Deep Learning II

CS771: Introduction to Machine Learning

Purushottam Kar

Announcements

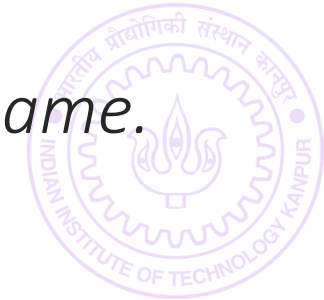
When submitting code for assignment 2 (and filling the Google form), please use the **latest group numbers** available at the following URL https://web.cse.iitk.ac.in/users/purushot/courses/ml/2019-20-a/material/assn_groups_16Oct2019.pdf

Your group number may have changed due to migrations/mergers

Please mention your **group number**, as well as names of group members, prominently on the **top of the first page** of the PDF file you submit to Gradescope.

This is to help us find out the group number when assigning coding question marks on Gradescope

For Assignment 1, graders had to perform a very tedious search by name. Please help them do their task more efficiently



Recap of Last Lecture

Neural networks offer a new way to learn non-linear models

- *Kernels implicitly create a large (infinite) number of features of which only few may be needed to solve problem whereas NN try to learn features themselves*
- *At the top, both kernel methods and NN usually learn a linear model*
- *Kernels do so over static features whereas NN learn model+features jointly*

Strictly speaking, NN are parameterized models

- *However, they are frequently* overparameterized – *billions of parameters*
- *Almost as good (bad) as non-parametric – bulky models, long train + test times*

NN contain layers – I/O layer (1 each) and 0 or more hidden layers

- *This layering is what makes deep learning "deep"*
- *Job of hidden layers is to learn good features, job of output layer is to learn a good linear model over those features, input layer simply supply input*

