Non-linear Models-V

CS771: Introduction to Machine Learning

Purushottam Kar

Outline of discussion

- More on architecture of neural networks
- Training methods for feedforward networks

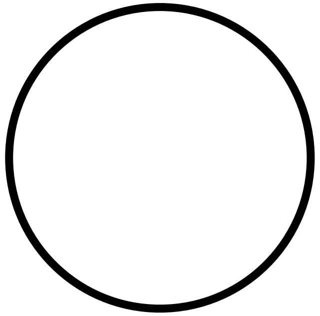
Answering the Fan Mail

- How does the notion of similarity (kernels) over notion of distance?
  - Two sides of the same coin
  - Kernels have nice math properties that algorithms exploit
- Please relate toy examples in Lec 17 to accelerated kernel methods
  - Better idea: download data from https://goo.gl/JXEQjr
  - Create feature maps for Gaussian kernel https://goo.gl/hBsX1E
  - See the magic happen!

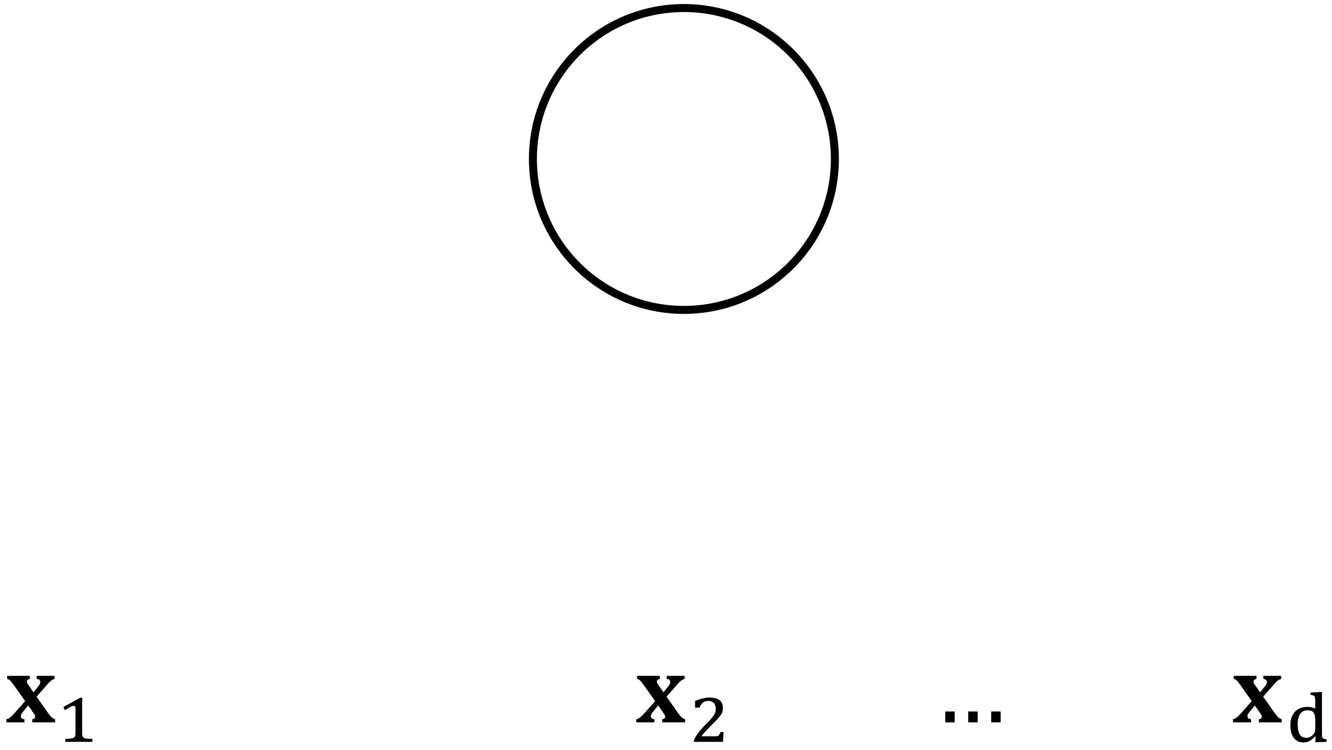
Neural Networks

The “neuron” in Neural Networks

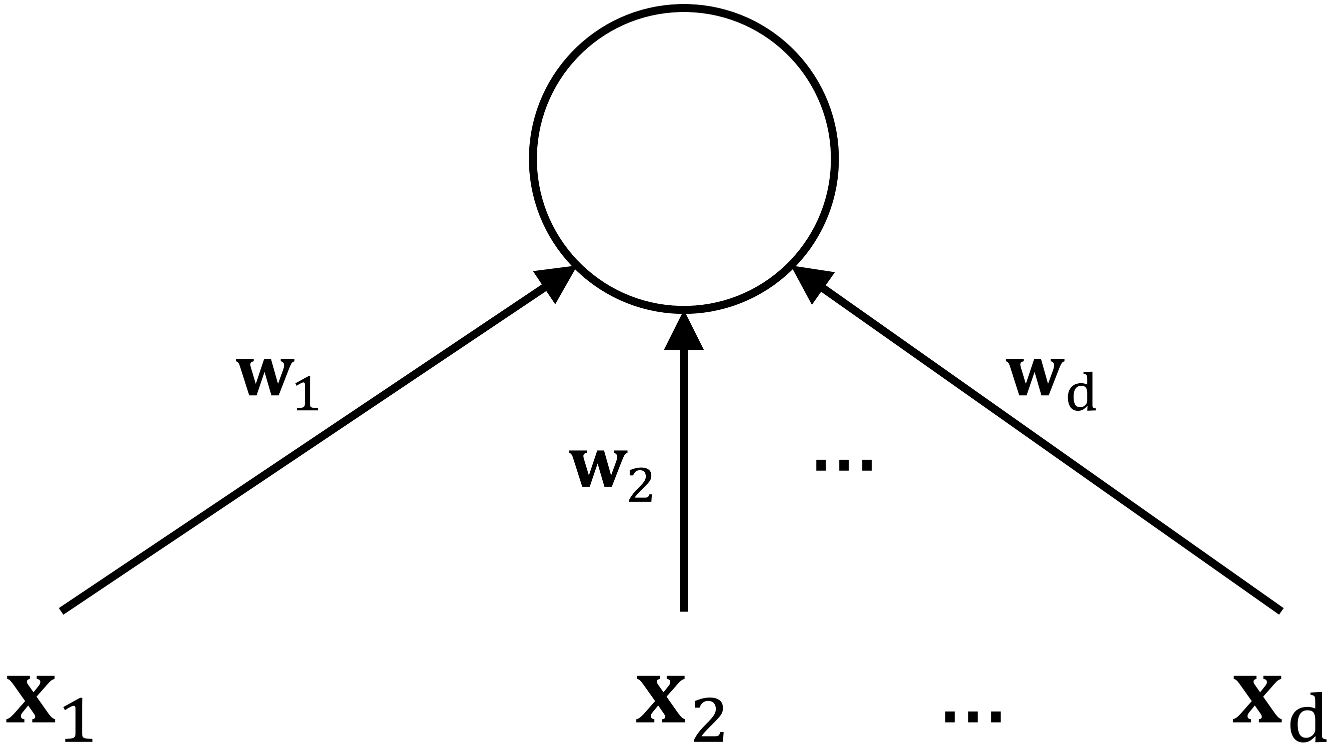
The "neuron" in Neural Networks



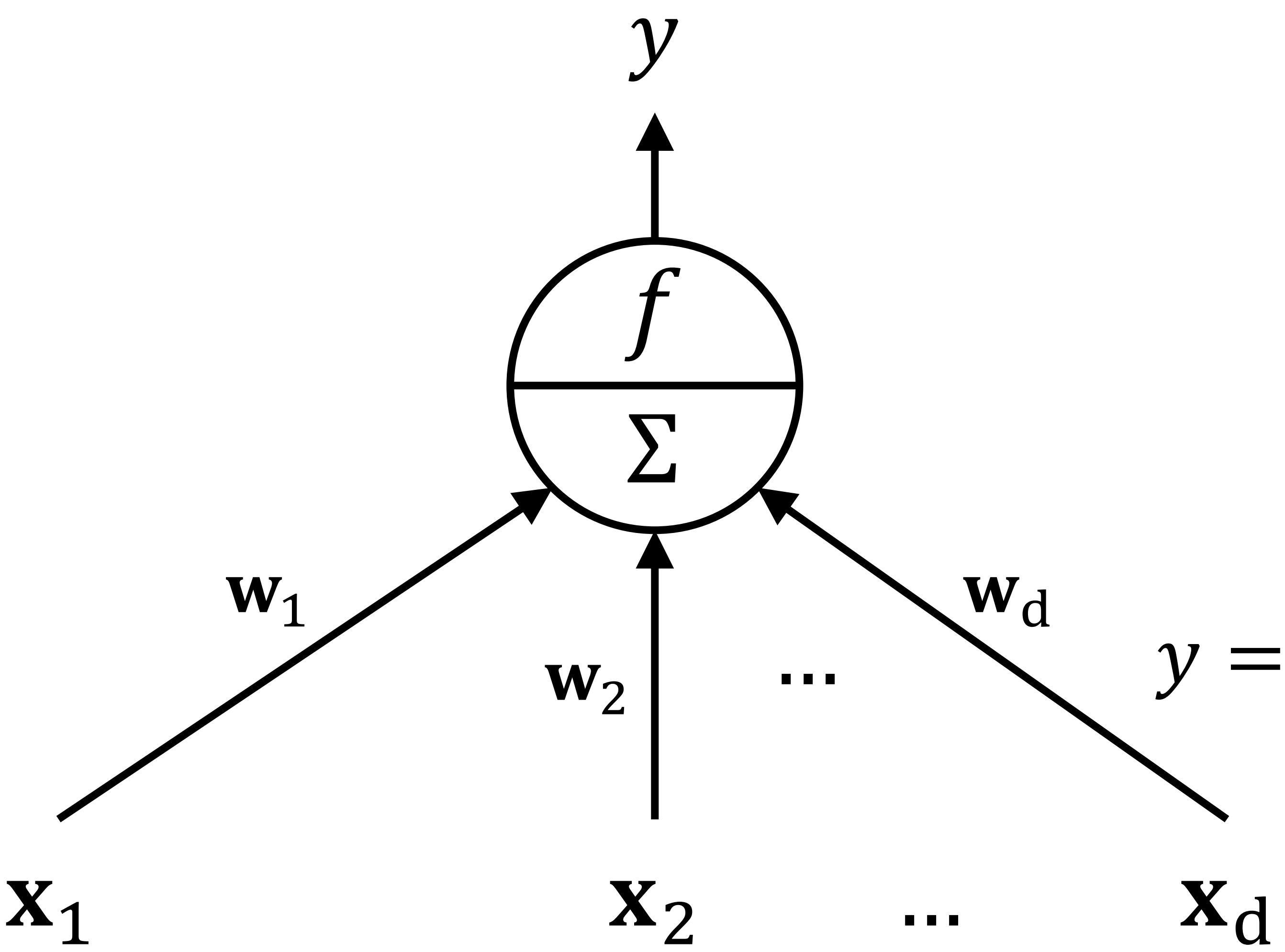
The “neuron” in Neural Networks



The "neuron" in Neural Networks

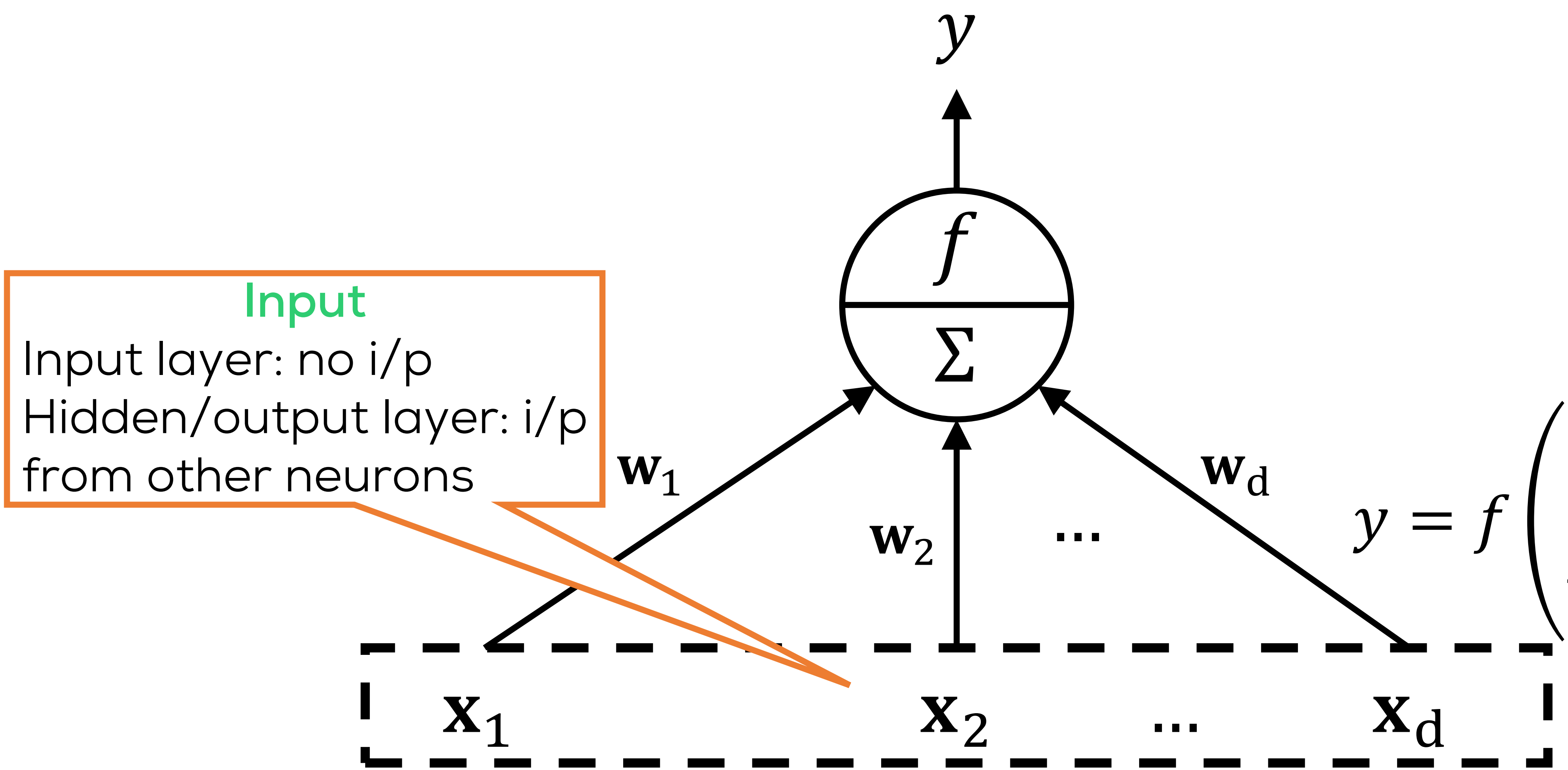


The "neuron" in Neural Networks



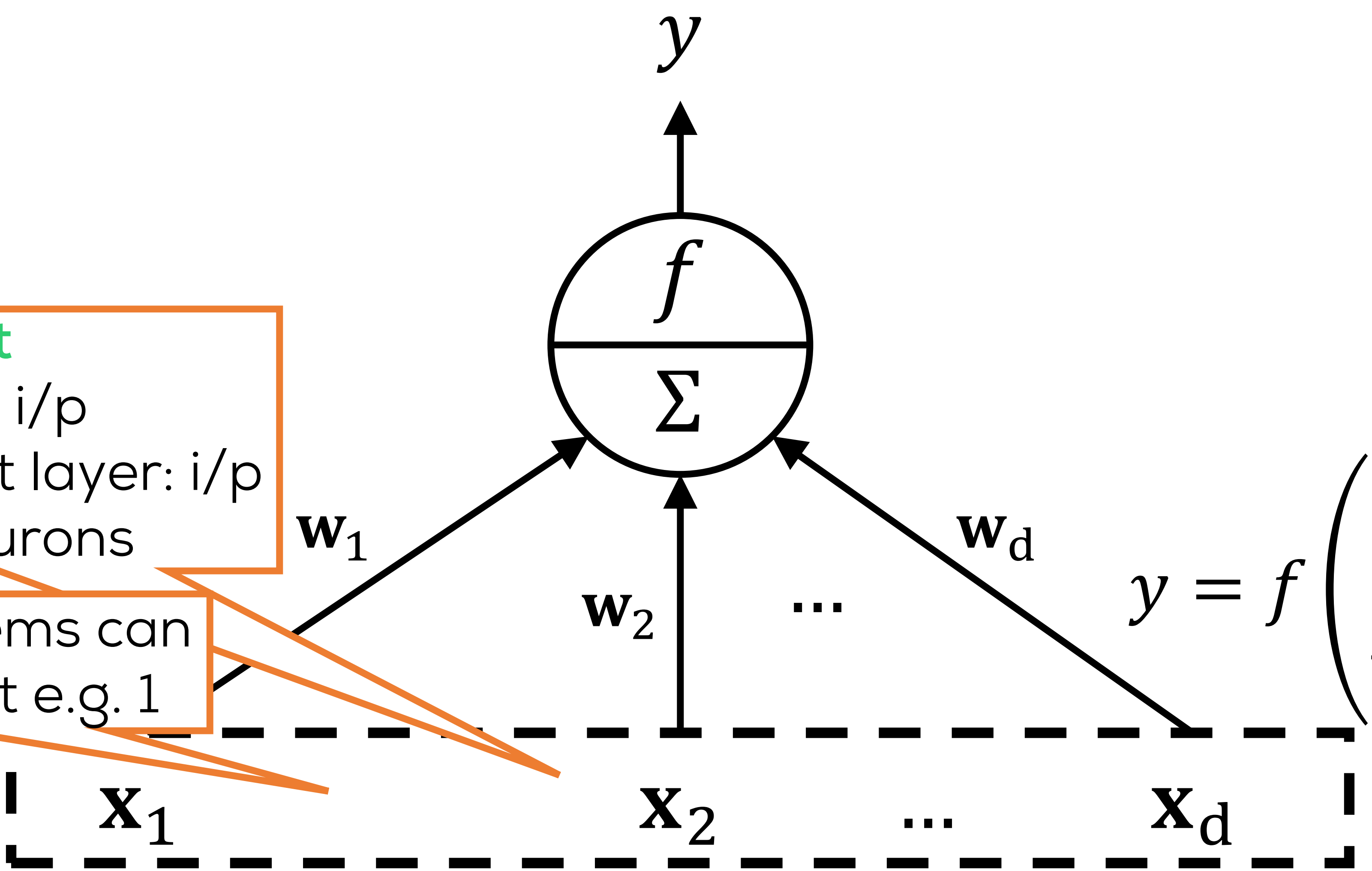
$$y = f\left(\sum_{i=1}^{d} \mathbf{w}_i \cdot \mathbf{x}_i\right)$$

