Deep Learning III

CS771: Introduction to Machine Learning

Purushottam Kar

Announcements

Quiz: November 01 (Friday), 6PM, **L20 – same as before**

Assigned seating – don't be late (will waste time finding your seat)

Syllabus is till whatever we covered till October 30 (today)

*Bring your **institute ID card** with you – will lose time if you forget*

*Bring a **pencil, pen, eraser, sharpener** with you – we wont provide!*

*Answers to be written on question paper itself. If you write with pen and make a mistake, no extra paper. Final answer **must be in pen***

***Auditors cannot appear** for quiz – please come to L20 at ~ 6:40PM*

Doubt clearing session: Oct 31 (Thu), 6PM **KD101**

Announcements

Cross platform issue with Cython usage in Assignment 2

- *Since Python is a scripting language, often C/C++ code needed for speed*
- *Cython allows C/C++ code to be called from Python code*
- *Unfortunately requires "shared" library (.so) files to be generated which are not very cross platform compatible*

Policy on shared library usage in Assignment 2

- *Groups are allowed to use Cython which requires .so files to be generated*
- *All such files must be supplied with the submission at submission deadline*
- *If such files are required but not submitted then zero marks for coding question*
- *If your .so files are incompatible with our system, you will be called for a demo where you would be asked to generate the .so files on our system*
- *Larger of the .so files would be counted to calculate model size*

Recap of Previous Lecture

NN are universal models – all powerful given a "big" enough network

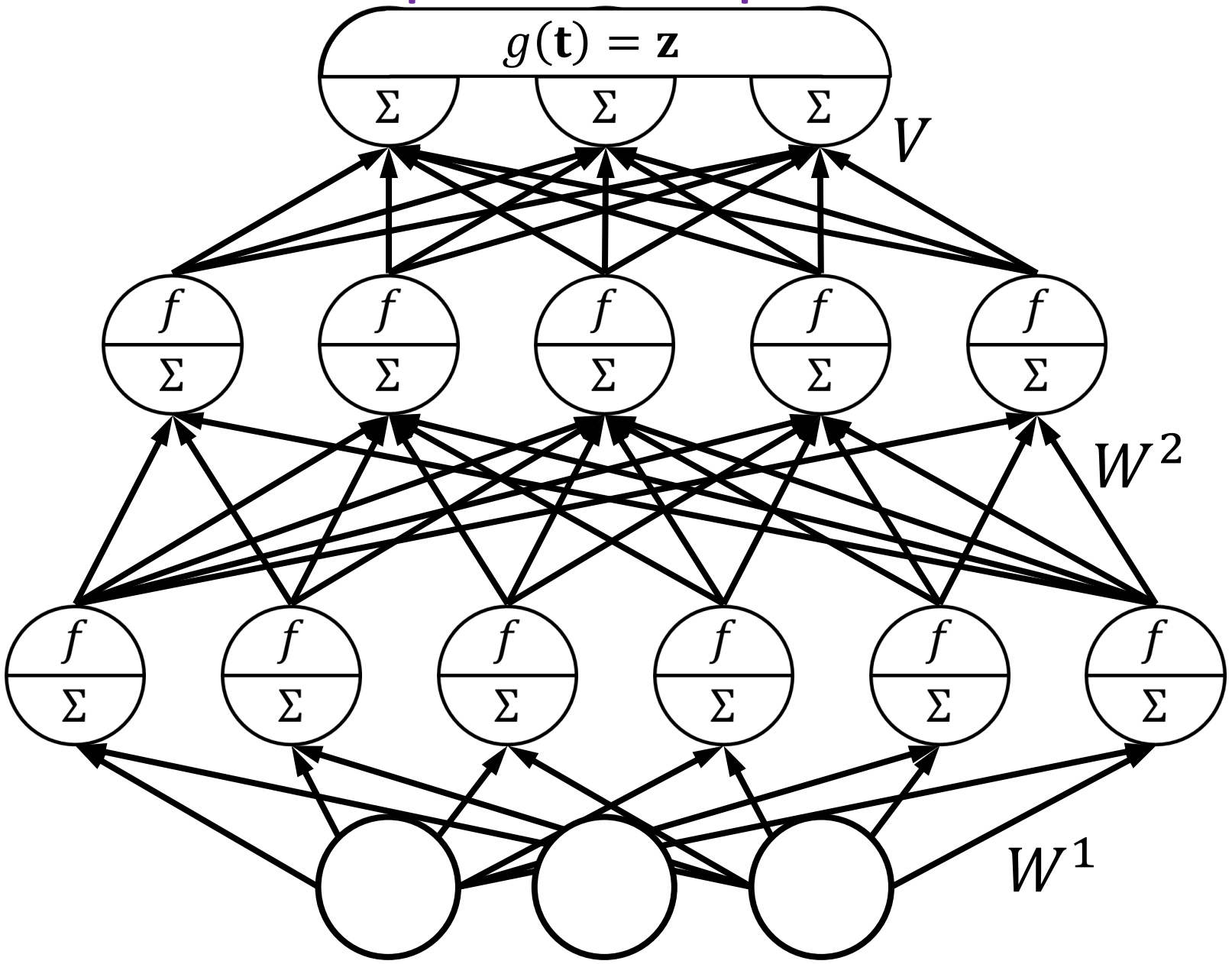
Using NN with multiple outputs to handle multiclass/label problems

Loss functions for DL and commonly associated o/p-layer activations

Training a depth 1 NN (a perceptron) via gradient descent

- *Choice of descent direction – notions of epoch, batch*
- *Techniques to efficiently detect convergence*
- *Techniques to decide on appropriate step length*
- *Techniques to prevent overfitting (NNs are powerful enough to memorize)*

Multi-output Deep Networks



Multi-output Deep Netv

This network learns the function

$$g\left(V^{\mathsf{T}} f\left((W^2)^{\mathsf{T}} f\left((W^1)^{\mathsf{T}}\mathbf{x}\right)\right)\right)$$