Recap of Last Lecture

Neural netv

Kernels im
may be ne

At the top

Kernels do

Strictly spea

However,

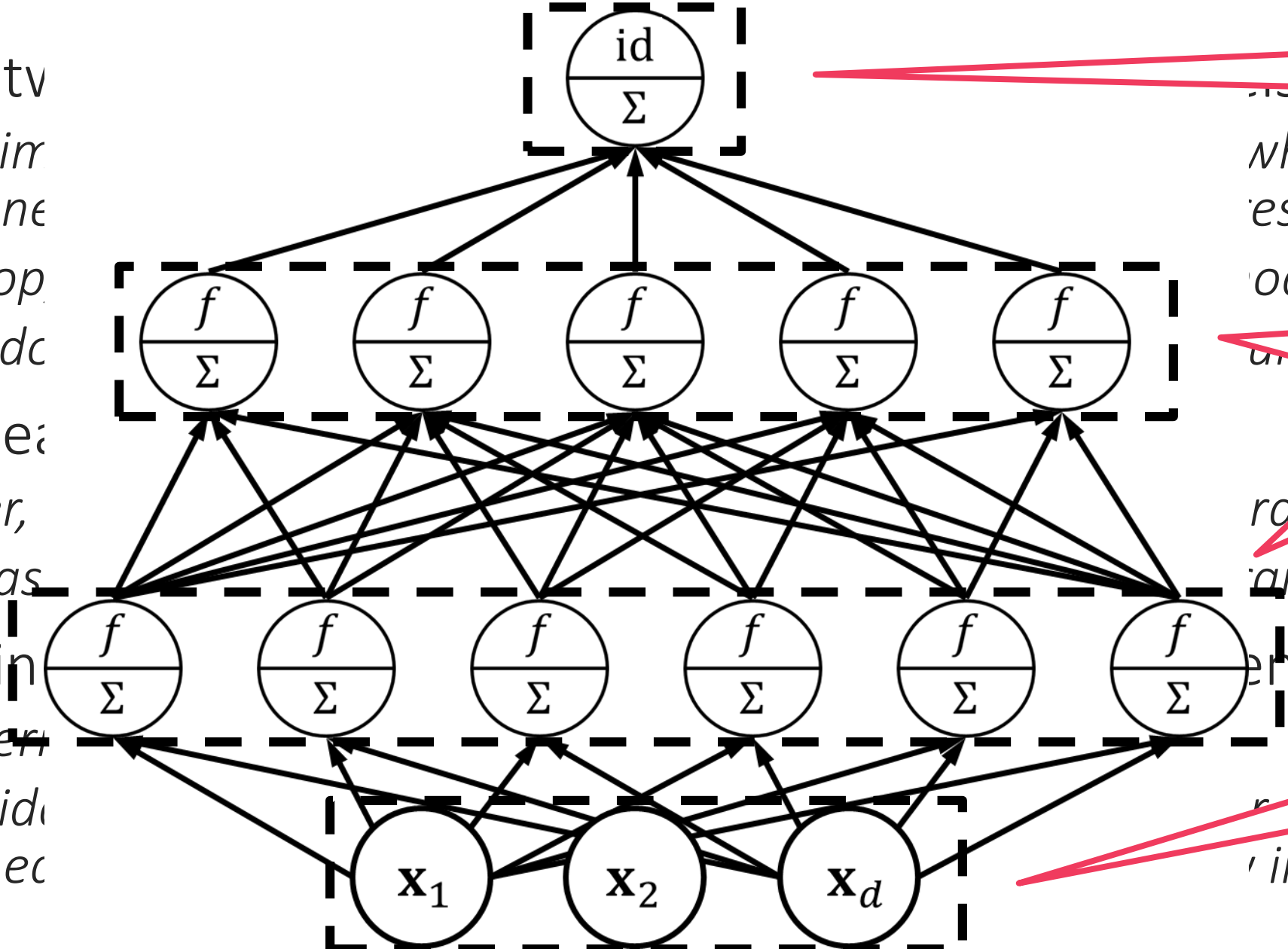
Almost as

NN contain

This layer

Job of hid
good linea

which only few
es themselves
ode
ra
ain + test times
en layers
input



Recap of Last Lecture

Neural netv

Kernels im
may be ne

At the to

Kernels c

Strictly spe

Howeve

Almost c

NN contai

This laye

Job of hid
good linec

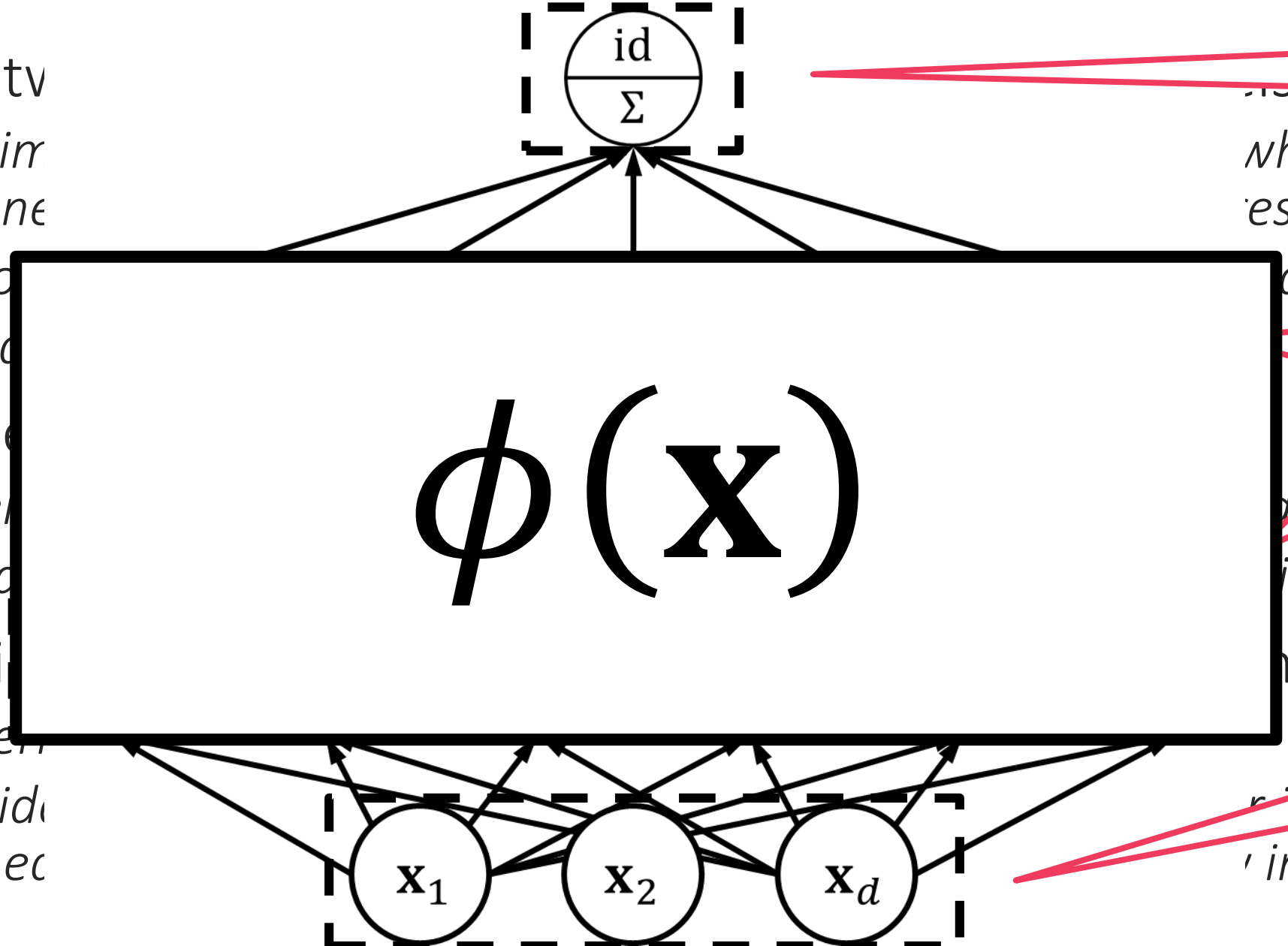
which only few
es themselves

de

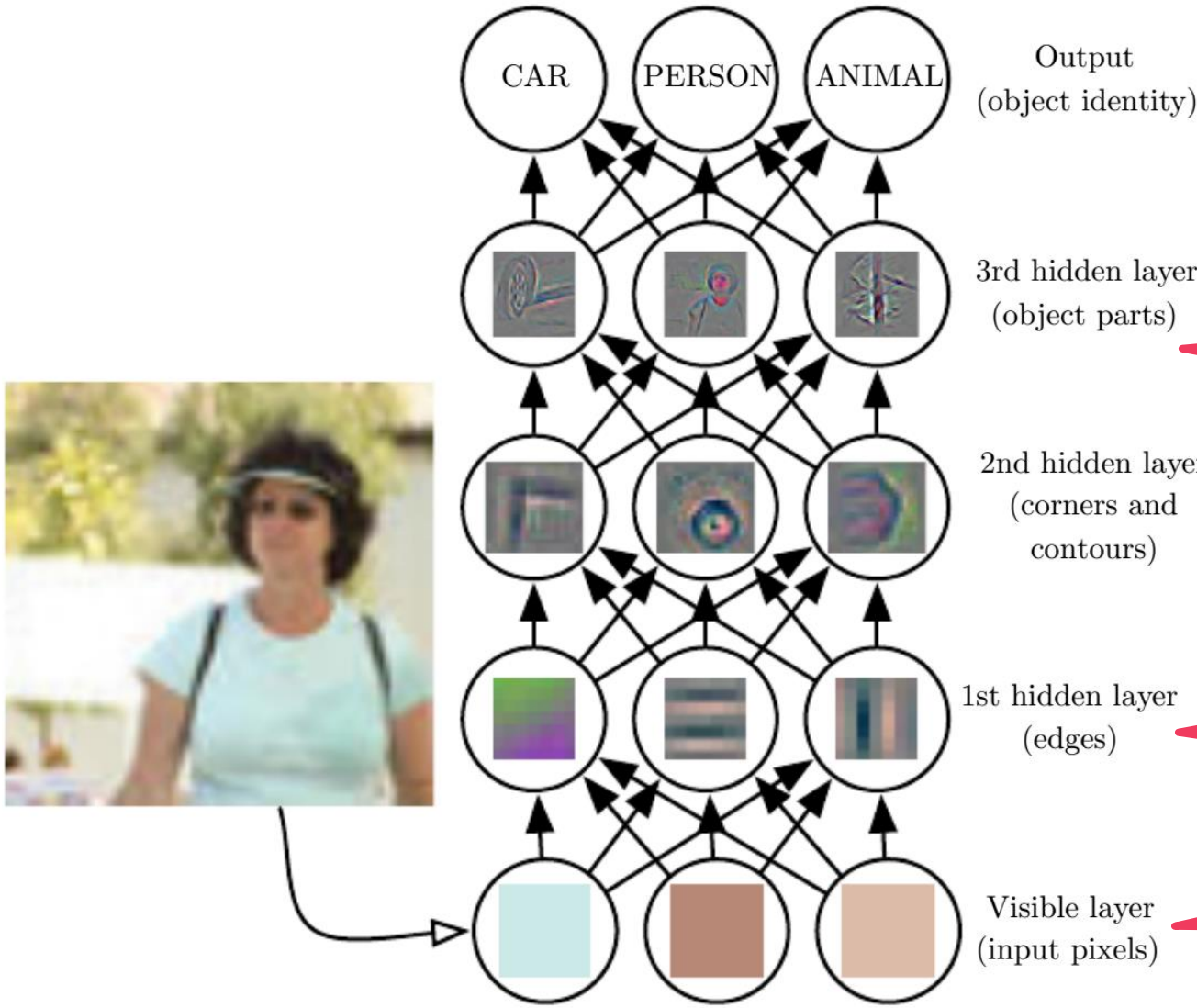
in + test times

layers

input



Deep Networks Learn Features



We know what are expected outputs of the o/p layer (they are the train labels). So we say that the o/p layer is "visible"

Hidden representations make it easier for a linear model to perform well

Hidden layers compute latent or "hidden" representations not supplied with train data

Input layer is fed with train features which are also "visible" in train data

FIGURE COURTESY ZEILER, M. D. AND FERGUS, R. ECCV 2014

Some General Comments

- Deep networks are universal i.e. all powerful
  - *Like universal kernels, a "big enough" network can learn any function*
    - Big enough can mean large number of "narrow layers" (i.e. each with few hidden nodes)
    - Big enough can mean just one impossibly wide hidden layer (shallow and wide)
- In practice, coming up with an appropriate architecture is challenging
  - *A lot of current progress in deep learning is due to people coming up with innovative architectures that do very well for various problems*