The "neuron" in Neural Networks



$$y = f\left(\sum_{i=1}^{d} \mathbf{w}_i \cdot \mathbf{x}_i\right)$$

The "neuron" in Neural Networks



Input
Input layer: no i/p
Hidden/output layer: i/p from other neurons

Some input items can be a constant e.g. 1

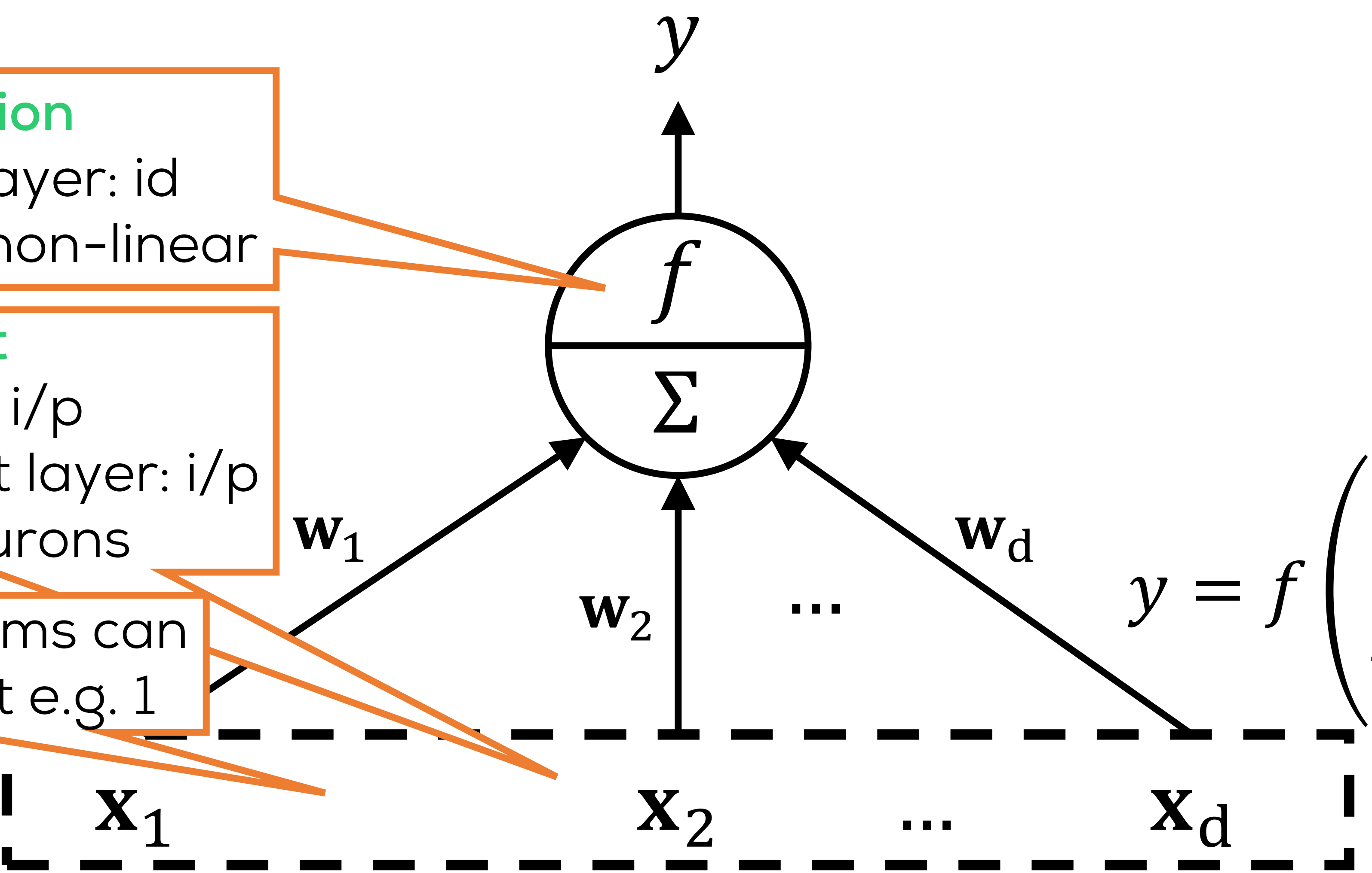
$$y = f\left(\sum_{i=1}^{d} \mathbf{w}_i \cdot \mathbf{x}_i\right)$$