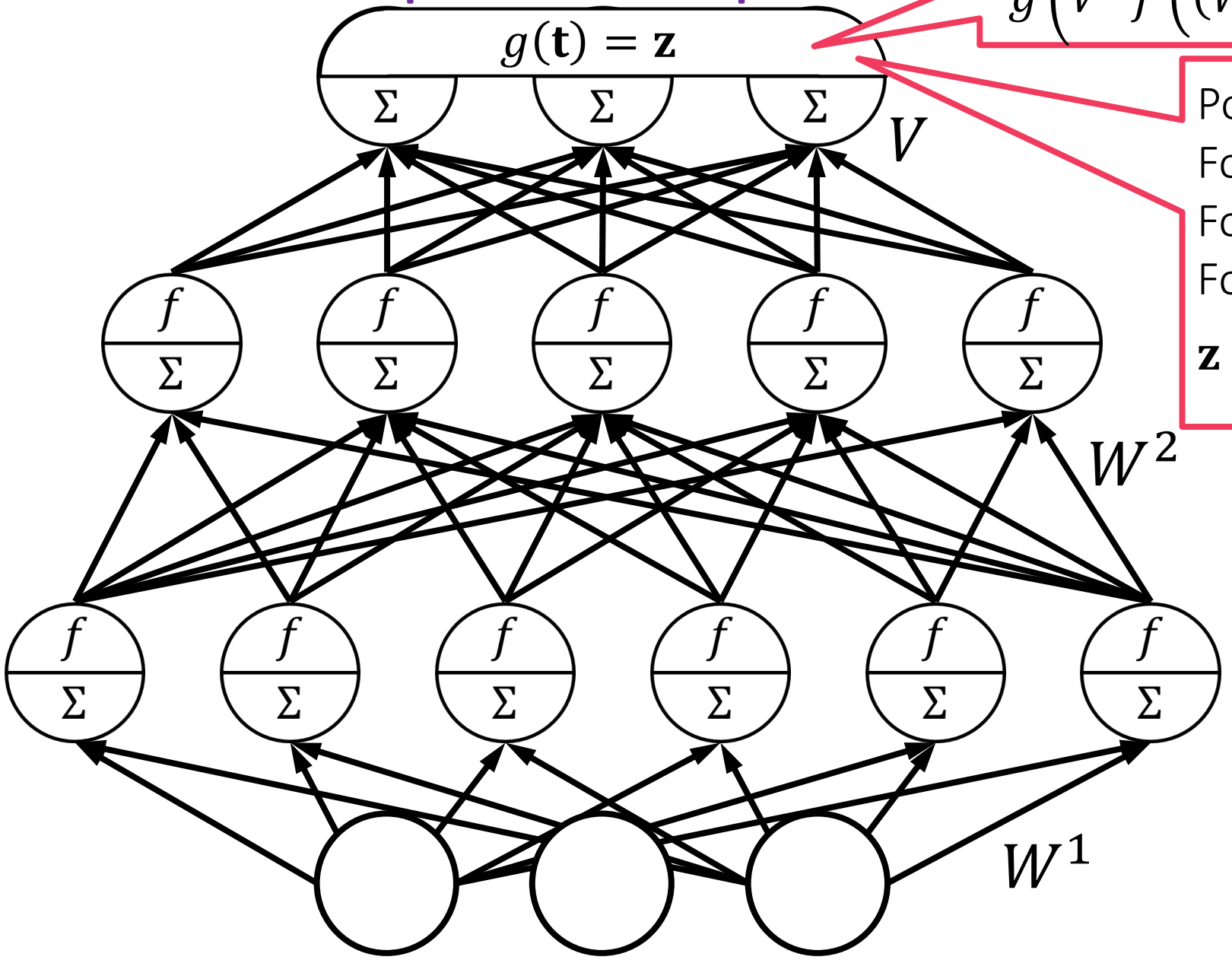
Popular output-layer activation

For vector regression $g(\mathbf{t}) = \mathbf{t}$

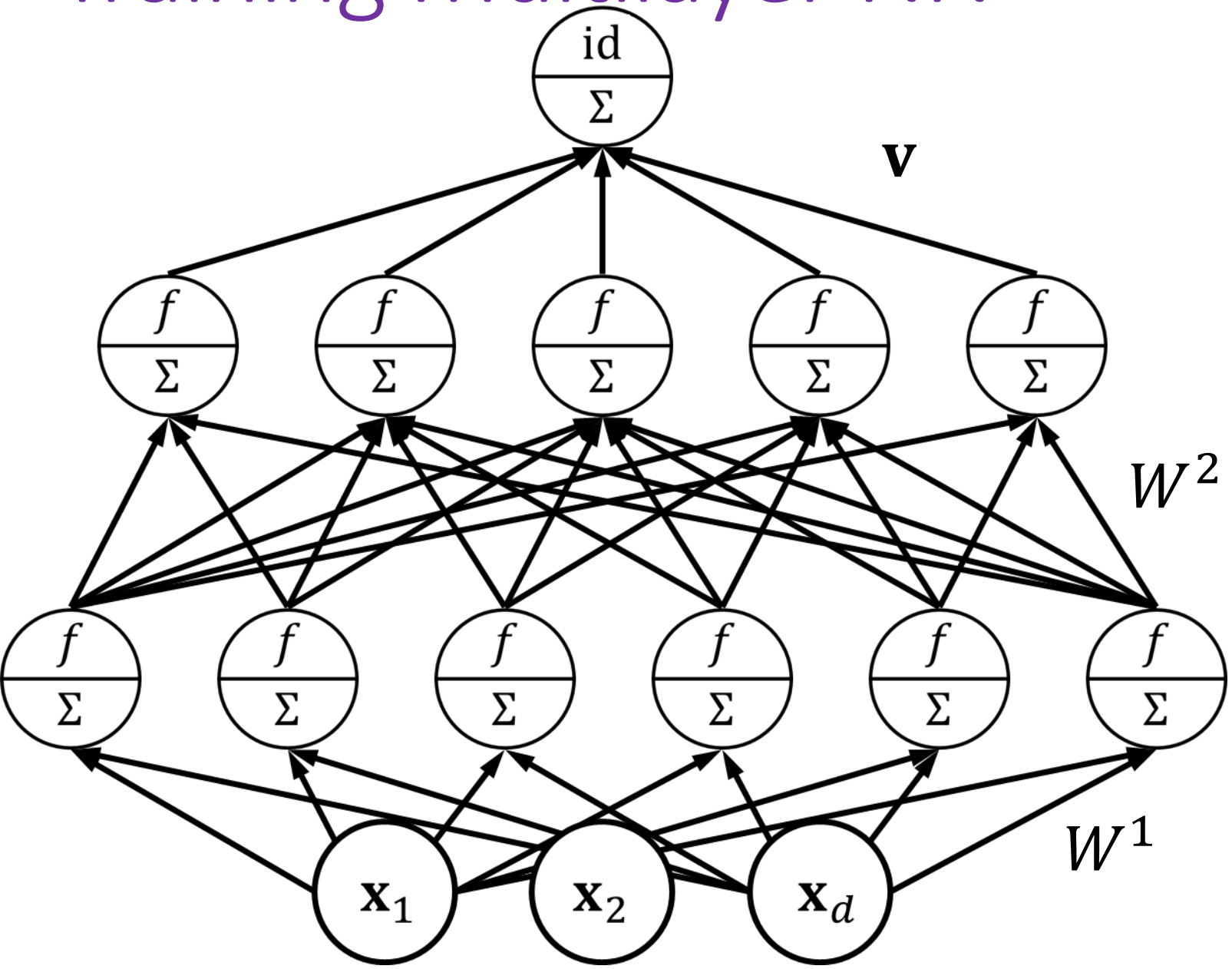
For multilabel $g(\mathbf{t}) = \mathbf{sign}(\mathbf{t})$

For multiclass, softmax

$$\mathbf{z} = g_{\mathrm{SM}}(\mathbf{t}),\ \mathbf{z}_i = \frac{\exp(\mathbf{t}_i)}{\sum_{j=1}^{K}\exp(\mathbf{t}_j)}$$



Training Multilayer NN



d i/p, $L-1$ hidden layers

k_l nodes in l-th layer

$k_1 = d$ *nodes in i/p 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*

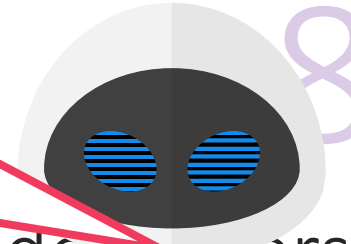
$$\mathbf{h}^l = f\left(\left(W^l\right)^\top \mathbf{h}^{l-1}\right)$$

$$f(\mathbf{u}) = [f(\mathbf{u}_1), \ldots, f(\mathbf{u}_k)]$$

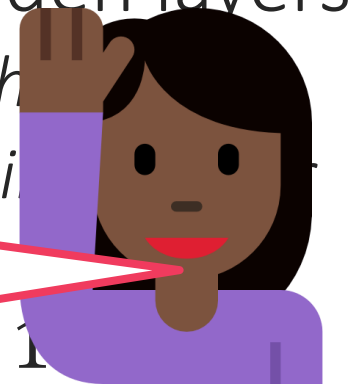
$$\hat{y} = \langle \mathbf{v}, \mathbf{h}^L \rangle$$

Trainin

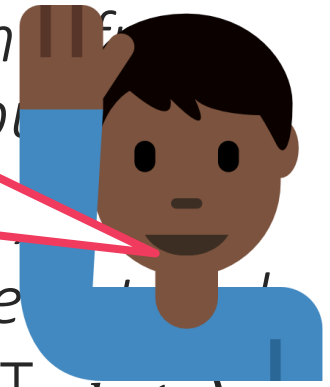
For sake of simplicity, use a single output NN with identity o/p layer activation to study backprop. Similar procedures apply even if multiple outputs with softmax o/p layer activation etc etc

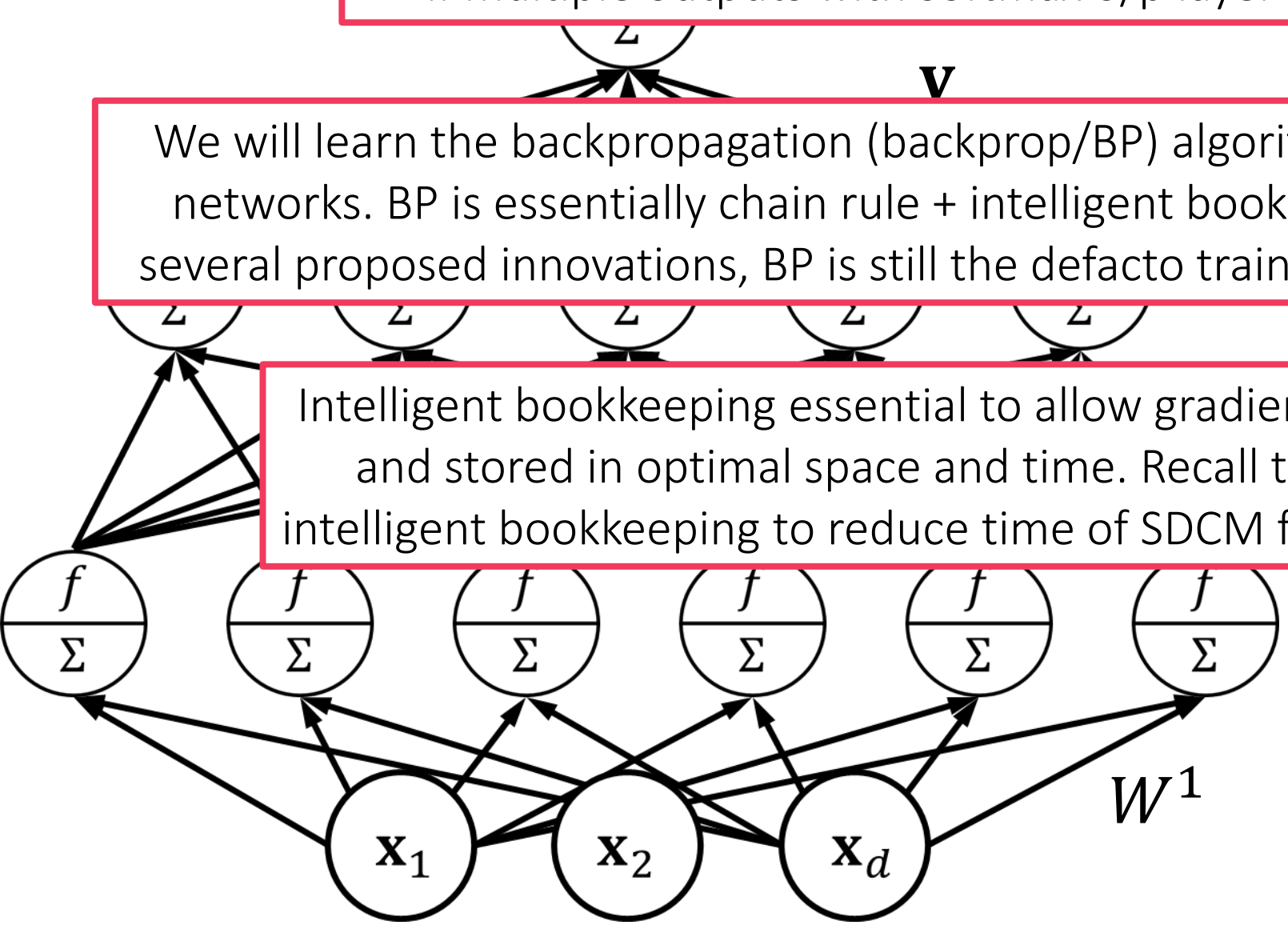


We will learn the backpropagation (backprop/BP) algorithm to train deep networks. BP is essentially chain rule + intelligent bookkeeping. Despite several proposed innovations, BP is still the defacto training method for DL