  - *No universal/well accepted rule for architecture design. Once an architecture is successful in a domain e.g. vision, people often reuse that same design*
  - *Often, even weights for lower layers reused – only top (few) layers retrained* ☺
- Training deep networks not very straightforward either
  - *Availability of GPUs speeds things up but several heuristics like dropout, batch normalization etc required to do very well on challenging problems*

Deep Learning or Wide Learning?

As stated before, NN are universal approximators when sufficiently big

(Cybenko 1989, Hornick 1991) show a single sufficiently wide layer (can need $\Omega(2^d)$ *nodes) can approx. "any" function i.e. 3 layer (depth 2) NN is universal*

Lu et al (2017) show that $\mathcal{O}(d)$ *layers each of* $\mathcal{O}(d)$ *width also suffice. However, the weights in the connections may have to become arbitrarily large*

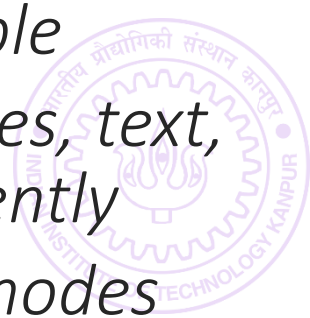
Large NN often overfit – simply memorize train data (just like RBF kernel)

Prevalent wisdom: large number of hidden nodes is the key to power of NN. Use this power with caution (regularization) to prevent overfitting

Domains where raw features do not have much structure e.g. RecSys, common to start with just one hidden layer and see if performance is acceptable

Domains where raw features are extremely well structured e.g. images, text, video, it is common to have several layers to learn new features patiently