The "neuron" in Neural Networks

Activation
Input/output layer: id
Hidden layer: non-linear

Input
Input layer: no i/p
Hidden/output layer: i/p from other neurons

Some input items can be a constant e.g. 1



$$y = f\left(\sum_{i=1}^{d} \mathbf{w}_i \cdot \mathbf{x}_i\right)$$

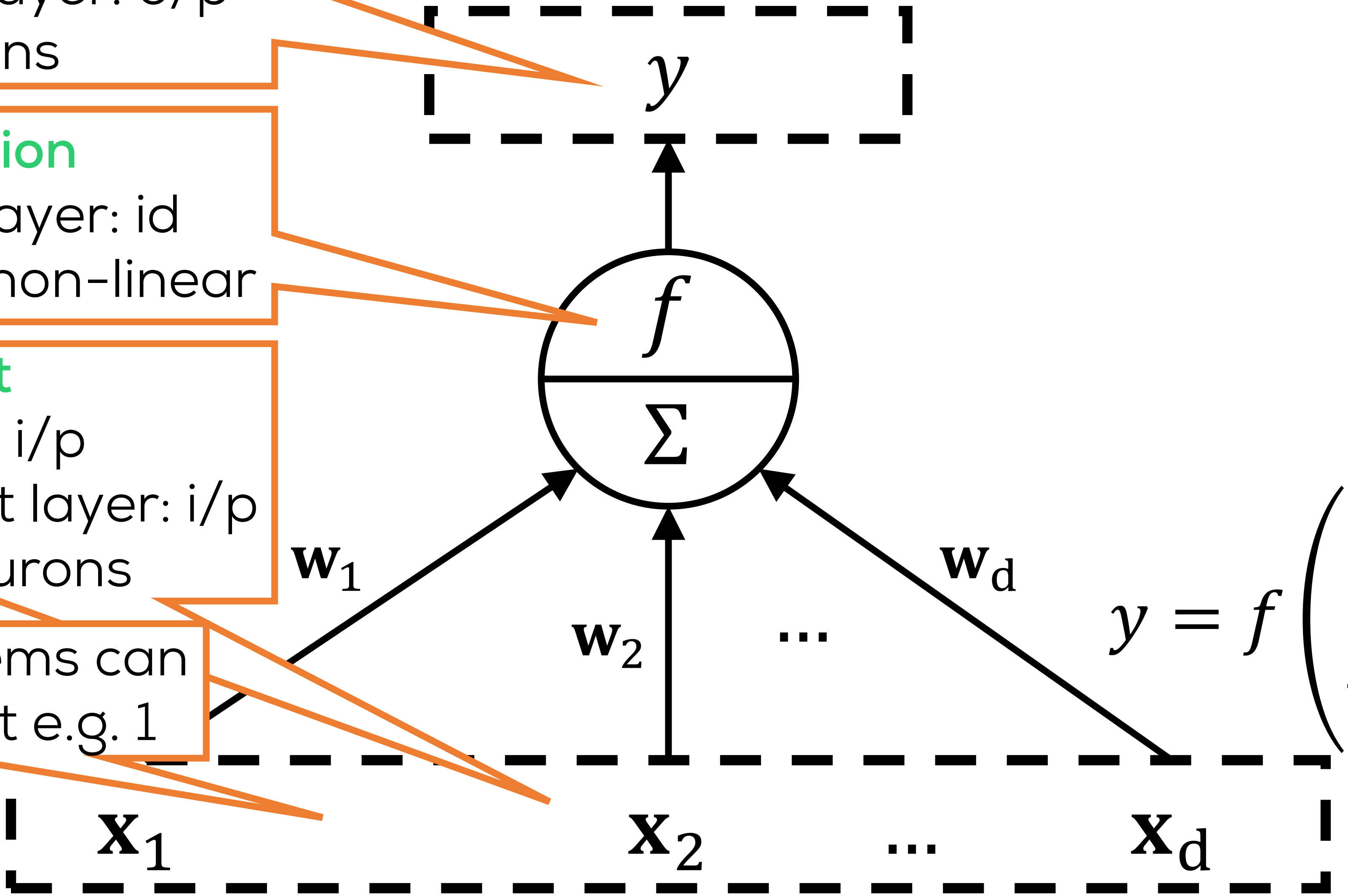
n Neural Networks

Output
Output layer: final o/p
Input/hidden layer: o/p to other neurons

Activation
Input/output layer: id
Hidden layer: non-linear

Input
Input layer: no i/p
Hidden/output layer: i/p from other neurons

Some input items can be a constant e.g. 1



$$y = f\left(\sum_{i=1}^{d} \mathbf{w}_i \cdot \mathbf{x}_i\right)$$

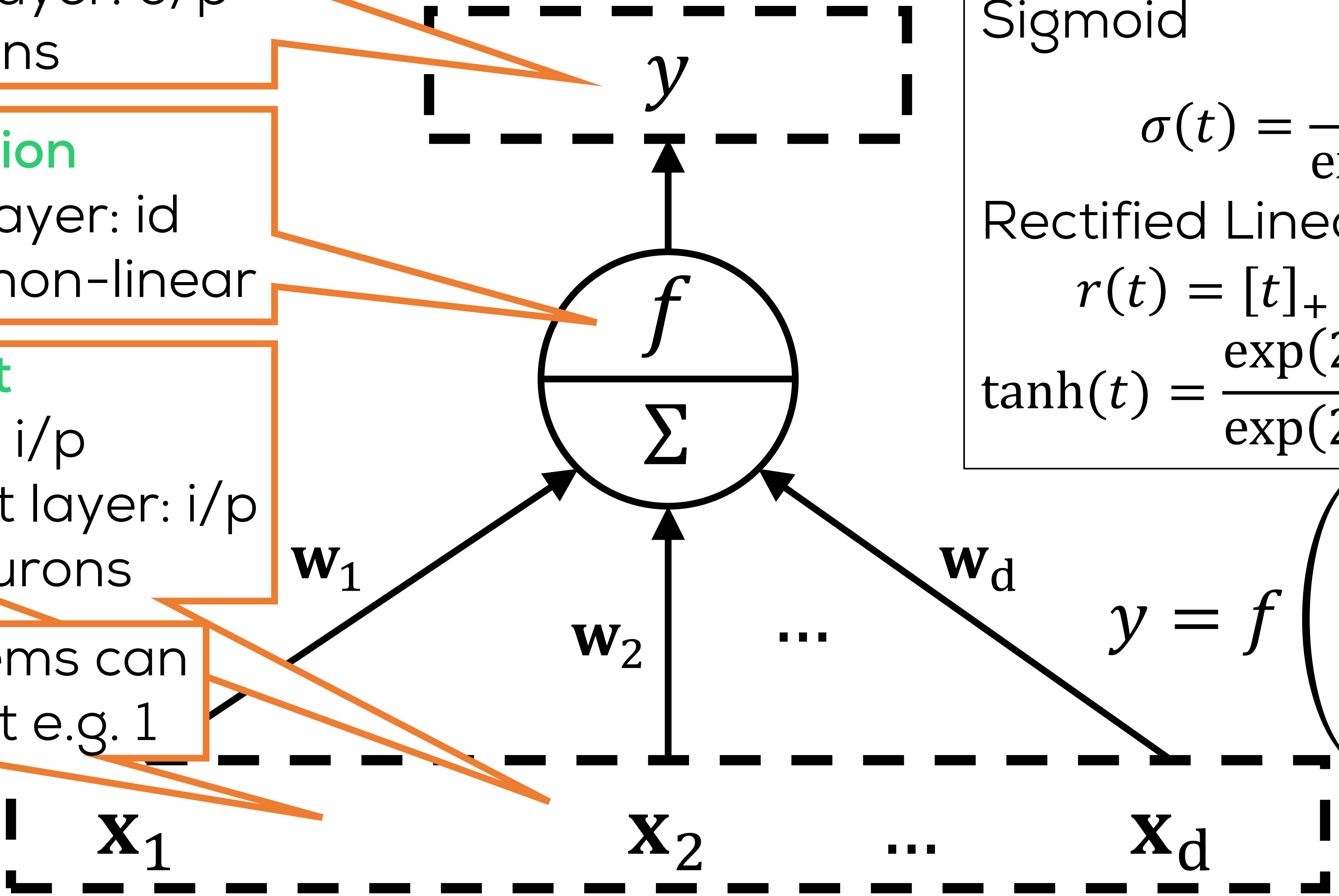
… in Neural Networks

Output
Output layer: final o/p
Input/hidden layer: o/p to other neurons

Activation
Input/output layer: id
Hidden layer: non-linear

Input
Input layer: no i/p
Hidden/output layer: i/p from other neurons

Some input items can be a constant e.g. 1



Common “activation” fns f

Sigmoid

$$\sigma(t) = \frac{\exp(t)}{\exp(t)+1}$$

Rectified Linear Unit (ReLU)

$$r(t) = [t]_+ = \max(t, 0)$$

$$\tanh(t) = \frac{\exp(2t)-1}{\exp(2t)+1}$$

$$y = f\left(\sum_{i=1}^{d} \mathbf{w}_i \cdot \mathbf{x}_i\right)$$

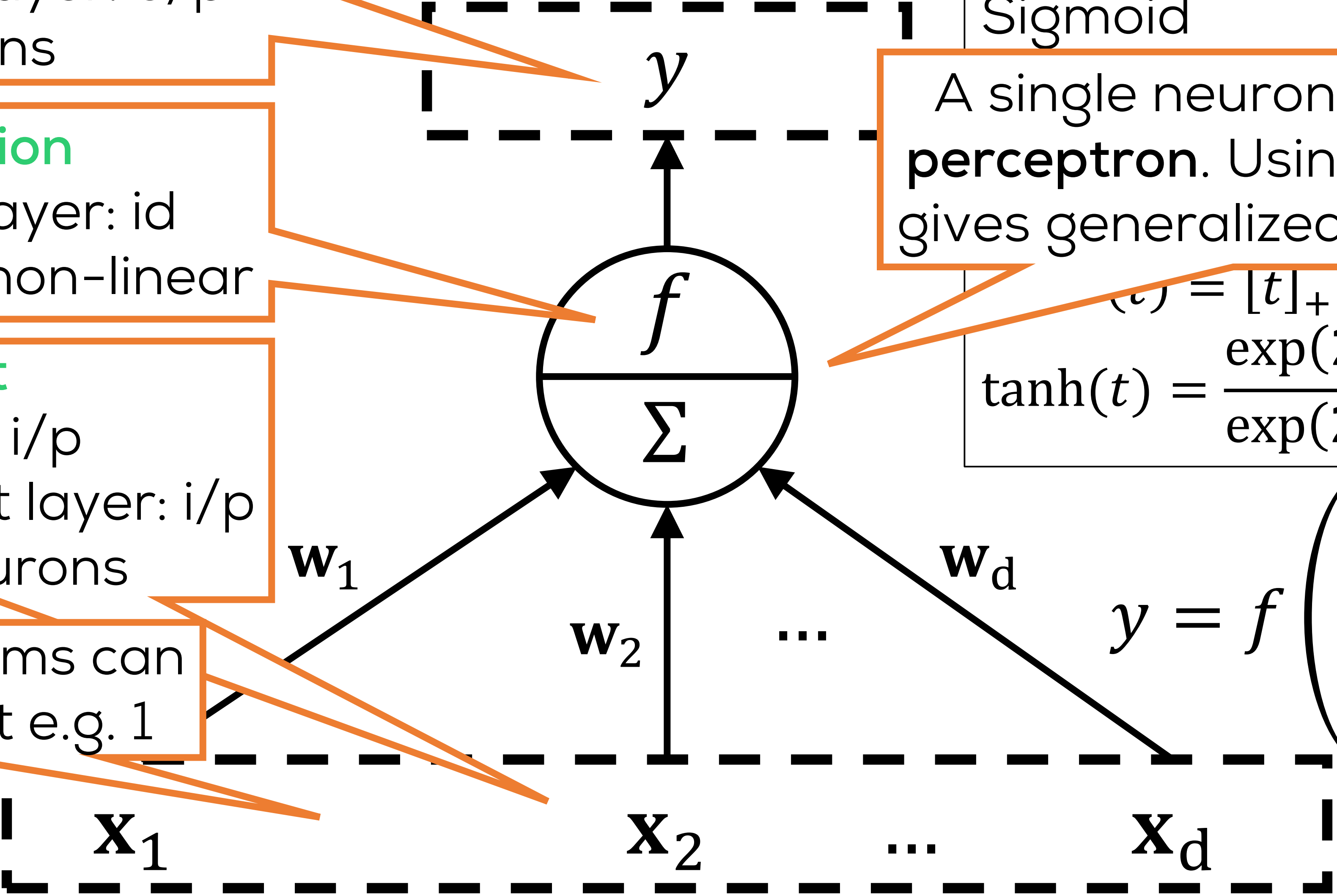
… in Neural Networks

Output
Output layer: final o/p
Input/hidden layer: o/p to other neurons

Activation
Input/output layer: id
Hidden layer: non-linear

Input
Input layer: no i/p
Hidden/output layer: i/p from other neurons

Some input items can be a constant e.g. 1



$$y = f\left(\sum_{i=1}^{d} \mathbf{w}_i \cdot \mathbf{x}_i\right)$$

Common “activation” fns f
Sigmoid

$(t) = [t]_+ = \max(t, 0)$

$$\tanh(t) = \frac{\exp(2t) - 1}{\exp(2t) + 1}$$

A single neuron also called a **perceptron**. Using non-linear f gives generalized linear models

Output
Output layer: final o/p
Input/hidden layer: o/p to other neurons

Sometimes output layer is given a non-id activation. Matter of convention

Common “activation” fns f

Sigmoid

$(t) = [t]_+ = \max(t, 0)$

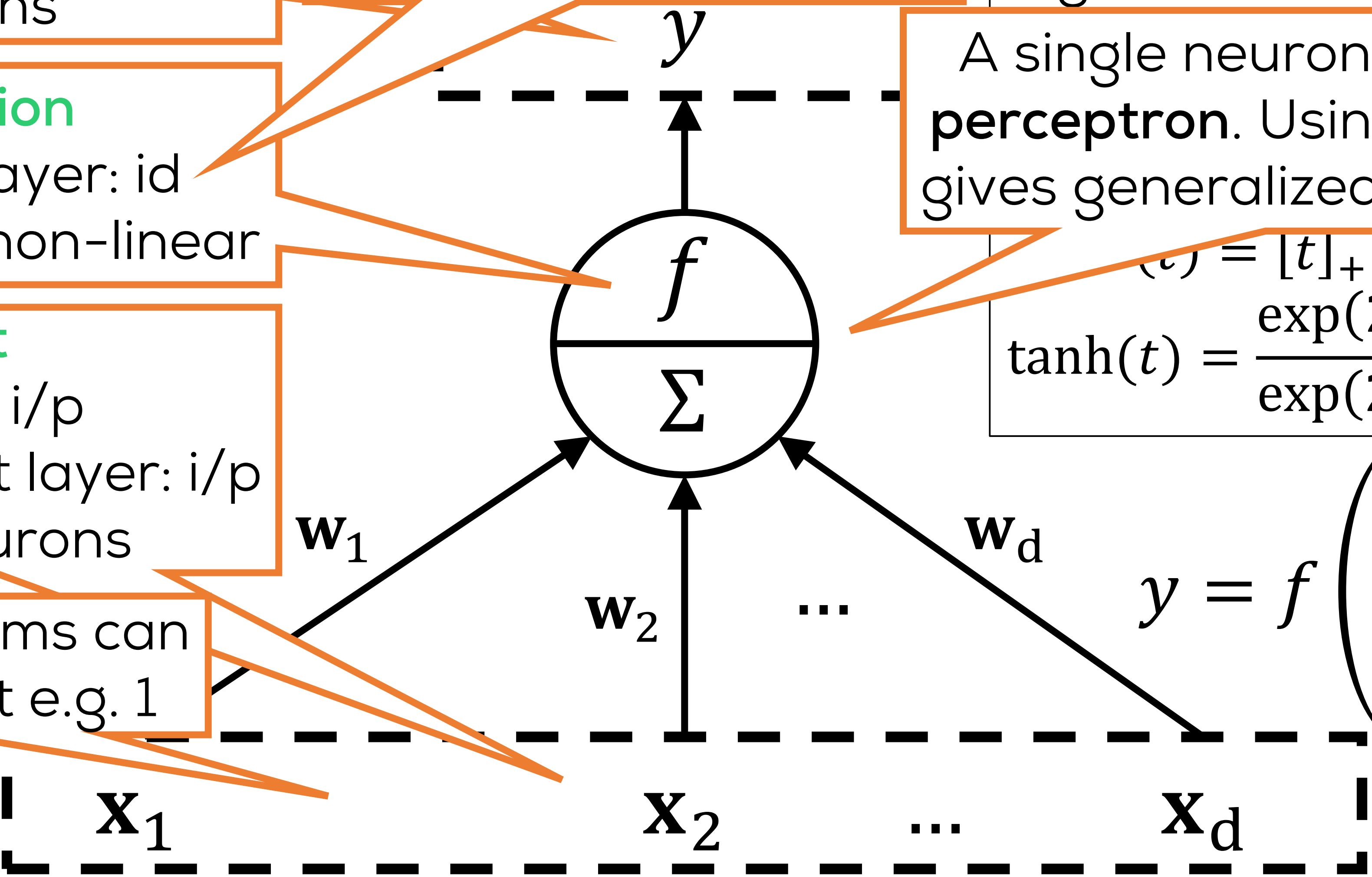
$$\tanh(t) = \frac{\exp(2t) - 1}{\exp(2t) + 1}$$