Intelligent bookkeeping essential to allow gradients to be computed and stored in optimal space and time. Recall that we had used intelligent bookkeeping to reduce time of SDCM from $\mathcal{O}(nd)$ to $\mathcal{O}(n)$





k_l nodes in l-th

from layer $l-1$

$\hat{y}$ output of the

$$\mathbf{h}^l = f\left(\left(W^l\right)^\top \mathbf{h}^{l-1}\right)$$

$$f(\mathbf{u}) = [f(\mathbf{u}_1), \ldots, f(\mathbf{u}_k)]$$

$$\hat{y} = \langle \mathbf{v}, \mathbf{h}^L \rangle$$

Chain Rule Revisited

Let $x = f(y), y = g(z), z = h(w)$ with $x, y, z, w \in \mathbb{R}$, f, g, h: $\mathbb{R} \rightarrow \mathbb{R}$

i.e. $x = f\Big(g\big(h(w)\big)\Big)$. *Chain rule* $\frac{dx}{dw} = \frac{dx}{dy} \cdot \frac{dy}{dz} \cdot \frac{dz}{dw} = f'(y) \cdot g'(z) \cdot h'(w)$

To compute $\frac{dx}{dw}$ at a point $w = w^0$ the above rule gives us a simple algo

First compute $z^0 = h(w^0), y^0 = g(z^0)$

Then compute $r = h'(w^0), q = g'(z^0), p = f'(y^0)$

Finally compute $\frac{dx}{dw} = p \cdot q \cdot r$

Backprop uses the multivariate version of this rule

Recall that if $x = f(\mathbf{y})$ *where* $x \in \mathbb{R}, \mathbf{y} \in \mathbb{R}^q$ *and* f: $\mathbb{R}^q \rightarrow \mathbb{R}$ *we have notion of gradient of the function that gives us* $\frac{dx}{d\mathbf{y}} = \nabla f(\mathbf{y}) \in \mathbb{R}^{1 \times q}$

Convention*: treat gradients as row vectors to make notation clean*

Multivariate Chain Rule

If we now have a multivariate function $\mathbf{x} = f(\mathbf{y})$ where $\mathbf{x} \in \mathbb{R}^p, \mathbf{y} \in \mathbb{R}^q$ and $f\colon \mathbb{R}^q \to \mathbb{R}^p$, then we have the notion of the *Jacobian*

Think of $f(\mathbf{y}) = \left[f_1(\mathbf{y}), \ldots, f_p(\mathbf{y})\right]^\top$ *as a vector of* p *real-valued functions*

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

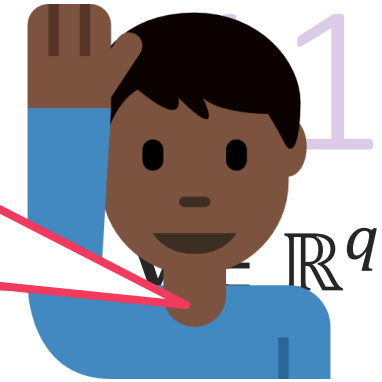
All that Jacobian does is consider $f_i(\cdot)\colon \mathbb{R}^q \to \mathbb{R}, i = 1, \ldots, p$ *separately, find their (row) gradient vectors, and stack them vertically as a matrix*

If $x = f(\mathbf{y}), \mathbf{y} = g(\mathbf{z}), x \in \mathbb{R}, \mathbf{y} \in \mathbb{R}^q, \mathbf{z} \in \mathbb{R}^r$ $f\colon \mathbb{R}^q \to \mathbb{R}, g\colon \mathbb{R}^r \to \mathbb{R}^q$ (actually happens in NN) then chain rule changes its form

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

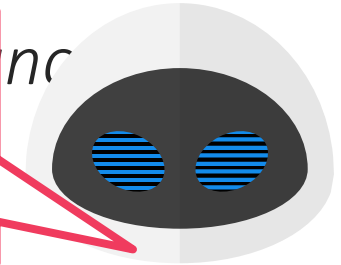
Mu

In general if we have a function $f: \mathbb{R}^p \rightarrow \mathbb{R}^{m \times n}$, its Jacobian is a 3rd order tensor (a 3D matrix) of dimensionality $\mathbb{R}^{m \times n \times p}$. If $g: \mathbb{R}^{p \times q} \rightarrow \mathbb{R}^{m \times n}$, its Jacobian is a 4th order tensor (a 4D matrix) of dimensionality $\mathbb{R}^{m \times n \times p \times q}$. In general very easy to predict dimensionality of Jacobian



If we … $\mathbb{R}^q$ and $J: \mathbb{R}^q \rightarrow \mathbb{R}^p$, then we have the notion of the *Jacobian*

Yes, it is quite simply a tensor whose dimensionality is the output dimensionality of the function "times" the input dimensionality of the function



Think of $f(\mathbf{y}) = \begin{bmatrix} f_1(\\ \end{bmatrix}$ … *nc*

$$\frac{\mathbf{dx}}{\mathbf{dy}} \triangleq J^f = \begin{bmatrix} \frac{\partial f_1(\mathbf{y})}{\partial \mathbf{y}_1} & \cdots & \frac{\partial f_1(\mathbf{y})}{\partial \mathbf{y}_q} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_p(\mathbf{y})}{\partial \mathbf{y}_1} & \cdots & \frac{\partial f_p(\mathbf{y})}{\partial \mathbf{y}_q} \end{bmatrix} = \begin{bmatrix} \nabla f_1(\mathbf{y}) \\ \vdots \\ \nabla f_p(\mathbf{y}) \end{bmatrix}$$

All that Jacobian does is consider $f_i(\cdot): \mathbb{R}^q \rightarrow \mathbb{R}, i = 1, \ldots, p$ *separately, find their (row) gradient vectors, and stack them vertically as a matrix*

If $x = f(\mathbf{y}), \mathbf{y} = g(\mathbf{z}), x \in \mathbb{R}, \mathbf{y} \in \mathbb{R}^q, \mathbf{z} \in \mathbb{R}^r$ $f: \mathbb{R}^q \rightarrow \mathbb{R}, g: \mathbb{R}^r \rightarrow \mathbb{R}^q$ (actually happens in NN) then chain rule changes its form

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

Multivariate Chain Rule

However, the above is simplified greatly by use of Jacobians

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

where J^g is the Jacobian of g and $\nabla f(\mathbf{y})$ is the Jacobian (gradient) of f

Jacobian of real-valued functions is simply their gradient (row) vector

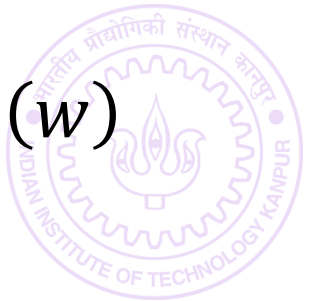
Notion of Jacobian allows us to apply chain rule to very general settings

If $\mathbf{x} = f(\mathbf{y}), \mathbf{y} = g(\mathbf{z}), \mathbf{z} = h(\mathbf{w})$ with $\mathbf{x} \in \mathbb{R}^p, \mathbf{y} \in \mathbb{R}^q, \mathbf{z} \in \mathbb{R}^r, \mathbf{w} \in \mathbb{R}^s$

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

Very elegant and mimics the real-valued case $\frac{dx}{dw} = f'(y) \cdot g'(z) \cdot h'(w)$

If we are careful with dimensions, Jacobians greatly simplify life

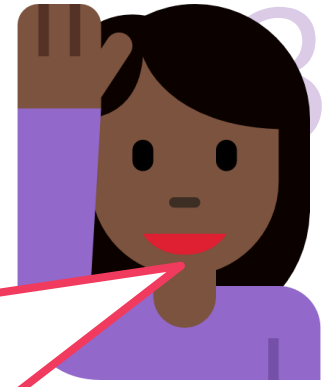


M

H

We may verify that Jacobian notation does preserve old results. 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 fn on vector space), Jacobian is a row vec
- If $p > 1$ and $q = 1$ and no $\mathbf{w}$ (vector valued function on the real line), Jacobian becomes a column vector



$$\sum_{i=1}$$

where J^g is the Jacobian of g and $\nabla f(\mathbf{y})$ is the Jacobian (gradient) of f