Need to do a bit of trial and error to identify good number of layers, nodes



Toy "Proof" for Universality of Sigmoid NN

A single neuron in the hidden layer can learn features (scores) of the form

$$\sigma(\mathbf{w}^{\mathsf{T}}\mathbf{x} + b)$$

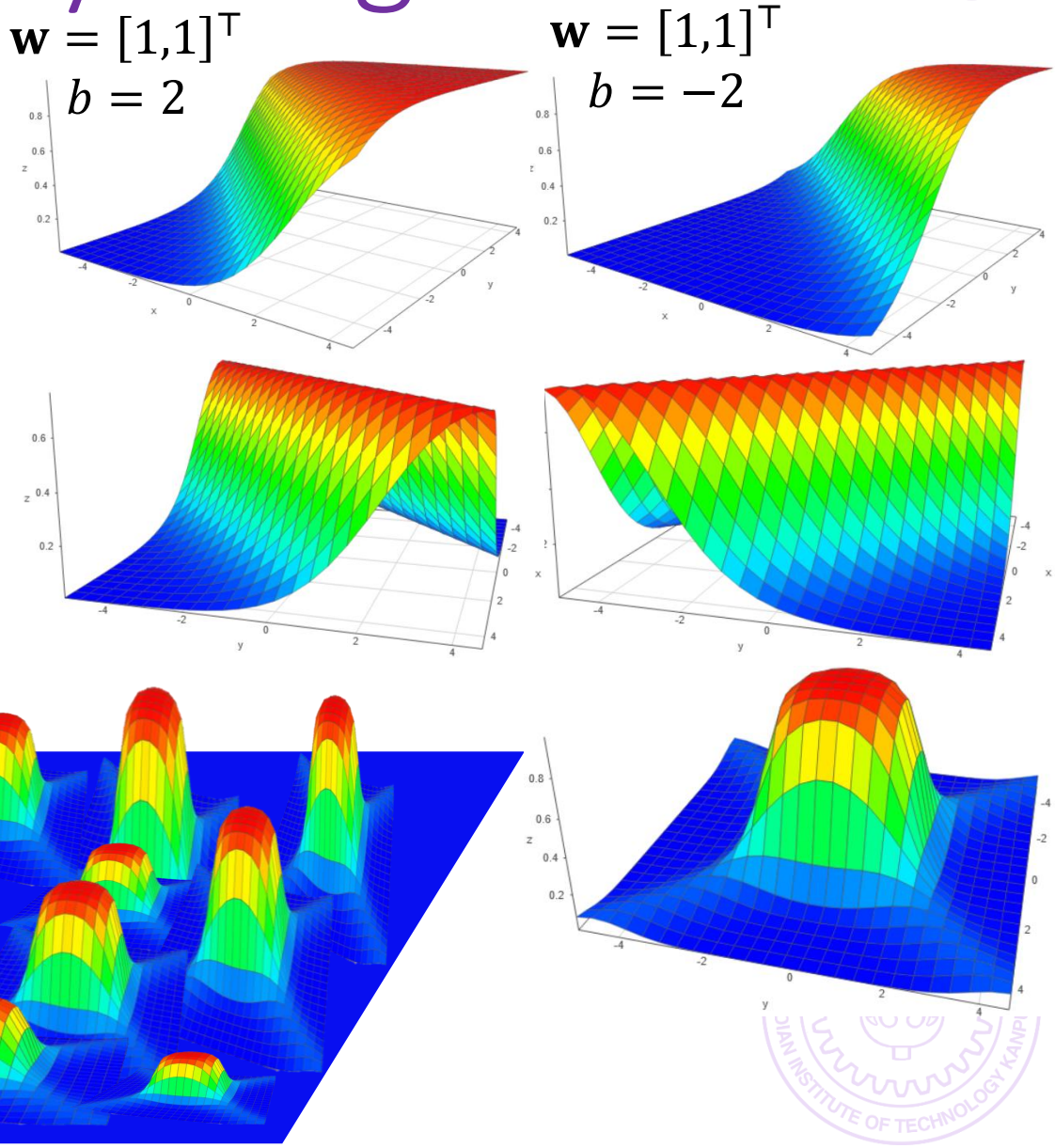
Combining two such features in second hidden or o/p layer gives features like

Combining two such features in third hidden or o/p layer gives features like

This allows us to place Dirac-delta like features anywhere, approximate any function we want, as closely we want

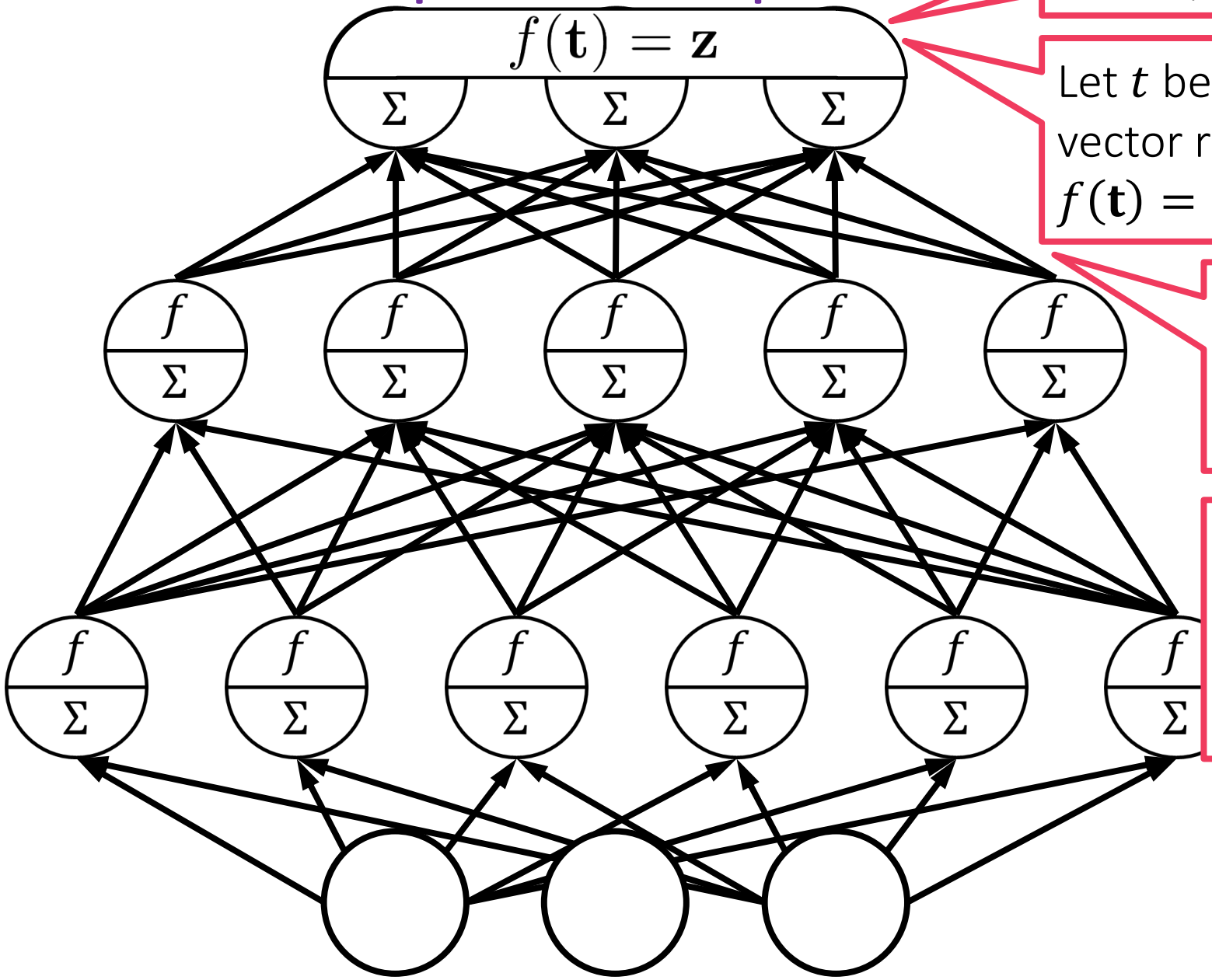
Not a proper proof though

Refer to papers for proofs



PLOTS COURTESY HTTPS://ACADEMO.ORG/DEMOS/3D-SURFACE-PLOTTER/

Multi-output Deep Netw



Useful for multi-class/label classfn, ranking, vector regression

Let t be pre-activations of o/p layer. For vector regression $f(\boldsymbol{t}) = \mathbf{t}$. For multilabel $f(\mathbf{t}) = \mathbf{sign}(\mathbf{t})$ (element-wise) often used

For multiclass, softmax popular.