A single neuron also called a **perceptron**. Using non-linear f gives generalized linear models

Activation
Input/output layer: id
Hidden layer: non-linear

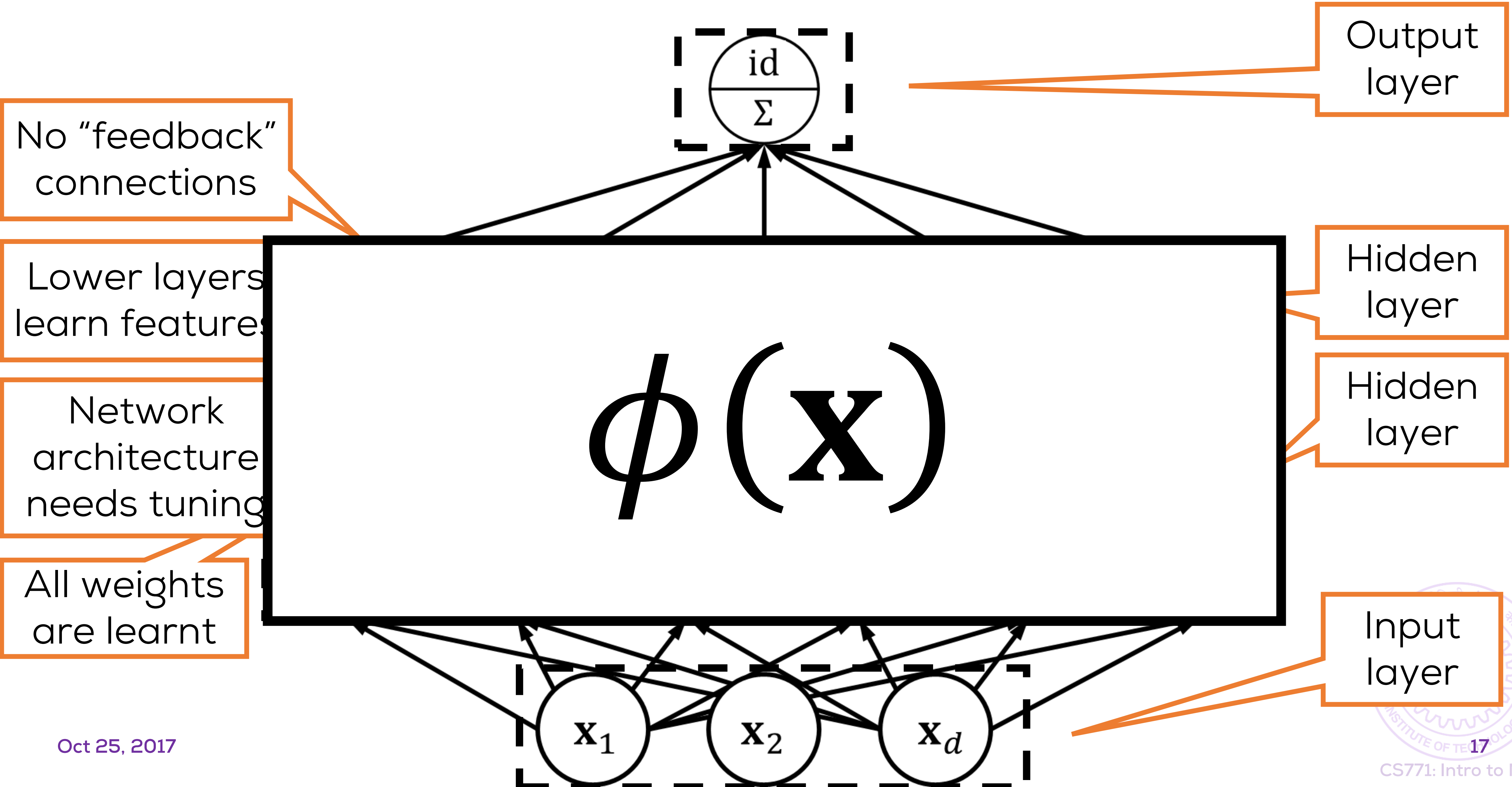
Input
Input layer: no i/p
Hidden/output layer: i/p from other neurons