Jacobian of real-valued functions is simply their gradient (row) vector

Notion of Jacobian allows us to apply chain rule to very general settings

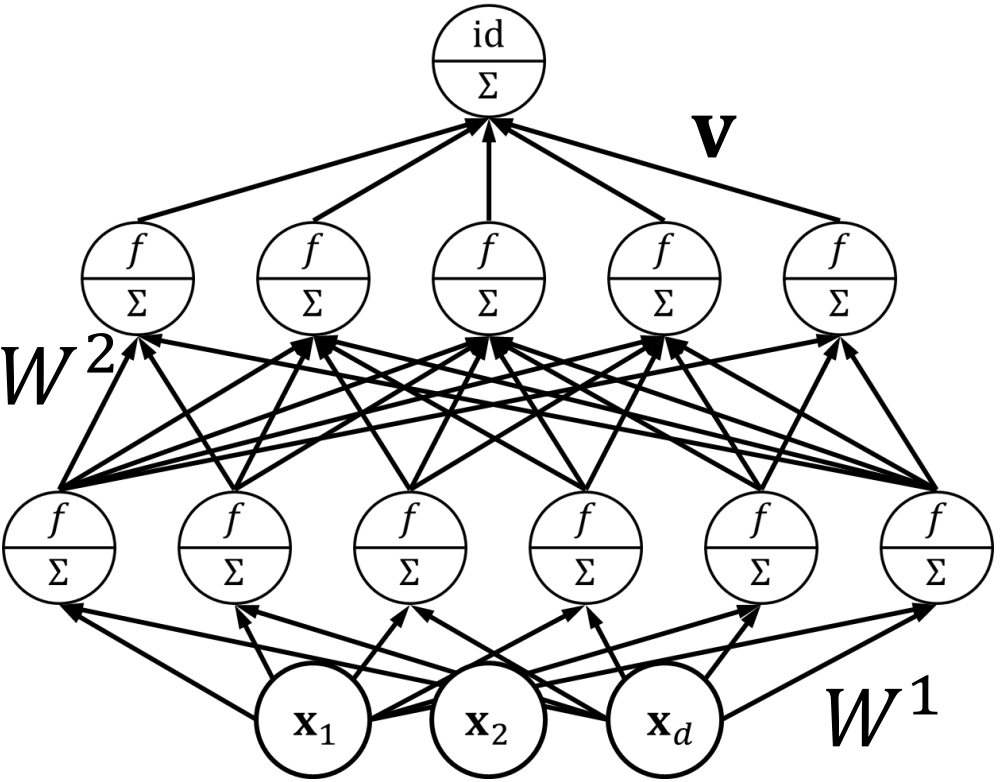
If $\mathbf{x} = f(\mathbf{y}), \mathbf{y} = g(\mathbf{z}), \mathbf{z} = h(\mathbf{w})$ with $\mathbf{x} \in \mathbb{R}^p, \mathbf{y} \in \mathbb{R}^q, \mathbf{z} \in \mathbb{R}^r, \mathbf{w} \in \mathbb{R}^s$

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

Very elegant and mimics the real-valued case $\frac{dx}{dw} = f'(y) \cdot g'(z) \cdot h'(w)$

If we are careful with dimensions, Jacobians greatly simplify life

Back to Backpropagation



NN output is $\hat{y} = \mathbf{v}^{\mathsf{T}} f\left((W^2)^{\mathsf{T}} f\left((W^1)^{\mathsf{T}}\mathbf{x}\right)\right)$

We wish to minimize the loss on train data

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

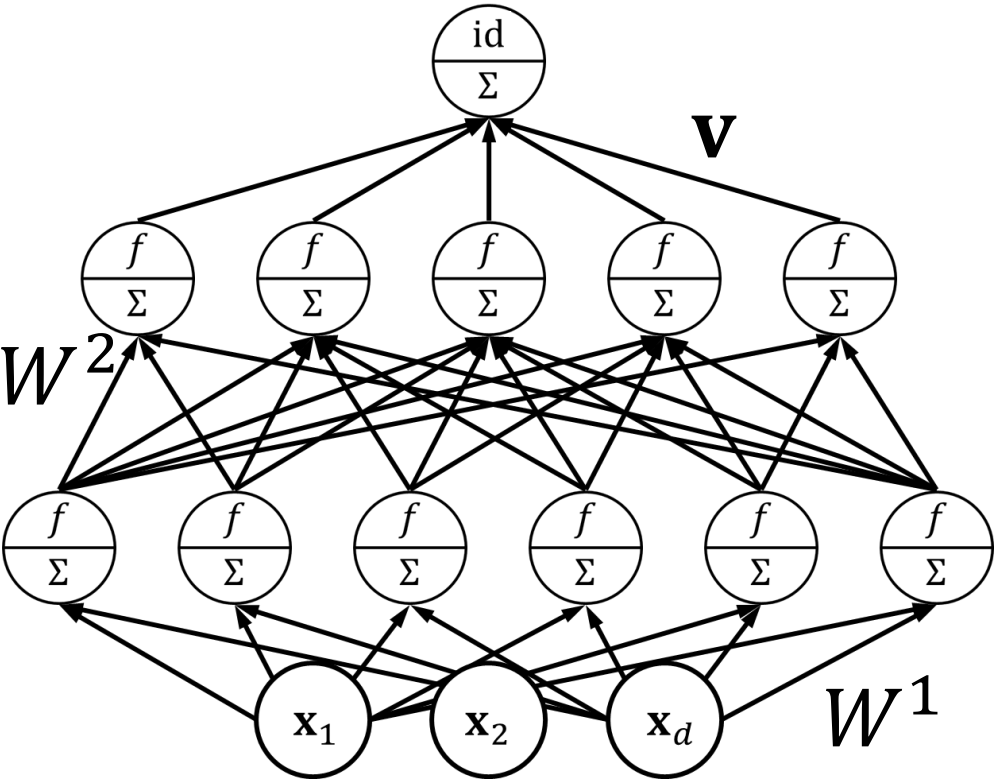
We want to apply GD updates of the form

$$\mathbf{v} \leftarrow \mathbf{v} - \eta \cdot \frac{\partial F}{\partial \mathbf{v}} \quad W^i \leftarrow W^i - \eta \cdot \frac{\partial F}{\partial W^i} \; i = 1,2$$

Thus, all we need to calculate are

$$\frac{\partial \ell^i}{\partial \mathbf{v}}, \frac{\partial \ell^i}{\partial W^2}, \frac{\partial \ell^i}{\partial W^1}$$

Back to Backpropagation



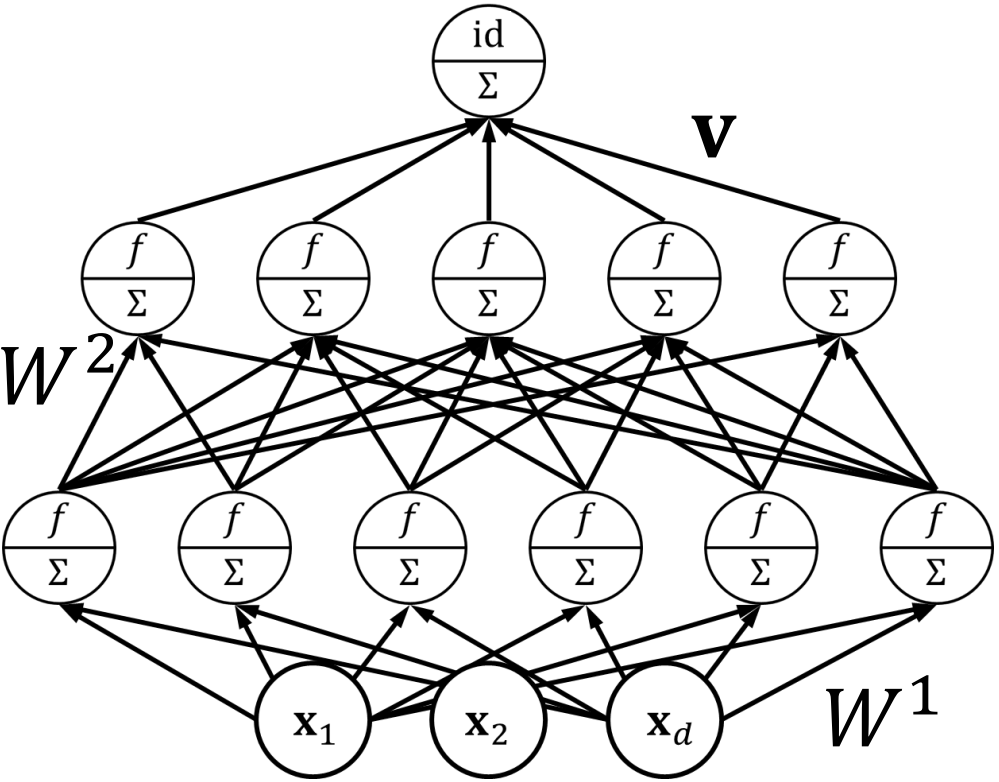
First is easy as we have $\ell^i = \ell\left(\hat{y}^i, y^i\right)$ and $\hat{y}^i = \mathbf{v}^\top \mathbf{h}^{2,i}$ where $\mathbf{h}^{2,i}$ is (post-activation) o/p of 2nd hidden layer on i-th data point

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

Verify that $\frac{d\ell^i}{d\mathbf{v}}$ does have dimensions of $\mathbf{v}^\top$ and can hence be used in GD updates for $\mathbf{v}$

Recall that we switched to row convention for gradients so this makes sense

Back to Backpropagation



$\frac{d\ell^i}{dW^2}$ is more fun – ℓ^i depends only on $\hat{y}^i$ so

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

$\hat{y}^i = \mathbf{v}^\top \mathbf{h}^{2,i}$ *depends on* $\mathbf{v}$ *and* $\mathbf{h}^{2,i}$

$$\frac{d\hat{y}^i}{dW^2} = \frac{d\hat{y}^i}{d\mathbf{h}^{2,i}} \cdot \frac{d\mathbf{h}^{2,i}}{dW^2} + \frac{d\hat{y}^i}{d\mathbf{v}} \cdot \frac{d\mathbf{v}}{dW^2} \text{ but } \frac{d\mathbf{v}}{dW^2} = \mathbf{0}$$