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

Good to normalize before use $\tilde{\mathbf{t}}_i = \mathbf{t}_i - \max_j \mathbf{t}_j$ to avoid overflow problems. Note that Note that $\mathrm{f}_{\mathrm{SM}}(\mathbf{t}) = \mathrm{f}_{\mathrm{SM}}(\tilde{\mathbf{t}})$

Loss Functions for Deep Learning

Squared loss $\ell_{\mathrm{LS}}(\hat{y}, y) = (\hat{y} - y)^2$

Absolute difference $\ell_{\mathrm{ABS}}(\hat{y}, y) = |\hat{y} - y|$

Squared/absolute loss most common for regression problems

Work well when ReLU/identity activation used in the output layer

Sigmoid/tanh do not give good gradients

Negative log-likelihood loss $\ell_{\mathrm{NLL}}(\hat{\mathbf{y}}, y) = -\log(\hat{\mathbf{y}}_y)$ where $y \in [K]$

Also called categorical cross entropy (CCE) – useful for multiclass problems

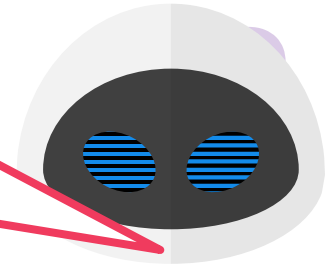
Binary CE $\ell_{\mathrm{CE}}(\hat{y}, y) = -y \cdot \log \hat{y} - (1 - y) \cdot \log(1 - \hat{y})$, $y \in \{0,1\}$

B/CCE popular for classification problems and most commonly used along with softmax activation for the output layer i.e. $\hat{\mathbf{y}} = f_{\mathrm{SM}}(\mathbf{t}), \mathbf{t} \in \mathbb{R}^K$

Hinge loss $\ell_{\mathrm{Hinge}}(\hat{y}, y) = [1 - y\hat{y}]_+$ where $y \in \{-1,1\}$

Loss Fu

Exact definitions, conventions may vary. Always check your library documentation (Keras, PyTorch etc) to avoid errors. E.g. Keras expects labels for even multiclass problems to be a $\{0,1\}$ vector



Squared loss $\ell_{\mathrm{LS}}(\hat{y}, y) = (\hat{y} - y)^2$

Absolute difference $\ell_{\mathrm{ABS}}(\hat{y}, y) = |\hat{y} - y|$

Squared/absolute loss most common for regression problems

Work well when ReLU/identity activation used in the output layer

Sigmoid/tanh do not give good gradients

Negative log-likelihood loss $\ell_{\mathrm{NLL}}(\hat{\mathbf{y}}, y) = -\log(\hat{\mathbf{y}}_y)$ where $y \in [K]$

Also called categorical cross entropy (CCE) – useful for multiclass problems

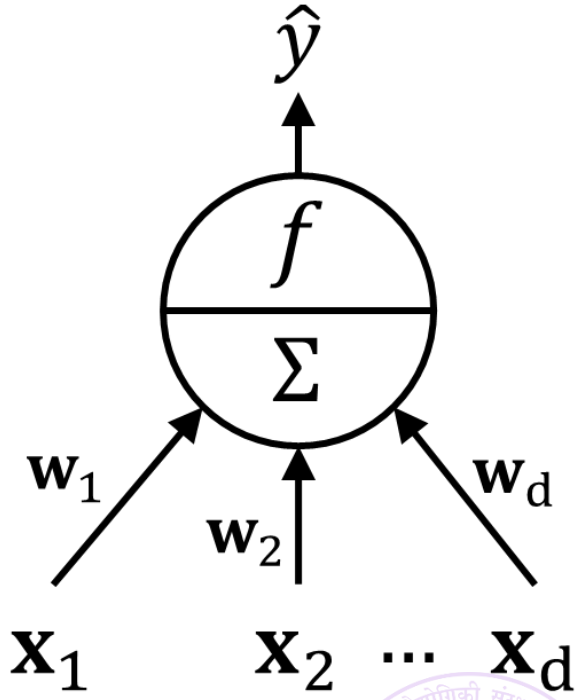
Binary CE $\ell_{\mathrm{CE}}(\hat{y}, y) = -y \cdot \log \hat{y} - (1 - y) \cdot \log(1 - \hat{y})$, $y \in \{0,1\}$

B/CCE popular for classification problems and most commonly used along with softmax activation for the output layer i.e. $\hat{\mathbf{y}} = f_{\mathrm{SM}}(\mathbf{t}), \mathbf{t} \in \mathbb{R}^K$

Hinge loss $\ell_{\mathrm{Hinge}}(\hat{y}, y) = [1 - y\hat{y}]_+$ where $y \in \{-1,1\}$

Training a Perceptron

Recall that a perceptron is simply an activation function wrapped around a linear model i.e. $\hat{y} = f(\langle \mathbf{w}, \mathbf{x} \rangle)$ i.e. a gen-lin model



Training a Perceptron

Recall that a perceptron is simply an activation function wrapped around a linear model i.e. $\hat{y} = f(\langle \mathbf{w}, \mathbf{x} \rangle)$ i.e. a gen-lin model

Given $(\mathbf{x}^1, y^1), \ldots, (\mathbf{x}^n, y^n), \mathbf{x}^i \in \mathbb{R}^d$ and loss fn ℓ: $\mathbb{R} \times \mathbb{R} \to \mathbb{R}_+$, we can train a perceptron by solving the following optimization problem