Some input items can be a constant e.g. 1



$$y = f\left(\sum_{i=1}^{d} \mathbf{w}_i \cdot \mathbf{x}_i\right)$$

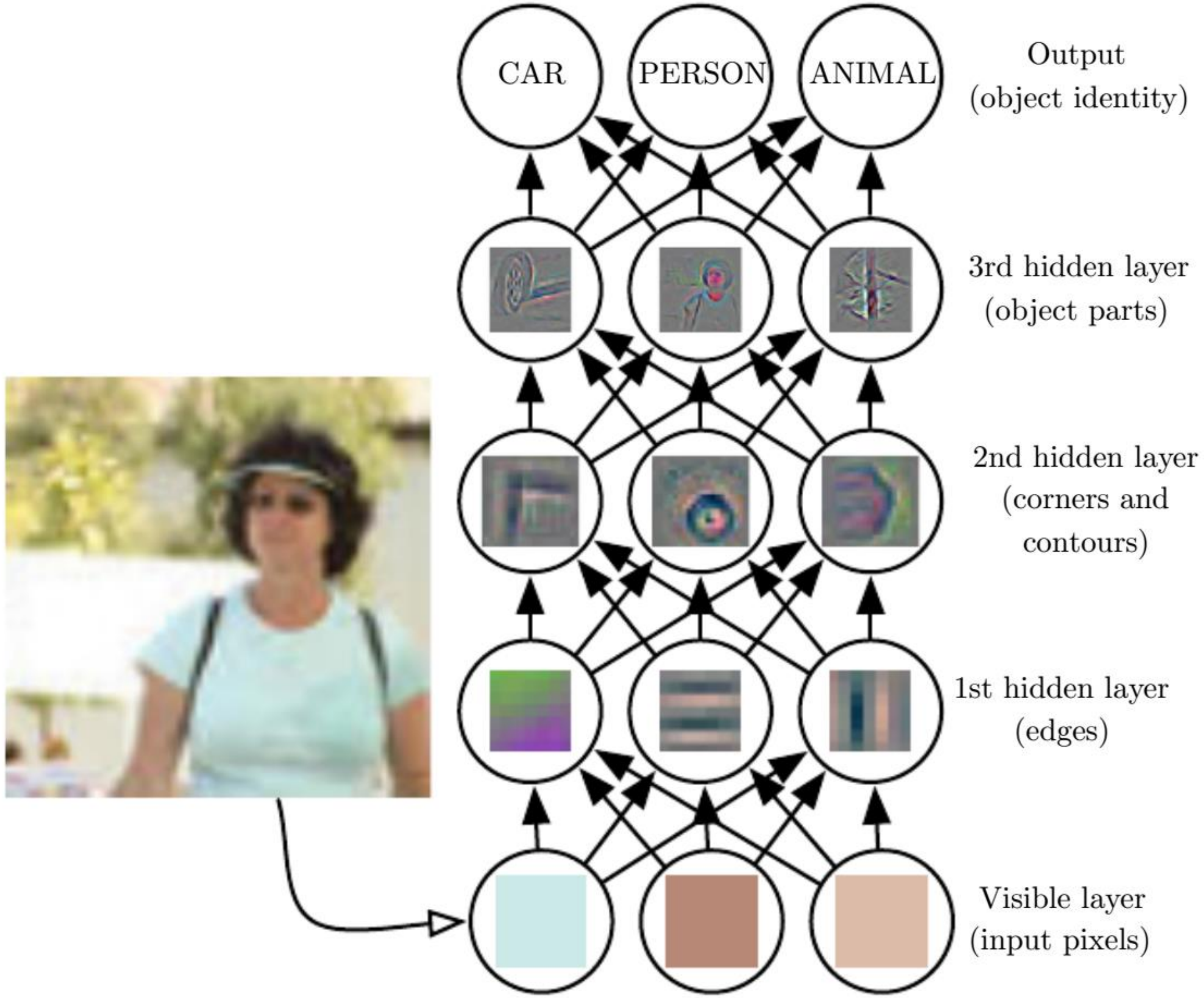
Feedforward Neural Network



Neural Networks as Feature Learners

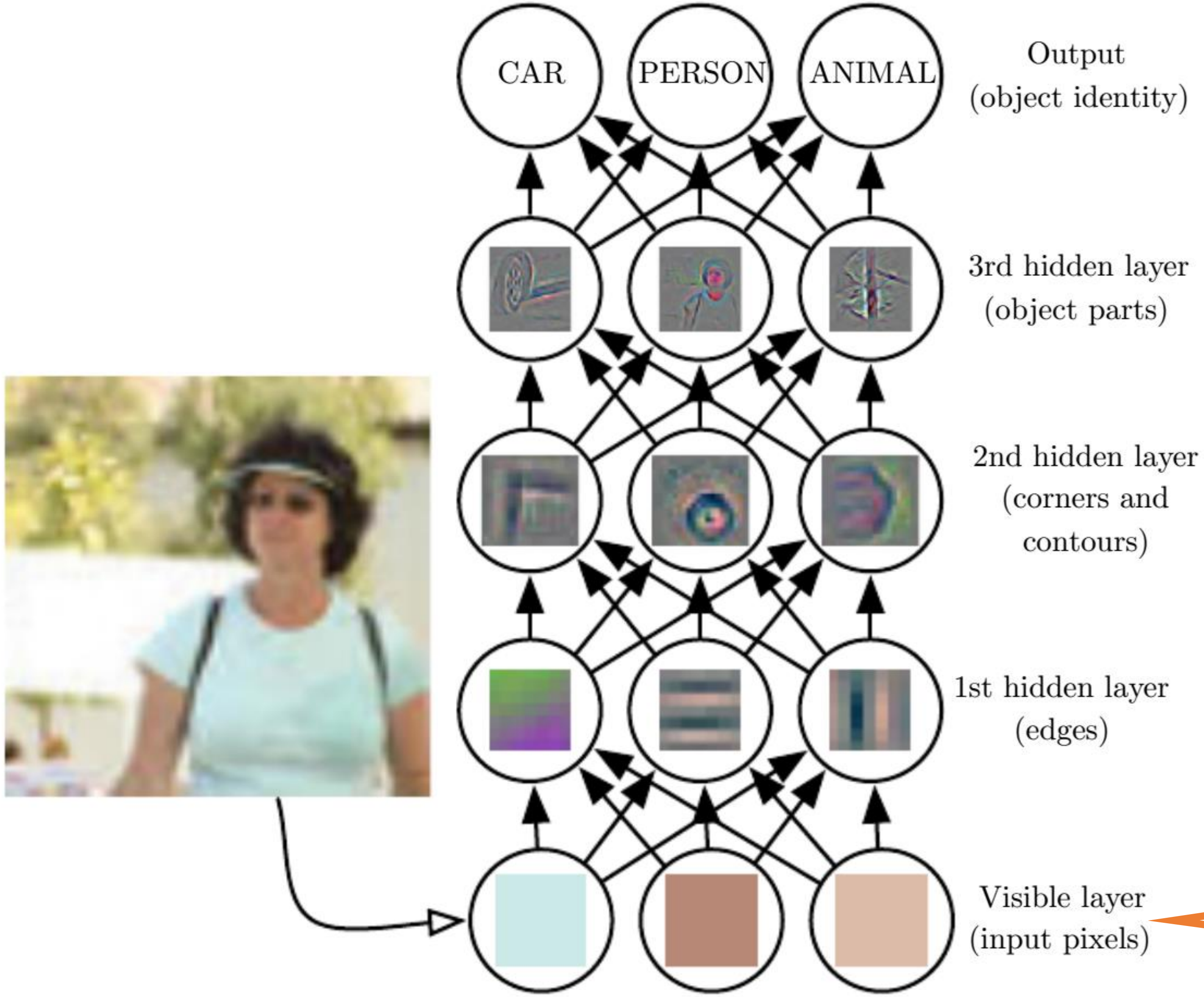
Zeiler, M. D. and Fergus, R. ECCV 2014

Neural Networks as Feature Learners



Zeiler, M. D. and Fergus, R. ECCV 2014

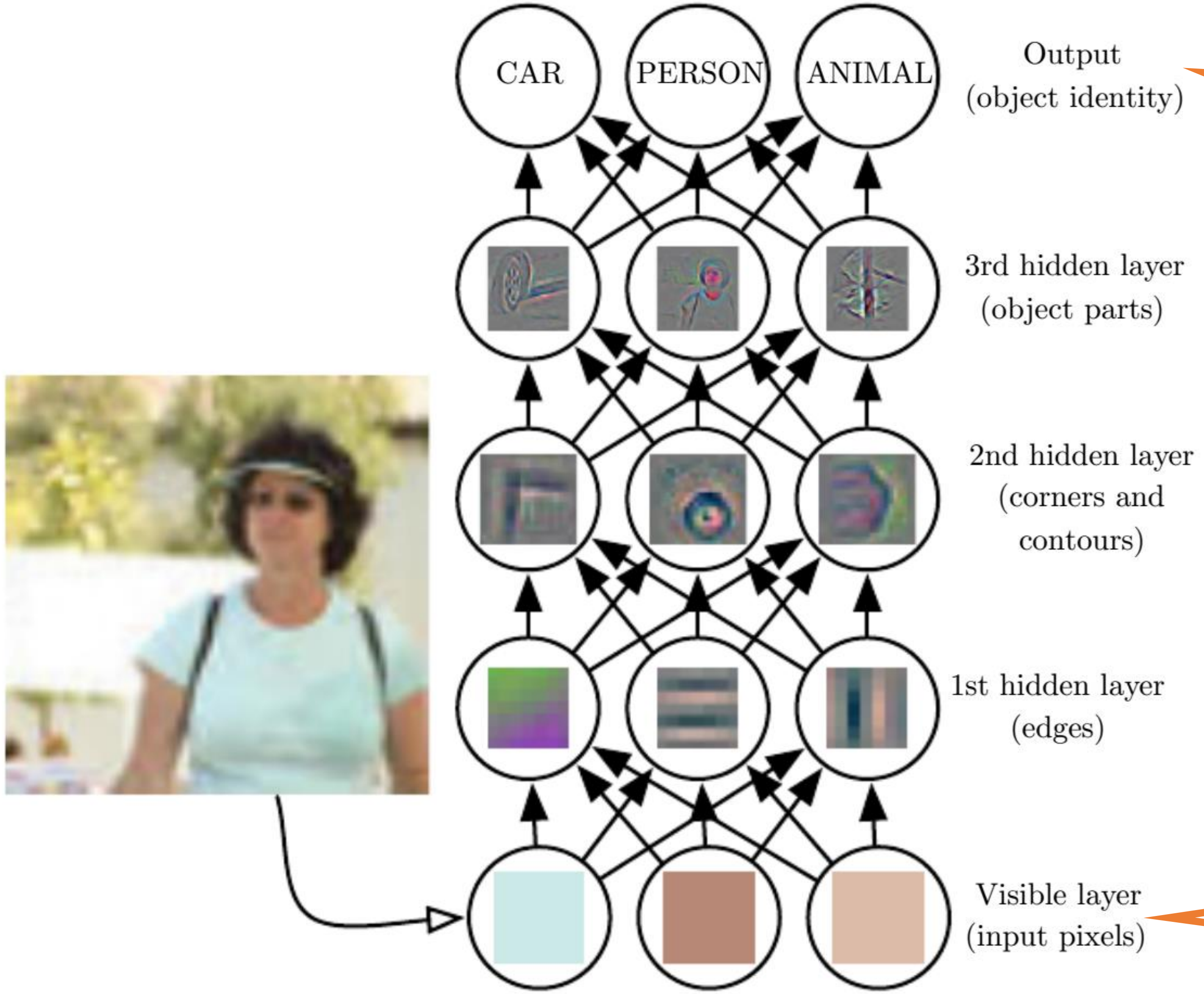
Neural Networks as Feature Learners



Input layer received with data i.e. “visible”

Zeiler, M. D. and Fergus, R. ECCV 2014

Neural Networks as Feature Learners

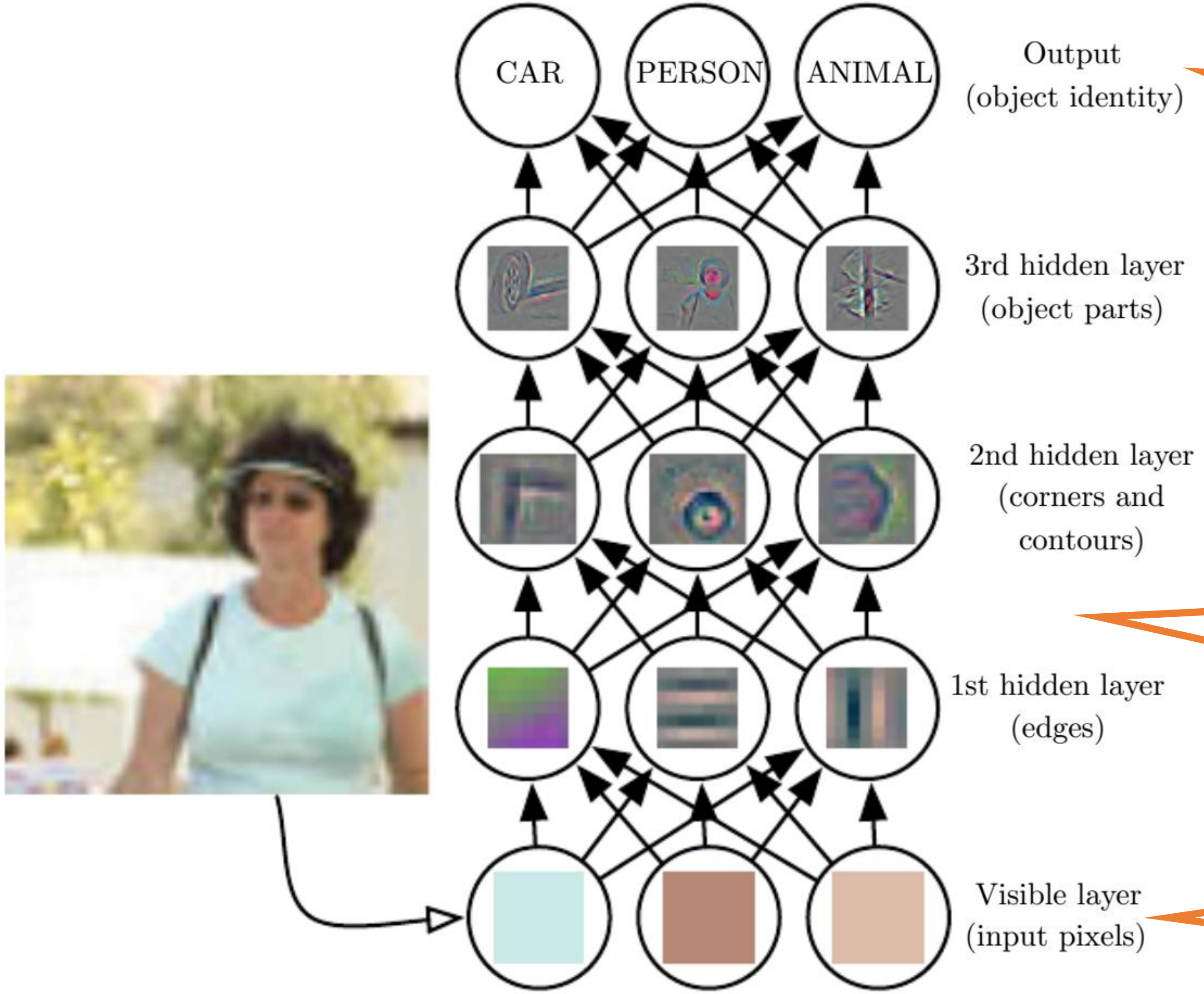


Output layer received with data i.e. "visible"