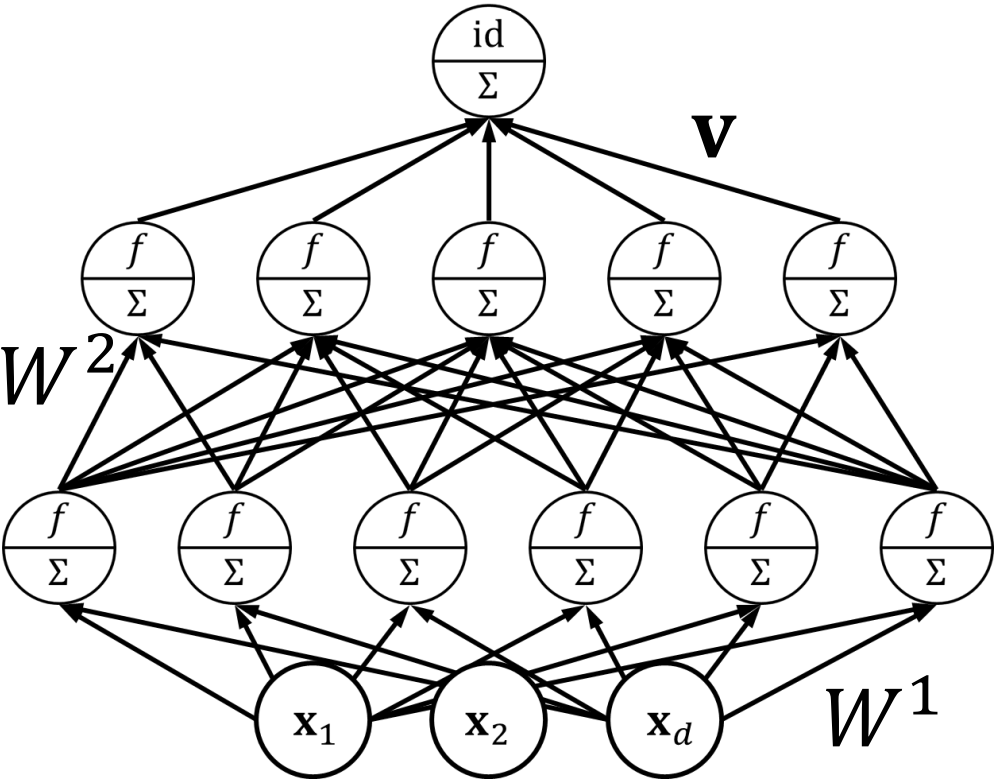
Since we assume $\mathbf{v}$ *is not a function of* W^2

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

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

Now we have to find out $d\mathbf{h}^{2,i}/dW^2$

Back to Backpropagation



We have $\mathbf{h}^{2,i} = f\left((W^2)^\top \mathbf{h}^{1,i}\right) = f\left(\mathbf{a}^{2,i}\right)$

$\mathbf{a}^{2,i}$ *are the pre-activations of second layer*

Thus, we have $\frac{d\mathbf{h}^{2,i}}{dW^2} = \frac{d\mathbf{h}^{2,i}}{d\mathbf{a}^{2,i}} \cdot \frac{d\mathbf{a}^{2,i}}{dW^2}$

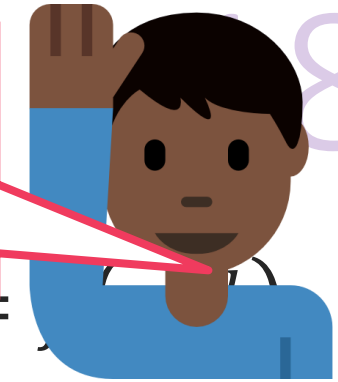
$\frac{d\mathbf{h}^{2,i}}{d\mathbf{a}^{2,i}} = \text{diag}\left(f'\left(\mathbf{a}_1^{2,i}\right) \ldots f'\left(\mathbf{a}_{k_3}^{2,i}\right)\right) \in \mathbb{R}^{k_3 \times k_3}$

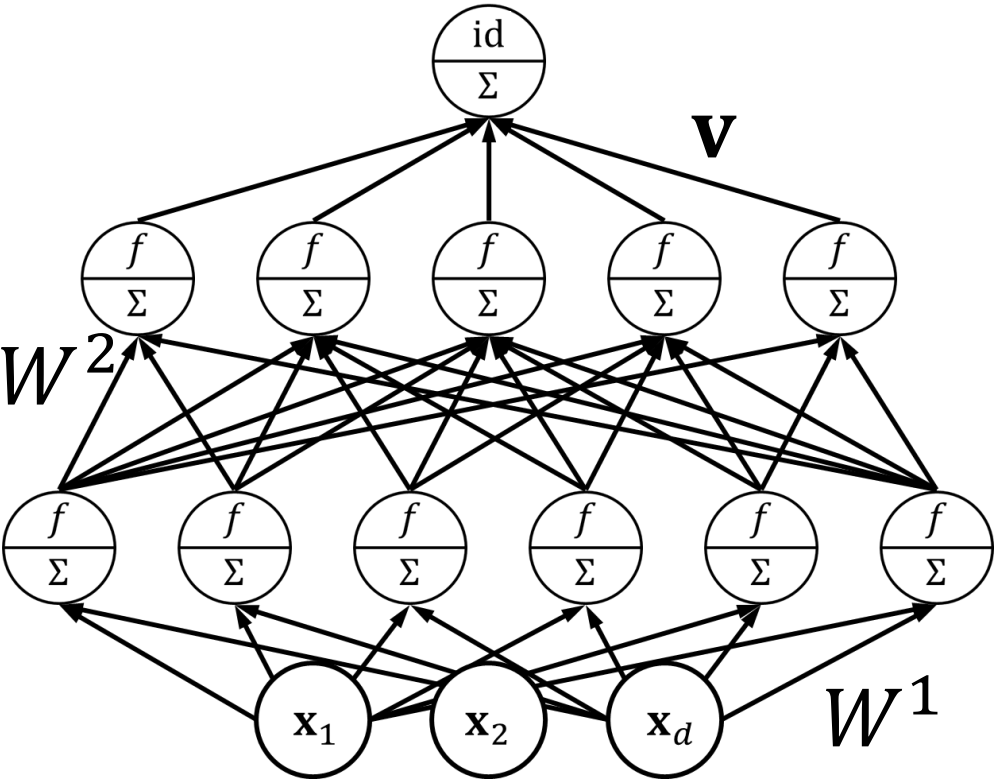
This is since $\mathbf{h}^{2,i} = f\left(\mathbf{a}^{2,i}\right)$

$\frac{d\mathbf{a}^{2,i}}{dW^2}$ on the other hand, is a 3D tensor (matrix) of dimensionality $k_3 \times k_3 \times k_2$

This is why *Tensor*Flow is named so ☺

Overall we have $\frac{d\ell^i}{dW^2} = \ell'(\hat{y}^i, y^i) \cdot \mathbf{v}^\top \cdot \frac{d\mathbf{h}^{2,i}}{d\mathbf{a}^{2,i}} \cdot \frac{d\mathbf{a}^{2,i}}{dW^2}$. The dimensionality of $\frac{d\ell^i}{dW^2}$ is $\mathbb{R}^{1\times 1} \cdot \mathbb{R}^{1\times k_3} \cdot \mathbb{R}^{k_3\times k_3} \cdot \mathbb{R}^{k_3\times k_3\times k_2} = \mathbb{R}^{1\times k_3\times k_2}$ which is as expected





We have $\mathbf{h}^{2,i} = f\left((W^2)^\top \mathbf{h}^{1,i}\right) =$

$\mathbf{a}^{2,i}$ *are the pre-activations of second layer*

Thus, we have $\frac{d\mathbf{h}^{2,i}}{dW^2} = \frac{d\mathbf{h}^{2,i}}{d\mathbf{a}^{2,i}} \cdot \frac{d\mathbf{a}^{2,i}}{dW^2}$

$\frac{d\mathbf{h}^{2,i}}{d\mathbf{a}^{2,i}} = \text{diag}\left(f'\left(\mathbf{a}_1^{2,i}\right) \ldots f'\left(\mathbf{a}_{k_3}^{2,i}\right)\right) \in \mathbb{R}^{k_3\times k_3}$

This is since $\mathbf{h}^{2,i} = f\left(\mathbf{a}^{2,i}\right)$

$\frac{d\mathbf{a}^{2,i}}{dW^2}$ on the other hand, is a 3D tensor (matrix) of dimensionality $k_3 \times k_3 \times k_2$

This is why *Tensor*Flow is named so ☺

Bookkeeping in Backprop

Thus, we derive the update

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

$$\frac{d\ell^i}{dW^2} = \ell'\left(\hat{y}^i, y^i\right) \cdot \mathbf{v}^\top \cdot \frac{d\mathbf{h}^{2,i}}{d\mathbf{a}^{2,i}} \cdot \frac{d\mathbf{a}^{2,i}}{dW^2}, \text{ and}$$

$$\frac{d\ell^i}{dW^1} = \ell'\left(\hat{y}^i, y^i\right) \cdot \mathbf{v}^\top \cdot \frac{d\mathbf{h}^{2,i}}{d\mathbf{a}^{2,i}} \cdot \frac{d\mathbf{a}^{2,i}}{d\mathbf{h}^{1,i}} \cdot \frac{d\mathbf{h}^{1,i}}{d\mathbf{a}^{1,i}} \cdot \frac{d\mathbf{a}^{1,i}}{dW^1}$$