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

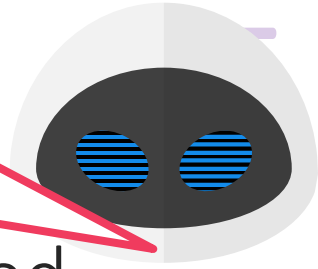
The training procedure is plain old gradient descent

However, this being a non-convex problem (due to f), requires more care

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

Training

Some texts may state that a perceptron is simply a linear model i.e. it cannot have a wrapper function around the linear model. We are using a broader definition of perceptron in our discussion



Recall that a perceptron is simply an activation function wrapped around a linear model i.e. $\hat{y} = f(\langle \mathbf{w}, \mathbf{x} \rangle)$ i.e. a gen-lin model

Given $(\mathbf{x}^1, y^1$ $\mathbb{R} \to \mathbb{R}_+$, we can train a pe tion problem

$\arg$ $(\mathbf{w})$

The training p

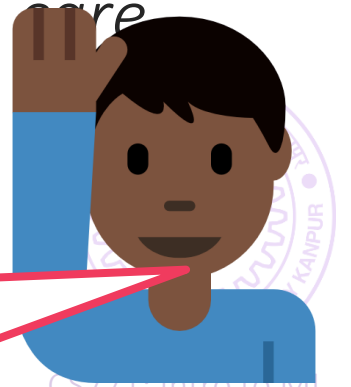
However, th *s more gre*

How to find a descent direction?

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 or until tired

Note that our definition of a perceptron is effectively equal to a NN with no hidden layers i.e. depth 1 (not considered "deep" – only depth ≥ 2 is deep). Let us see how to train a perceptron before seeing how deep NN are trained



Choosing a Descent Direction

Batch gradient (use chain rule)

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

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

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

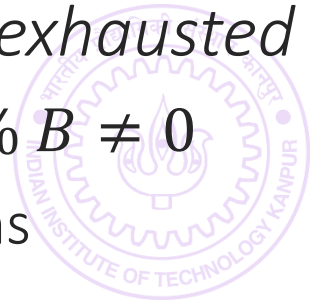
DL techniques usually decide batch size based on memory available

Popular choice – randPerm (similar to the strategy we used for SDCM)

Choose a random permutation σ *of* $[n]$*. First batch is* $\sigma(1), \ldots, \sigma(B)$*, second batch is* $\sigma(B+1), \ldots, \sigma(2B)$*, repeat for* $\lceil n/B \rceil$ *batches till all points exhausted*

This is called one **epoch**. *Note*: last batch may contain less than B points if $n \,\%\, B \neq 0$

For next epoch, choose a new random permutation and repeat for blah epochs



How to detect convergence

Method 1: Tolerance technique

For a pre-decided tolerance value ϵ, if $F(\mathbf{w}^t) < \epsilon$, stop

Method 2: Zero-th order technique

If fn value has not changed for many epochs, stop (or else tune learning rate)!

$$\left|F\left(\mathbf{w}_{\text{epoch}}^{t+1}\right) - F\left(\mathbf{w}_{\text{epoch}}^{t}\right)\right| < \tau$$

Method 3: First order technique

If gradient has become too "small" $\|\nabla F(\mathbf{w}^t)\|_2 < \delta$, stop!

Method 4: Cross validation technique

Test the current model on validation data – if performance acceptable, stop!

Other techniques e.g. primal-dual techniques are usually infeasible for DL problems and hence not used to decide convergence

How to decide step length?

Simple choices e.g. $\eta_t = C/\sqrt{t}$ or $\eta_t = C/t$ for $C > 0$ (simple to try)

Basic idea is to choose $\eta_t \to 0$ (diminishing) and $\sum \eta_t \to \infty$ (infinite travel)

Line search e.g. $\eta_t = \arg\min_{\eta \geq 0} F(\mathbf{w}^t - \eta \cdot \mathbf{g}^t)$ become too expensive

Adaptive *momentum*-based methods are more popular for DL

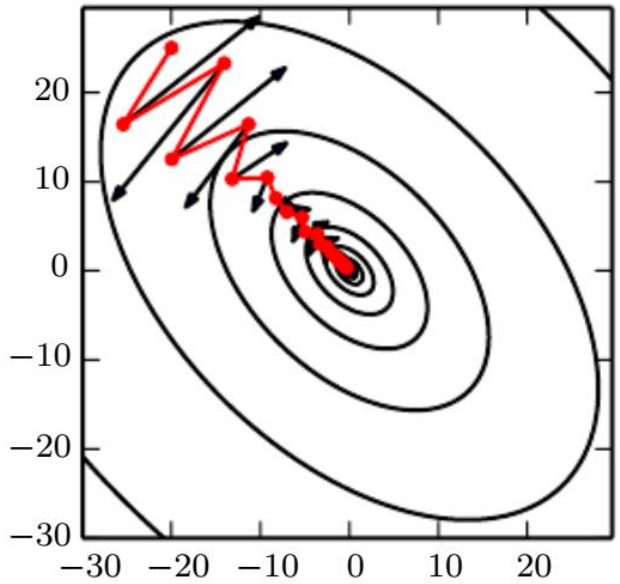
Introduce a "velocity" term to push GD along, avoid oscillations

$$\mathbf{v}^t = \gamma \cdot \mathbf{v}^{t-1} + \eta \cdot \nabla F(\mathbf{w}^t)$$
$$\mathbf{w}^{t+1} \leftarrow \mathbf{w}^t - \mathbf{v}^t$$

GD experiences oscillations (see figure) even in a simple least squares problem that waste time and slow progress

***Nesterov's accelerated gradient** (NAG): pioneer in the area*

For differentiable convex problems, NAG ensures ϵ-convergence in just $\mathcal{O}(1/\epsilon)$ steps hence "accelerated"



Adaptive Learning Rates

Take inspiration from *Newton method,* a sort-of autotuning GD variant

$\mathbf{g}^t = \left(\nabla^2 F(\mathbf{w}^t)\right)^{-1} \nabla F(\mathbf{w}^t)$ - *costly but offers fast convergence*

