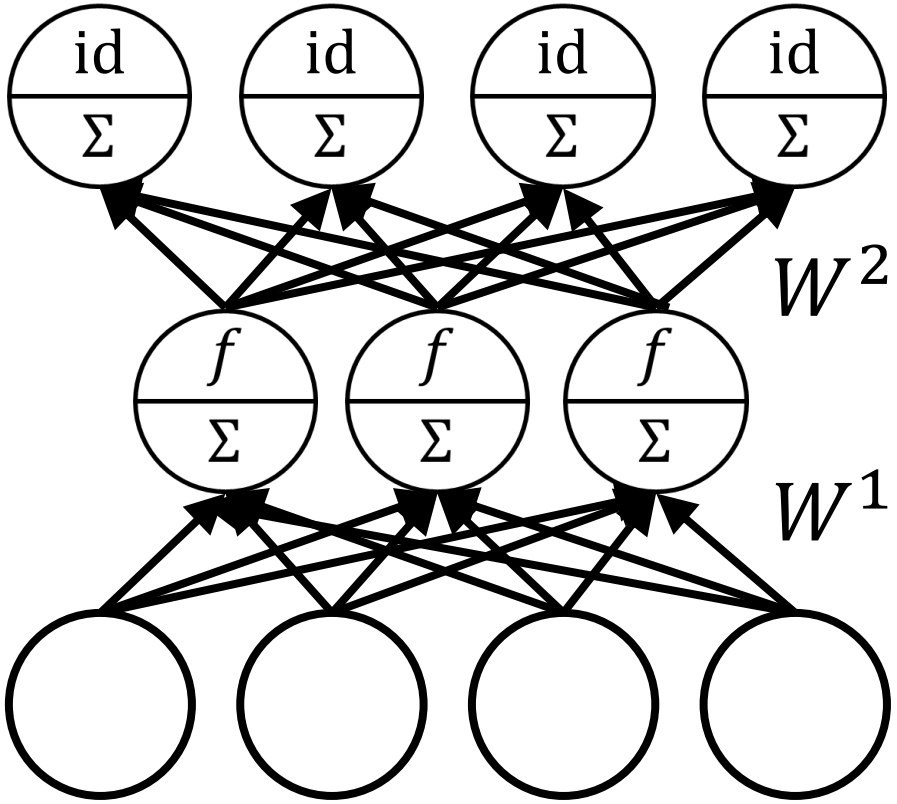
Networks that model/generate data

- Note: feedforward networks are discriminative
- Generative networks exist too!
- Too vast an area to do justice in one-and-a-half lectures
- Several flavours exist
  - Autoencoders (akin to PCA/PPCA)
  - Boltzmann machines (restricted/convolutional/fully-connected)
  - Deep belief networks
- Will just look at autoencoders briefly since they mimic PCA
- [**GBC**] contains a detailed description of other architectures

Autoencoders

- Recall that in PCA, a latent variable $\mathbf{z}$ generated the data as
$$\mathbf{x} = W\mathbf{z} + \boldsymbol{\epsilon}$$
- Also recall that if we could find W, then we had $\mathbf{z} = W^{\top}\mathbf{x}$
- Thus, we cast the problem as that of regenerating the data
$$\|\mathbf{x} - WW^{\top}\mathbf{x}\|_2^2$$
- However, since W is low-rank, we are forced to regenerate the data using a low-rank representation
- Autoencoders do pretty much the same
- An encoder converts data to a hidden representation $\mathbf{h} = e(\mathbf{x})$
- A decoder produces a reconstruction $\mathbf{r} = d(\mathbf{h})$
- Want to minimize loss $\ell(d(\mathbf{h}), \mathbf{x}) = \ell\big(d\big(e(\mathbf{x})\big), \mathbf{x}\big)$

Deep Autoencoders



- Encoder is $e(\mathbf{x}) = f\left((W^1)^\top \mathbf{x}\right)$
- Decoder is $d(\mathbf{h}) = (W^2)^\top \mathbf{h}$
- Reconstruction is $(W^2)^\top f\left((W^1)^\top \mathbf{x}\right)$
- Training done using

$$\ell\left((W^2)^\top f\left((W^1)^\top \mathbf{x}\right), \mathbf{x}\right)$$

- For example

$$\left\|(W^2)^\top f\left((W^1)^\top \mathbf{x}\right) - \mathbf{x}\right\|_2^2$$

- Undercomplete AE: # hidden nodes less than # input nodes
- Prevents encoded and decoder from being trivial indentity function

Autoencoder Variants

- Often some noise is added to input $\mathbf{x}$ to avoid overfitting,

$$\ell\big(d\big(e(\mathbf{x}+\boldsymbol{\epsilon})\big),\mathbf{x}\big)$$

  Denoising autoencoder!
- Note: PCA is an autoencoder too … just a linear one
- Also, it fixes $W^2 = W^1 = W$
- Interestingly, this was done in the past for autoencoders too
- Pretraining phase – train with constraint $W^2 = W^1$
- Fine-tuning phase – train allowing the two to differ
- However, these days it is common to learn W^1, W^2 separately from scratch. No pretraining and fine-tuning phases

Autoencoder Variants

- Can have overcomplete autoencoders as well – need to be careful
- Sparse autoencoders – force lots of edge weights to be zero
- Can have several hidden layers – learn compressed representations in several stages
- Possible to take 780-dimensional MNIST dataset and reduce to 7-dimensional representation without loss of too much information
- SVMs applied to 7-dimensional version do almost as well as SVMs applied to 780-dimensional version [LeCun]
- Such a drastic reduction not possible in practice in one shot
- AE needs several layers to gradually reduce dimensionality e.g.

$$780 \rightarrow 500 \rightarrow 100 \rightarrow 50 \rightarrow 7 \rightarrow 50 \rightarrow 100 \rightarrow 500 \rightarrow 780$$

Please give your Feedback

http://tinyurl.com/ml17-18afb