Input layer received with data i.e. "visible"

Zeiler, M. D. and Fergus, R. ECCV 2014

Neural Networks as Feature Learners



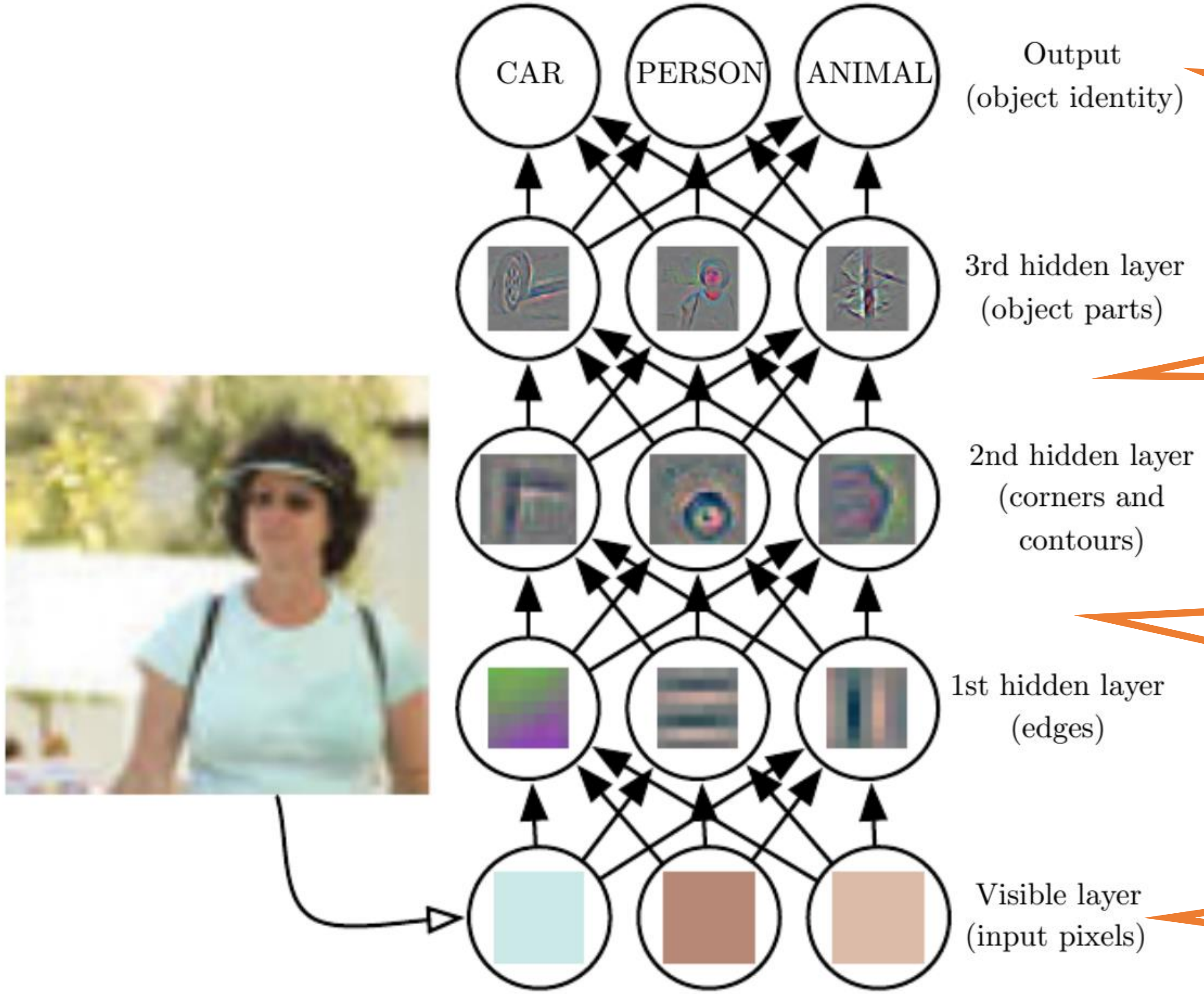
Output layer received with data i.e. "visible"

Hidden layer compute latent representations "hidden" from data

Input layer received with data i.e. "visible"

Zeiler, M. D. and Fergus, R. ECCV 2014

Neural Networks as Feature Learners



Output layer received with data i.e. "visible"

Greatly simplify learning task

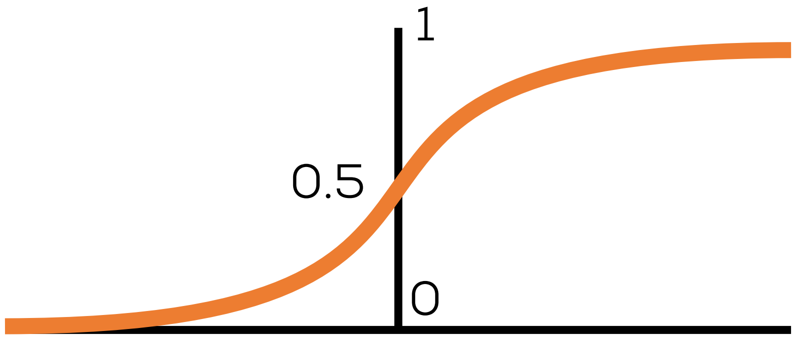
Hidden layer compute latent representations "hidden" from data

Input layer received with data i.e. "visible"