Terms such as $\ell'\left(\hat{y}^i, y^i\right)$, $\frac{d\mathbf{h}^{2,i}}{d\mathbf{a}^{2,i}}$ *used repeatedly in multiple expressions*

TensorFlow, PyTorch do bookkeeping to avoid repeated computation

Save time but also use up a lot of memory (which GPUs don't always have ☹)

Note: $\frac{d\mathbf{h}^{l,i}}{d\mathbf{a}^{l,i}} = \text{diag}\left(f'\left(\mathbf{a}_1^{l,i}\right) \ldots f'\left(\mathbf{a}_{k_{l+1}}^{l,i}\right)\right)$, *if* f' *is small (e.g. sigmoid), then the terms multiplied together may become very small – vanishing gradients*

One reason why ReLU etc preferred – do not have vanishing gradients issue

Generative Neural Networks

- So far we have looked at networks that solve prediction problems like (binary,multiclass/label)classification, regression etc
- Neural networks can also be used to learn generative models
- Can be used for dim. redn. as well as to generate new data points
- (Variational) Autoencoders (Kingma & Welling 2013, Rezende et al. 2014) and Generative adversarial networks (Goodfellow et al 2014) are two popular generative models that use neural networks
- Will describe them with feedforward networks but can be formed out of CNNs or RNNs as well

Autoencoders

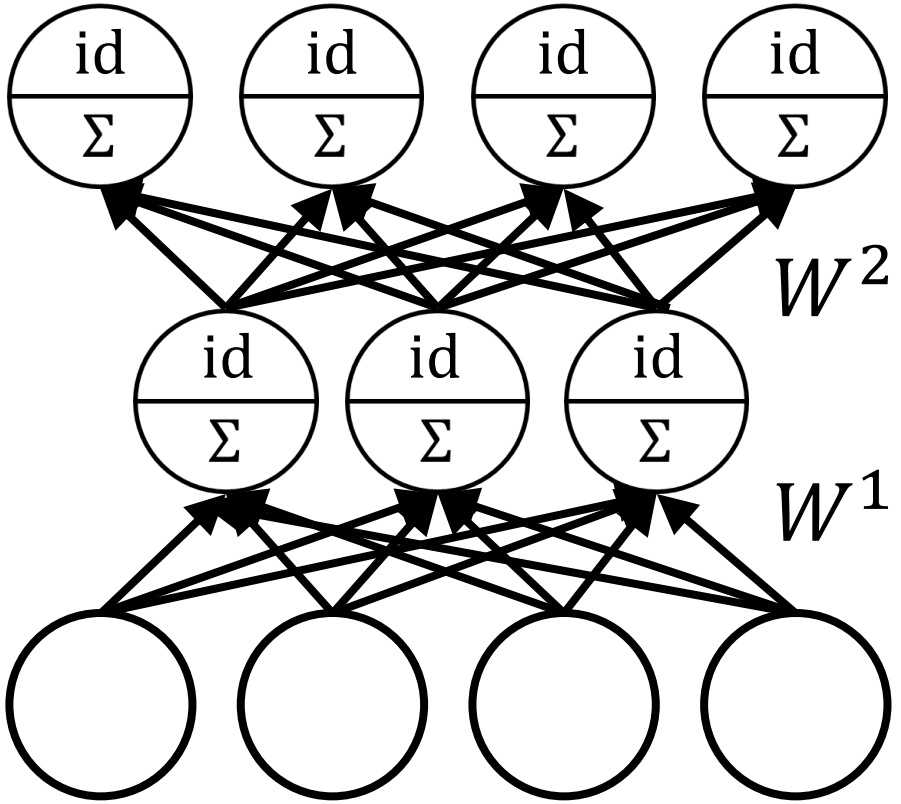
In PCA, a low dim latent variable $\mathbf{z}$ generated the data as $\mathbf{x} = W\mathbf{z} + \boldsymbol{\epsilon}$

***Note**: a linear map (factor loading matrix) was applied to $\mathbf{z}$ to obtain data*

If we find W, then we could recover the low-dim $\mathbf{z} = W^\top \mathbf{x}$

PCA offers the best reconstruction error

$$\|\mathbf{x} - WW^\top \mathbf{x}\|_2^2$$



Autoencoders do this using non-linear maps

An "encoder" converts data to a latent representation $\mathbf{h} = e(\mathbf{x})$

A "decoder" produces a reconstruction $\mathbf{r} = d(\mathbf{h})$

Want to minimize loss $\ell(d(\mathbf{h}), \mathbf{x}) = \ell\big(d\big(e(\mathbf{x})\big), \mathbf{x}\big)$

Autoencoders

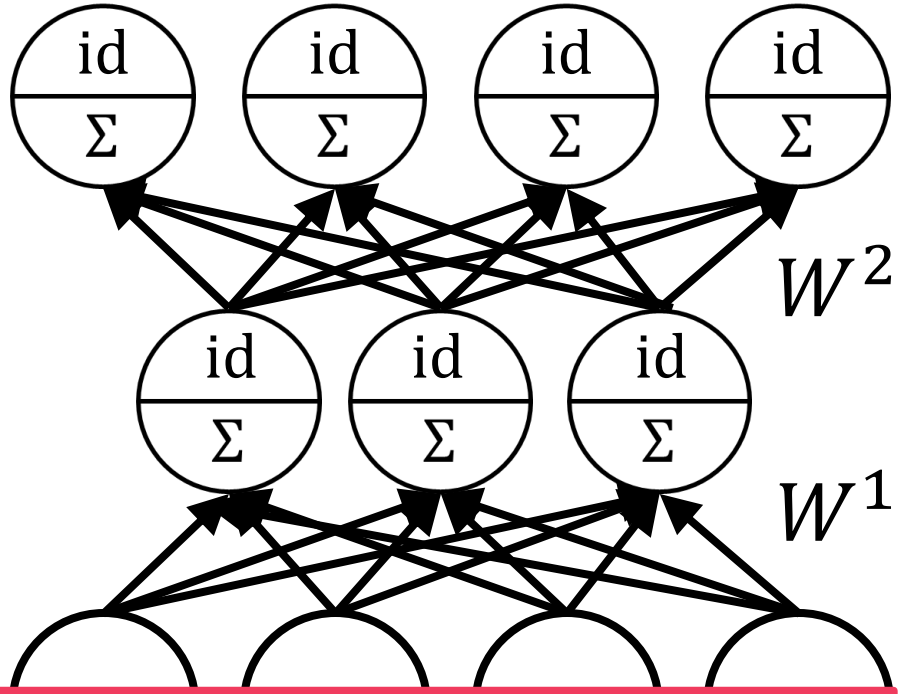
In PCA, a low dim latent variable $\mathbf{z}$ generated the data as $\mathbf{x} = W\mathbf{z} + \boldsymbol{\epsilon}$

***Note**: a linear map (factor loading matrix) was applied to $\mathbf{z}$ to obtain data*

If we find W, then we could recover the low-dim $\mathbf{z} = W^\top \mathbf{x}$

PCA offers the best reconstruction error

$$\|\mathbf{x} - WW^\top \mathbf{x}\|_2^2$$



Autoencoders do this using non-linear maps

An "encoder" converts data to a latent representation $\mathbf{h} = e(\mathbf{x})$

A "decoder" produces a reconstruction $\mathbf{r} = d(\mathbf{h})$

Want to minimize loss $\ell(d(\mathbf{h}), \mathbf{x}) = \ell\big(d\big(e(\mathbf{x})\big), \mathbf{x}\big)$