***Key Idea**: replace $\nabla^2 F(\mathbf{w}^t)$ with a diagonal matrix H^t – much cheaper*

Adagrad (Duchi et al. 2011) – use past updates to calculate H^t

$H^t = \text{diag}(h_1^t, \ldots, h_d^t)$ *where* $h_i^t = \sqrt{\epsilon + \sum_{\tau=1}^{t} \left(\mathbf{g}_i^\tau\right)^2}$

If a coordinate got very vigorous updates in the past $|g_i^\tau| \gg 0$, mellow its future updates

Reduces to $\eta_t \approx \eta/t$ if all coordinates got roughly similar gradients in the past since then we would have had $h_i^t \approx \mathcal{O}(t)$ for all $i \in [d]$

If some coordinate is static, not getting updated at all i.e. $\mathbf{g}_i^\tau \equiv 0$ for all τ, then $h_i^t = \sqrt{\epsilon}$ to prevent a divide-by-zero error when we take inverse of H^t

Adaptive Learning Rates

RMSProp (Hinton 2012) – apply momentum idea to Adagrad

$$h_i^t = h_i^t = \sqrt{\epsilon + v_i^t} \text{ where } v_i^t = \gamma \cdot v_i^{t-1} + (1-\gamma) \cdot \left(\mathbf{g}_i^t\right)^2$$

Adagrad sometimes forces step sizes to go down too much – this avoids that

Adam (Kingma and Ba 2014) – combine NAG and RMS-Prop

$$\mathbf{g}^t = (H^t)^{-1}\mathbf{u}^t \text{ where } H^t = \text{diag}(h_1^t, \ldots, h_d^t),\ h_i^t = \sqrt{v_i^t} + \epsilon, \text{ and}$$

$$\mathbf{u}^t = \gamma_1 \cdot \mathbf{u}^{t-1} + (1-\gamma_1) \cdot \mathbf{g}^t$$

$$v_i^t = \gamma_2 \cdot v_i^{t-1} + (1-\gamma_2) \cdot \left(\mathbf{g}_i^t\right)^2$$

Also does some bias corrections (reweighting) on H^t and $\mathbf{u}^t$ (not shown above)

All these methods are implemented and readily available in libraries

How to prevent overfitting?

Method 1: Regularize the weights

1.1: add an explicit regularization term $\arg\min_{\mathbf{w}\in\mathbb{R}^d} F(\mathbf{w}) + \lambda \cdot \|\mathbf{w}\|_2^2$

1.2: don't allow any weight to get big (clip them) $\arg\min_{\|\mathbf{w}\|_\infty < r} F(\mathbf{w})$

Most libraries implement these strategies themselves e.g. weight clipping

Sometimes even gradient coordinates are clipped to stabilize training

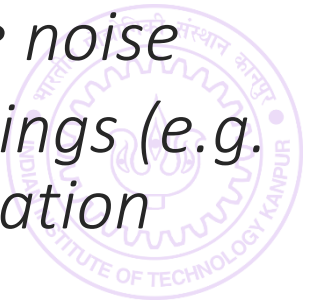
Method 2: Deliberately inject noise into the labels

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* $\epsilon^i \sim \mathcal{N}(0, \sigma^2)$

Unlikely that a NN with limited # of nodes would be able to memorize noise

For NN setting these are just heuristics but when applied to nicer settings (e.g. linear models), label noise can be shown to be equivalent to regularization



How to prevent overfitting?

Method 3: Learn Sparse Models

3.1 *Learn a NN that has sparse (instead of dense) connections b/w layers*

See Frankle-Carbin (ICLR 2019) – The Lottery Ticket Hypothesis

3.2 Parameter sharing*: force some of the weights to take the same value by adding constraints of the form* $\mathbf{w}_i = \mathbf{w}_j$

Convolutional NN do this implicitly and are very successful

3.3 Dropout*: Implicitly trains on multiple sparse networks in parallel*

While executing a GD update, randomly remove edges or entire nodes from network so they do not get updated in that iteration. Put them back in after update is over

Can again be shown to be equivalent to L_2 regularization in "nice" settings

Method 4: Validation

Use early stopping – do not rely on training loss but rather performance on a held-out (or k-fold) validation set to decide when to stop training

Dropout

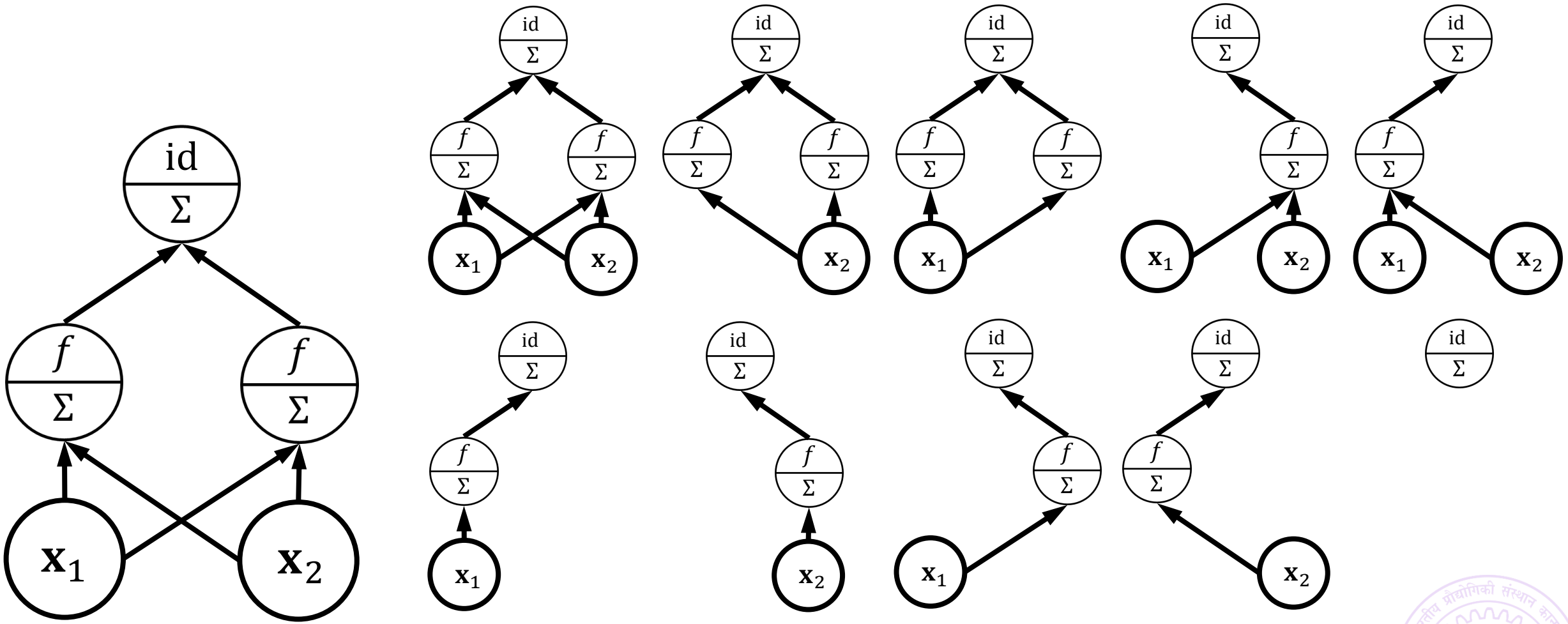
During training, before applying mini-batch gradient descent