E.g., Euclidean error

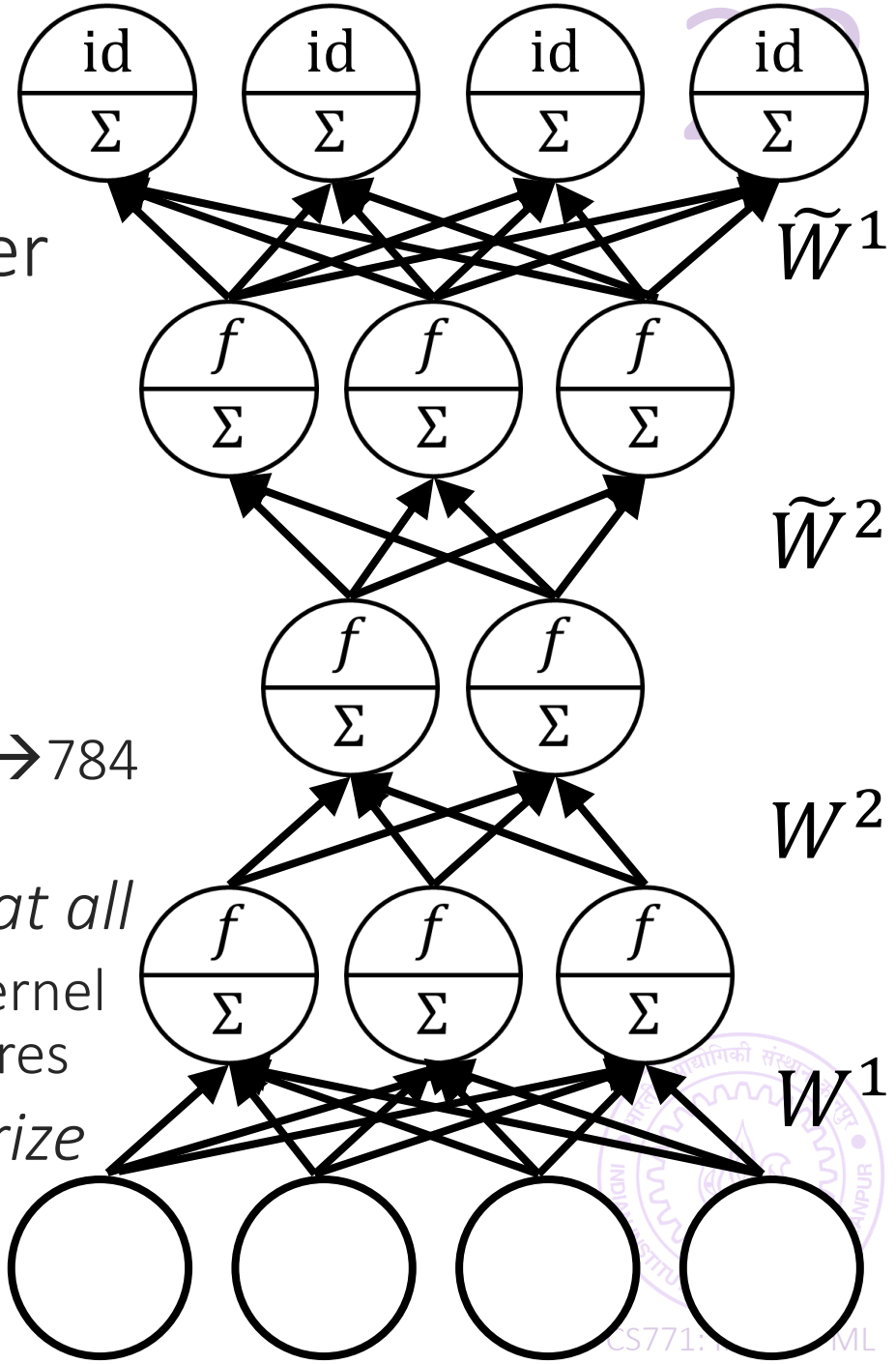
$$\ell\big(d\big(e(\mathbf{x})\big), \mathbf{x}\big) = \big\|d\big(e(\mathbf{x})\big) - \mathbf{x}\big\|_2^2$$

Can use backprop to train AE NN

Autoencoders

PCA can be thought of as a linear autoencoder

- *Just like PCA, autoencoders are trained to lower reconstruction error using backpropagation*
- *The "hourglass" architecture very popular in AE*
- *Gradually decreases dimensionality of data*
  - However, a monotonic decrease not necessary
  - E.g. 784 →1000→500→250→30→250→500→1000→784 on MNIST data (Hinton and Salakhutdinov)
- *In fact some autoencoders do not decrease dim at all*
  - Called overcomplete autoencoders. Recall that as in kernel PCA, goal may be to find better features not less features
- *Autoencoders can overfit badly too – just memorize data and offer no useful dim red on test data*
- *Various techniques to regularize AE*



Autoencoders

PCA can be thought of as a linear autoencoder

Just like PCA, autoencoders are trained to lower

"Bottleneck". For overcomplete AE, bottleneck may have more neurons than input layer. For undercomplete neurons, less neurons than input layer in bottleneck layer so dim redn

However, a monotonic decrease not necessary

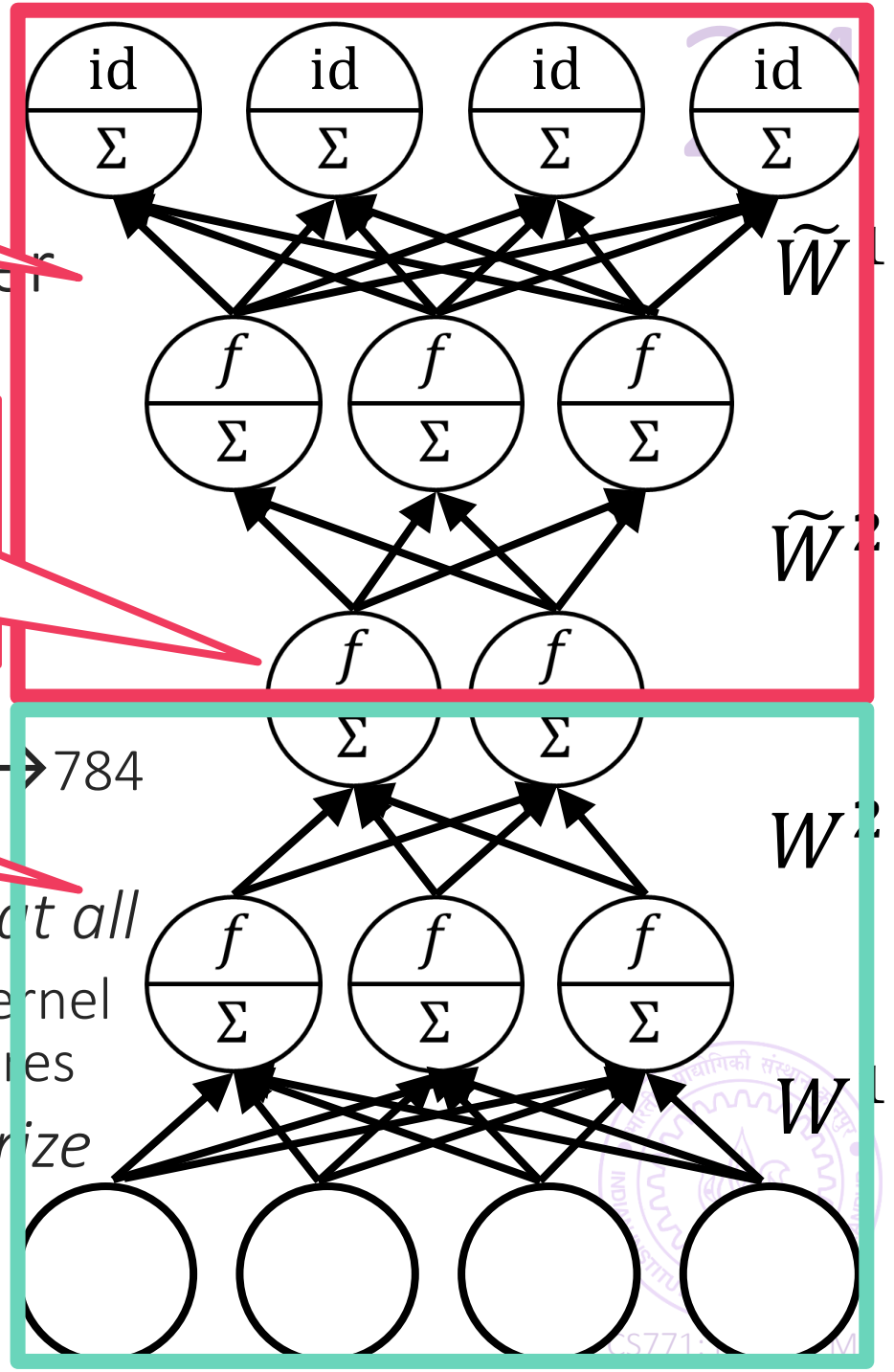
E.g. 784 →1000→500→250→30[illegible]000→784
on MNIST data (Hinton and Salakh[illegible]

In fact some autoencoders do not decrease dim at all

Called overcomplete autoencoders. Recall that as in kernel PCA, goal may be to find better features not less features

Autoencoders can overfit badly too – just memorize data and offer no useful dim red on test data

Various techniques to regularize AE



Autoencoders

PCA can be thought of as a linear autoencoder

Just like PCA, autoencoders are trained to lower

"Bottleneck". For overcomplete AE, bottleneck may have more neurons than input layer. For undercomplete neurons, less neurons than input layer in bottleneck layer so dim redn

However, a monotonic decrease not necessary

E.g. 784 →1000→500→250→30[illegible]000→784

on MNIST data (Hinton and Salakh[illegible]

In fact some autoencoders do not decrease dim at all

Called[illegible]