- *Randomly mark each input node (e.g. choose each with prob 20%)*
- *Randomly mark each hidden node (e.g. choose each with prob 50%)*
- *Remove marked nodes, and corresponding edges from the network*
- *Apply mini-batch gradient descent (or backprop) to the remaining network*
  - Backprop is GD applied to multilayer perceptrons – will study this next
- *NAG, AdaGrad, Adam etc can be used along with dropout as usual*
- ***Dropout at test time**: scale the (post-activation) output of each node in the NN with the prob with which that node would have been spared from marking*
  - An approximation but a scalable one that gives acceptable performance in practice
  - $\mathbb{E}[h] = h \cdot \mathbb{P}[\text{not drop}] + 0 \cdot \mathbb{P}[\text{drop}] = h \cdot (1-p)$ ($p = 0.2/0.5$ for ip/hidden nodes)

Forces nodes to learn to work in absence of other nodes – robust!

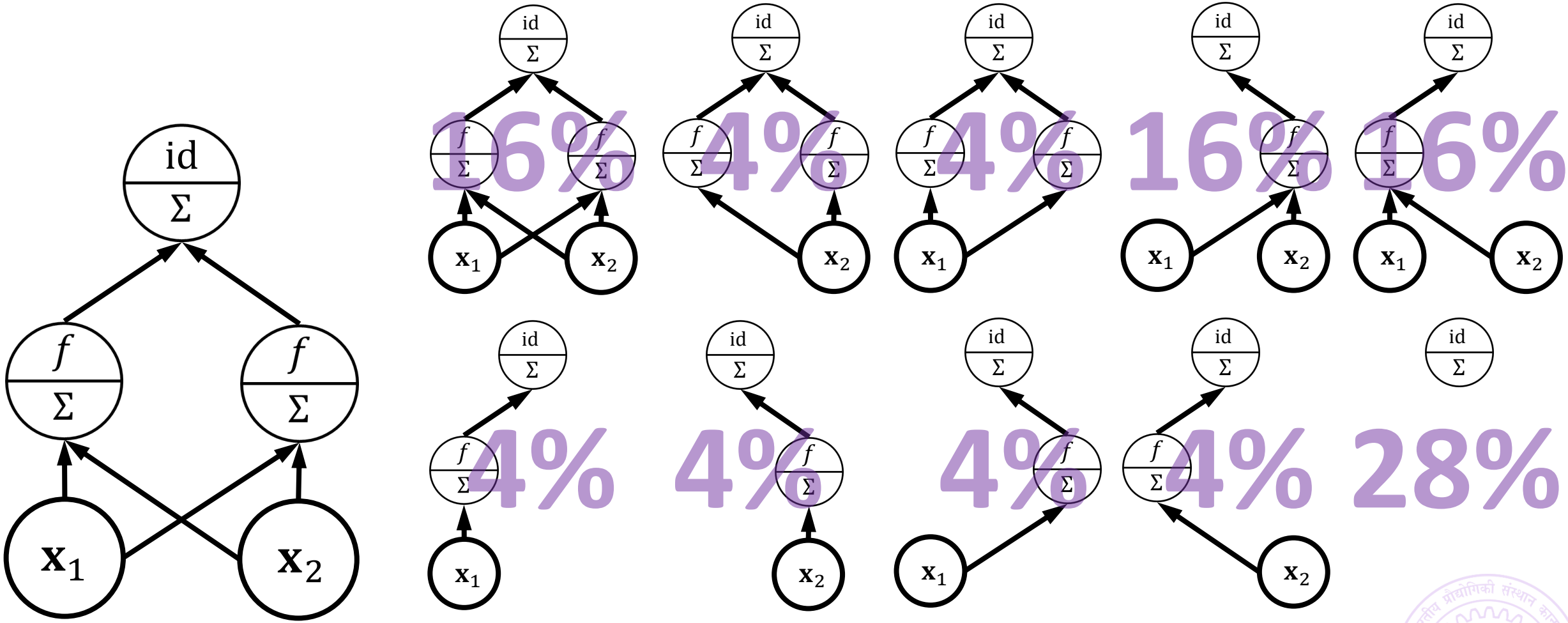
- *Side effect is slightly faster training and regularization*

Dropout at Work



In this toy example, 28% NN have no i/p nodes or no path connecting at least one i/p node to the o/p node i.e. cannot apply GD to them. For large networks, it is unlikely that sampling will result in such a disconnected network. In practice a large fraction of nodes do get retained

Dropout at Work



In this toy example, 28% NN have no i/p nodes or no path connecting at least one i/p node to the o/p node i.e. cannot apply GD to them. For large networks, it is unlikely that sampling will result in such a disconnected network. In practice a large fraction of nodes do get retained