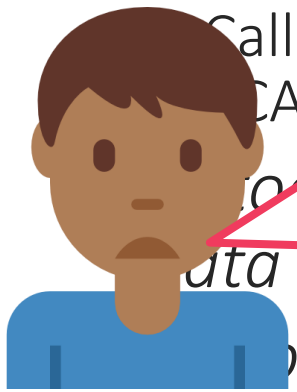
CA,

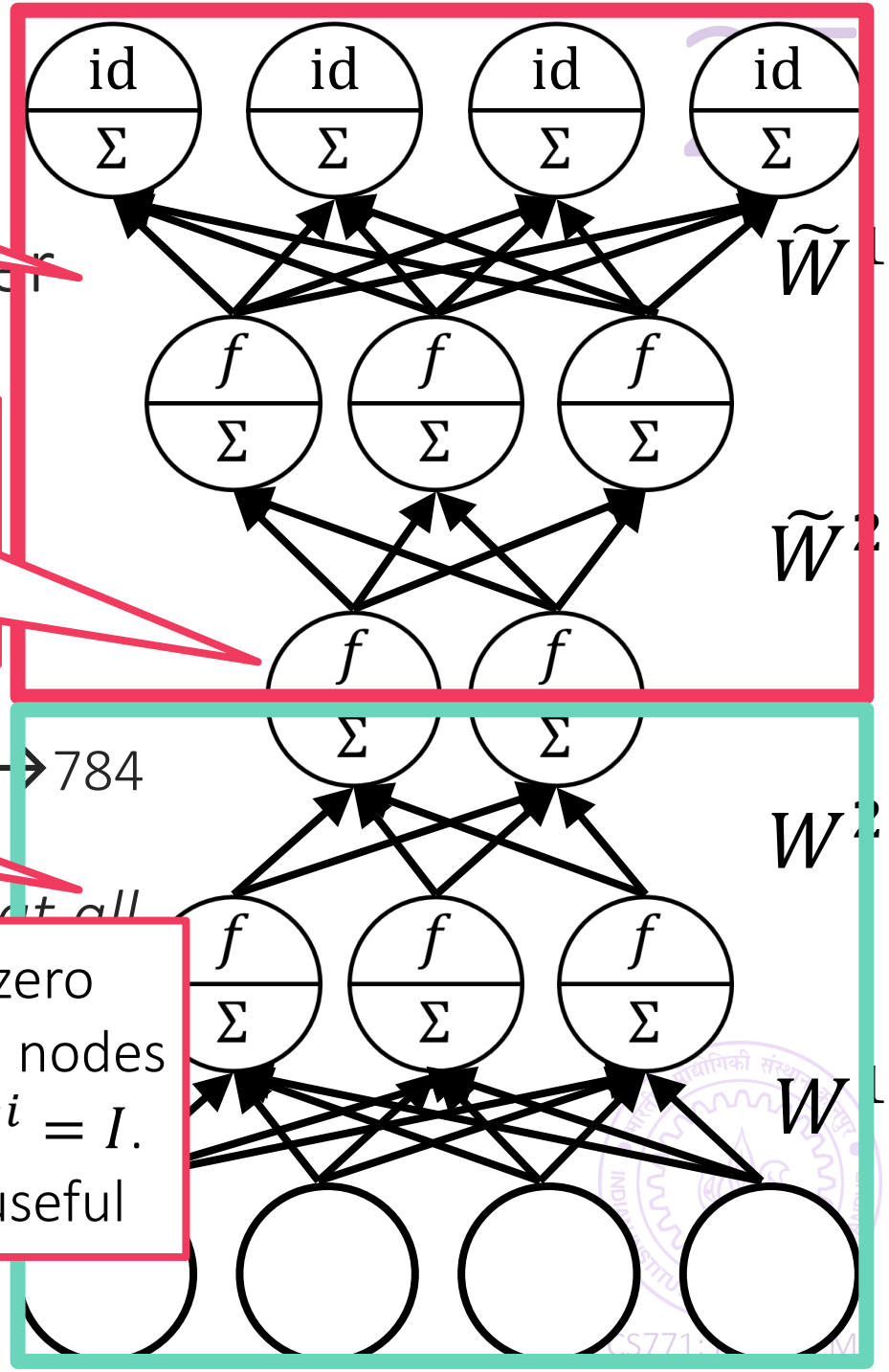
[illegible]

ata a

ous techniques to regularize AE

Indeed, with overcomplete AE, we can have zero reconstruction error by simply having all hidden nodes use identity activation and making sure that $W^i = I$. However, such an AE has not learnt anything useful





Regularized Autoencoders

Techniques similar to we have studied before are used

Method 1 *(Denoising AE) Add noise to i/p feat but expect original feat at o/p*

$\ell\big(d\big(e(\mathbf{x}+\boldsymbol{\epsilon})\big),\mathbf{x}\big) = \big\|d\big(e(\mathbf{x}+\boldsymbol{\epsilon})\big) - \mathbf{x}\big\|_2^2$, $\boldsymbol{\epsilon}$ *is noise added to input*

Method 2 *(Sparse AE) Force the bottleneck layer activations to be sparse*

For every data point, penalize the AE if the vector $e\big(\mathbf{x}^i\big), i = 1 \ldots n$ has lots of non-zeros

This trick can allow even overcomplete AE to be trained without risk of overfitting

Method 3 *(Variational AE) Force bottleneck layer activations to fit to a Gaussian*

Find $\boldsymbol{\mu}, \Sigma$ and train encoder and decoder e, d such that not only does reconstruction loss become small but also the latent representations $e\big(\mathbf{x}^i\big), i = 1 \ldots n$ are well approximated by the Gaussian $\mathcal{N}(\boldsymbol{\mu}, \Sigma)$

Can make this more powerful by taking mixture of Gaussians instead of a single Gaussian

A variational AE (VAE) can easily generate new samples

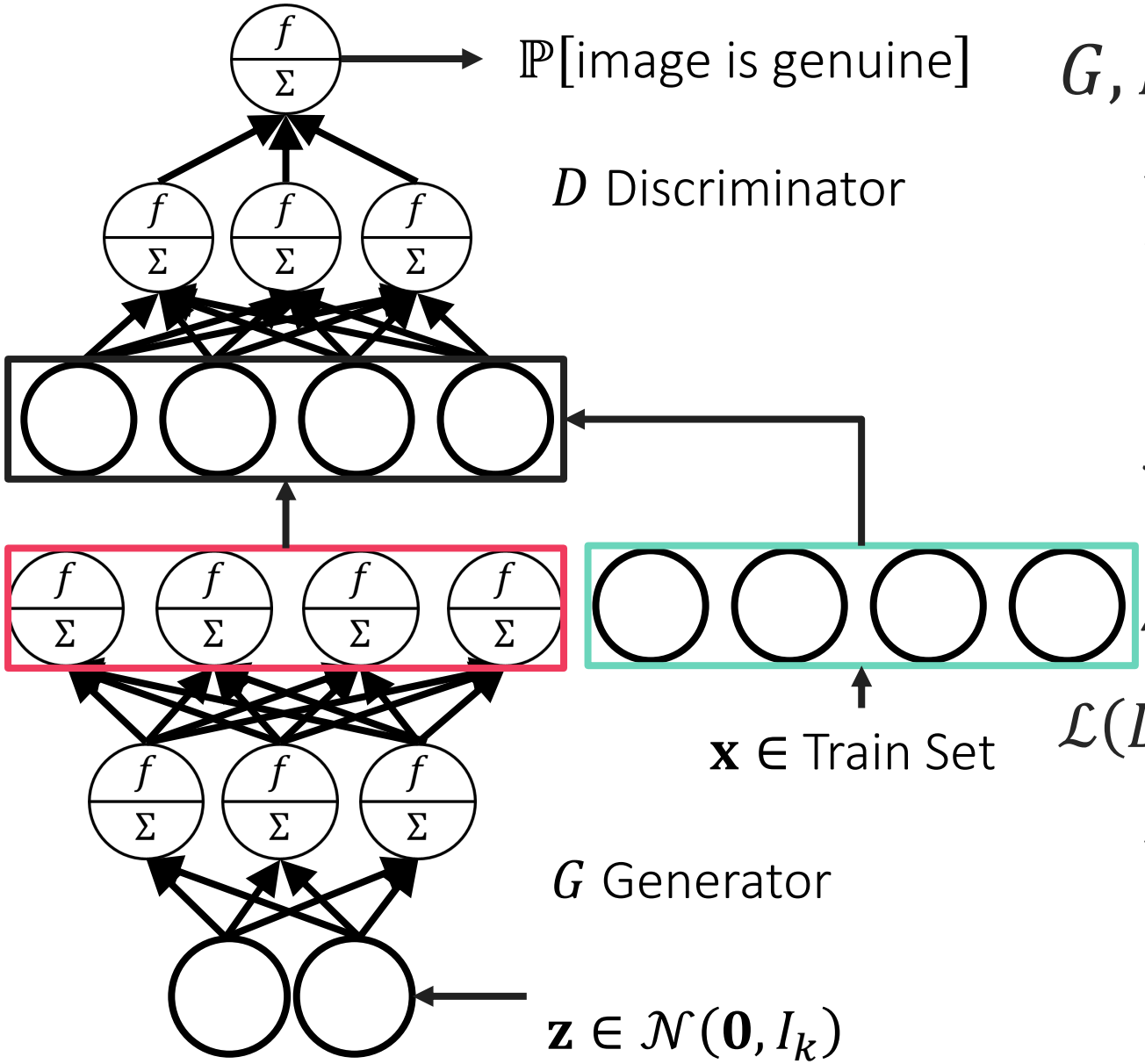
Sample $\mathbf{z}^{\mathbf{new}} \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$ and use decoder $d(\mathbf{z}^{\mathbf{new}})$ to obtain a new data point

Generative Adversarial Networks

Do not aim to offer good reconstruction error – sole aim is to produce new data points that are realistic and distributed as training data points

- *E.g. given tons of cat images, generate new ones (as if there weren't enough of them around already) but make sure the new ones do look like cats*
- *Uses two networks to perform this – a generator which attempts to generate new samples, and a discriminator which attempts to distinguish the samples from actual training data*
- *If the discriminator is unable to tell the generator's samples from actual data then this means the generator has learnt to generate realistic data (or else that the discriminator is weak and needs to be strengthened)*
- *Interesting interpretation of GAN tells us that the discriminator supplies the generator with the gradients it needs to train itself*
- *Thus, instead of calculating backprop gradients from data, another NN (the discriminator) "learns" the gradients for the generator* ☺

GANs – a toy depiction



G, D play a game to defeat the other

D tries to predict whether data given to it is real train data or fake samples

Wants $\mathbb{P}_D[\mathbf{x}^i] \rightarrow 1$ and $\mathbb{P}_D[G(\mathbf{z}^i)] \rightarrow 0$

G wants to generate very realistic samples and thus confuse D

Wants $\mathbb{P}_D[G(\mathbf{z}^i)] \rightarrow 1$

A single loss function is used

$$\mathcal{L}(D,G) = \log\big(D(\mathbf{x})\big) + \log\Big(1 - D\big(G(\mathbf{z})\big)\Big)$$

D tries to maximize this loss function

G tries to minimize this loss function