Zeiler, M. D. and Fergus, R. ECCV 2014

Activation/link functions

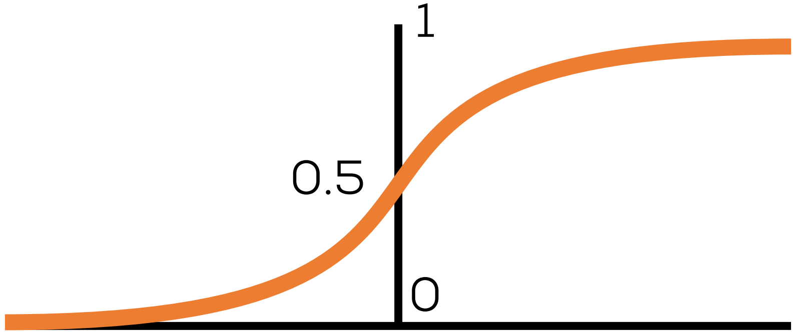
Activation/link functions



$$\sigma(t) = \frac{\exp(t)}{\exp(t) + 1}$$

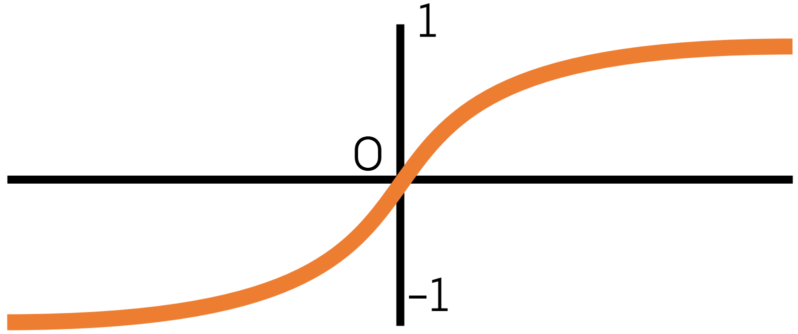
Sigmoid

Activation/link functions



$$\sigma(t) = \frac{\exp(t)}{\exp(t)+1}$$

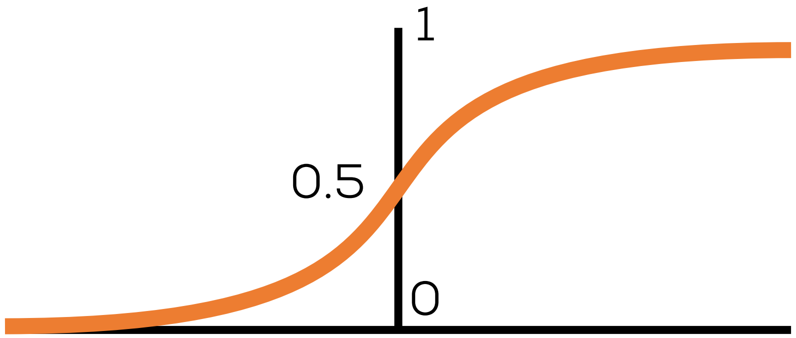
Sigmoid



$$\tanh(t) = \frac{\exp(2t)-1}{\exp(2t)+1} = 2\sigma(2t) - 1$$

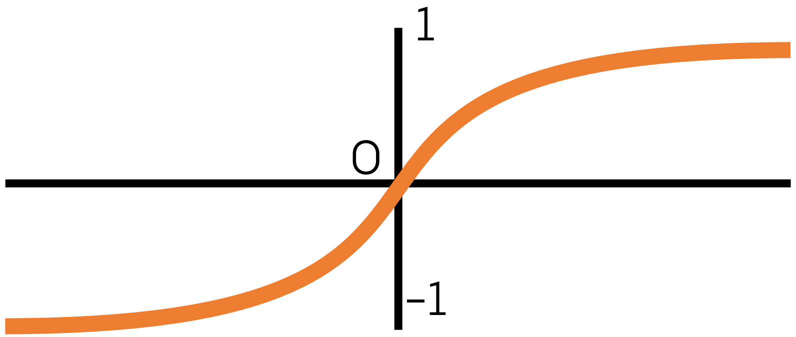
Hyperbolic Tangent

Activation/link functions



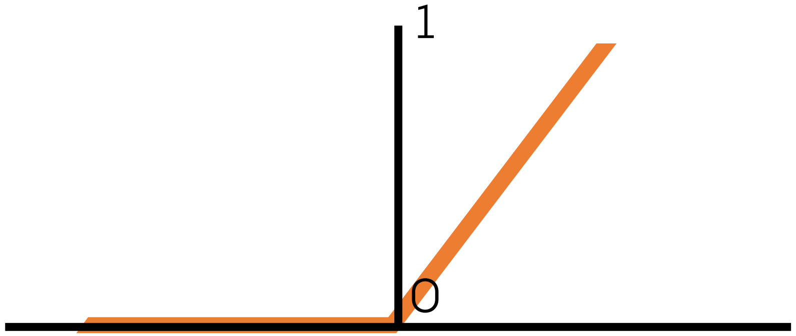
$$\sigma(t) = \frac{\exp(t)}{\exp(t) + 1}$$

Sigmoid



$$\tanh(t) = \frac{\exp(2t) - 1}{\exp(2t) + 1} = 2\sigma(2t) - 1$$

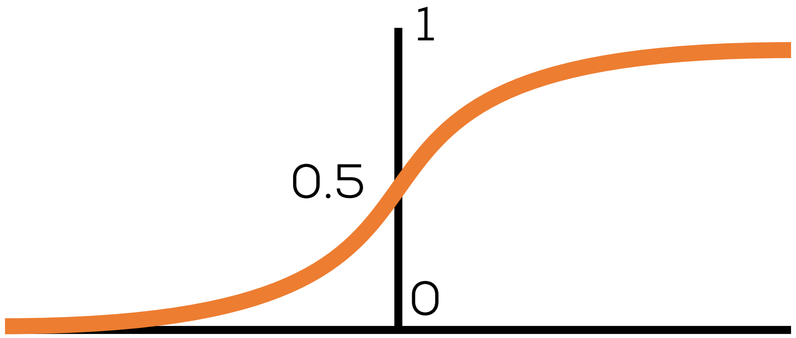
Hyperbolic Tangent



$$f(t) = \max(t, 0)$$

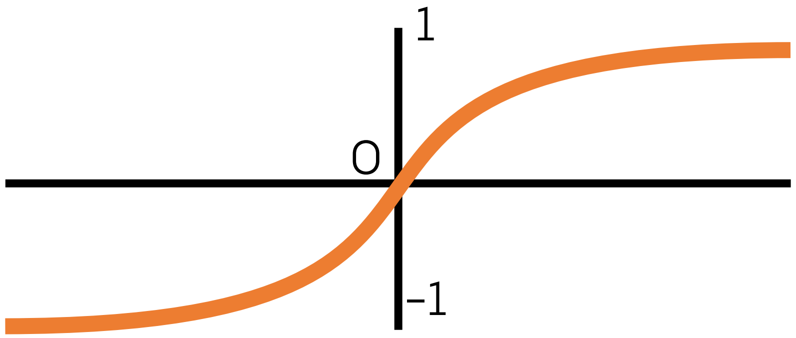
Rectified Linear Unit (ReLU)

Activation/link functions



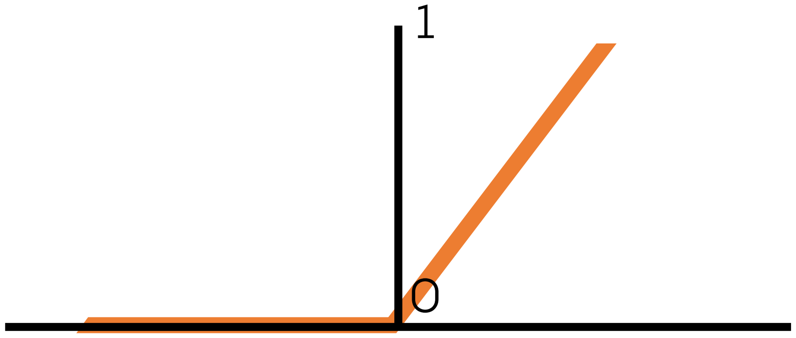
$$\sigma(t) = \frac{\exp(t)}{\exp(t)+1}$$

Sigmoid



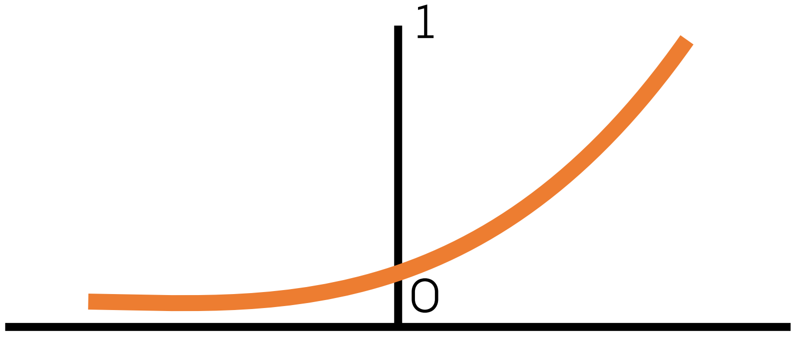
$$\tanh(t) = \frac{\exp(2t)-1}{\exp(2t)+1} = 2\sigma(2t)-1$$

Hyperbolic Tangent



$$f(t) = \max(t, 0)$$

Rectified Linear Unit (ReLU)

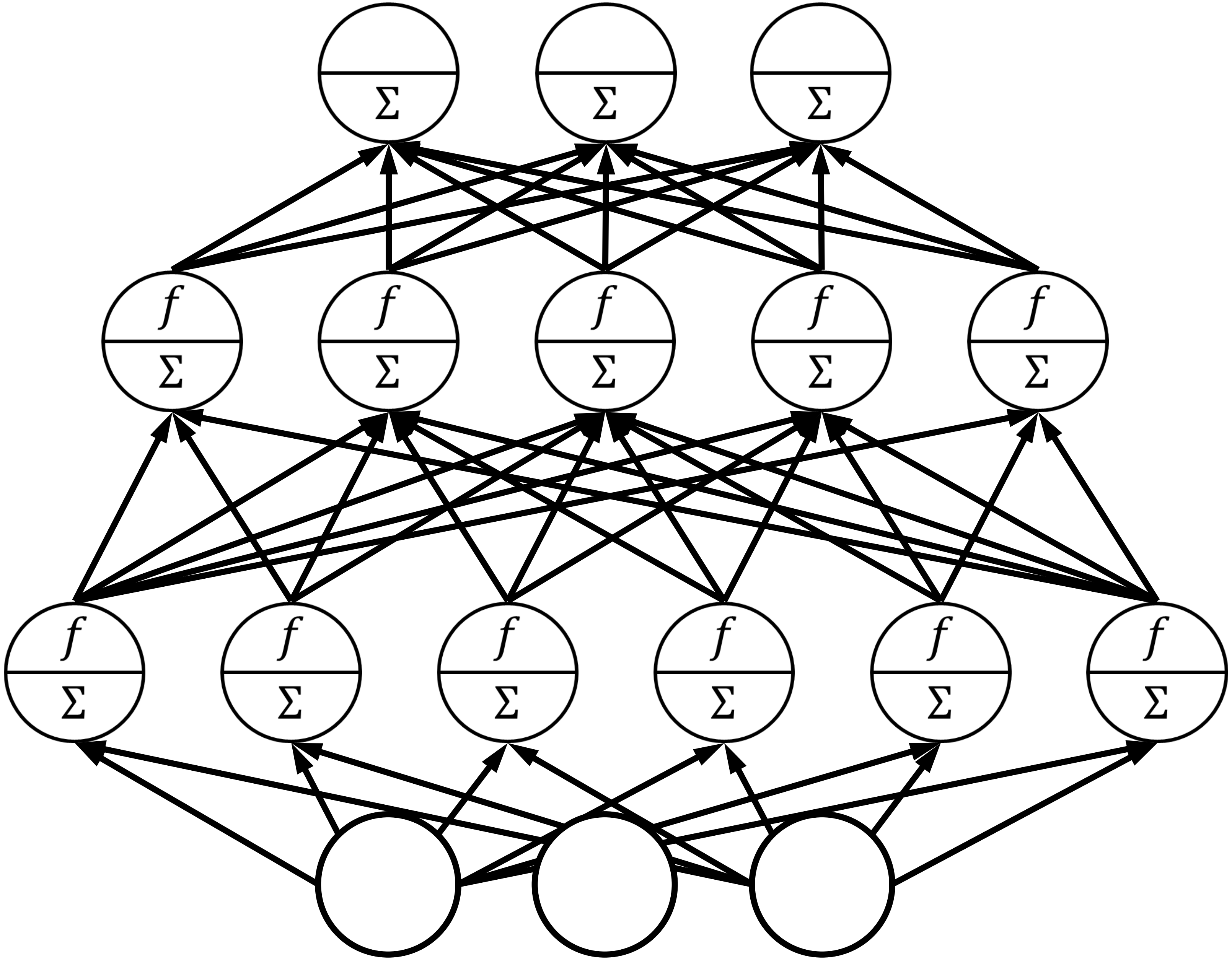


$$f(t) = \log(1+\exp(t))$$

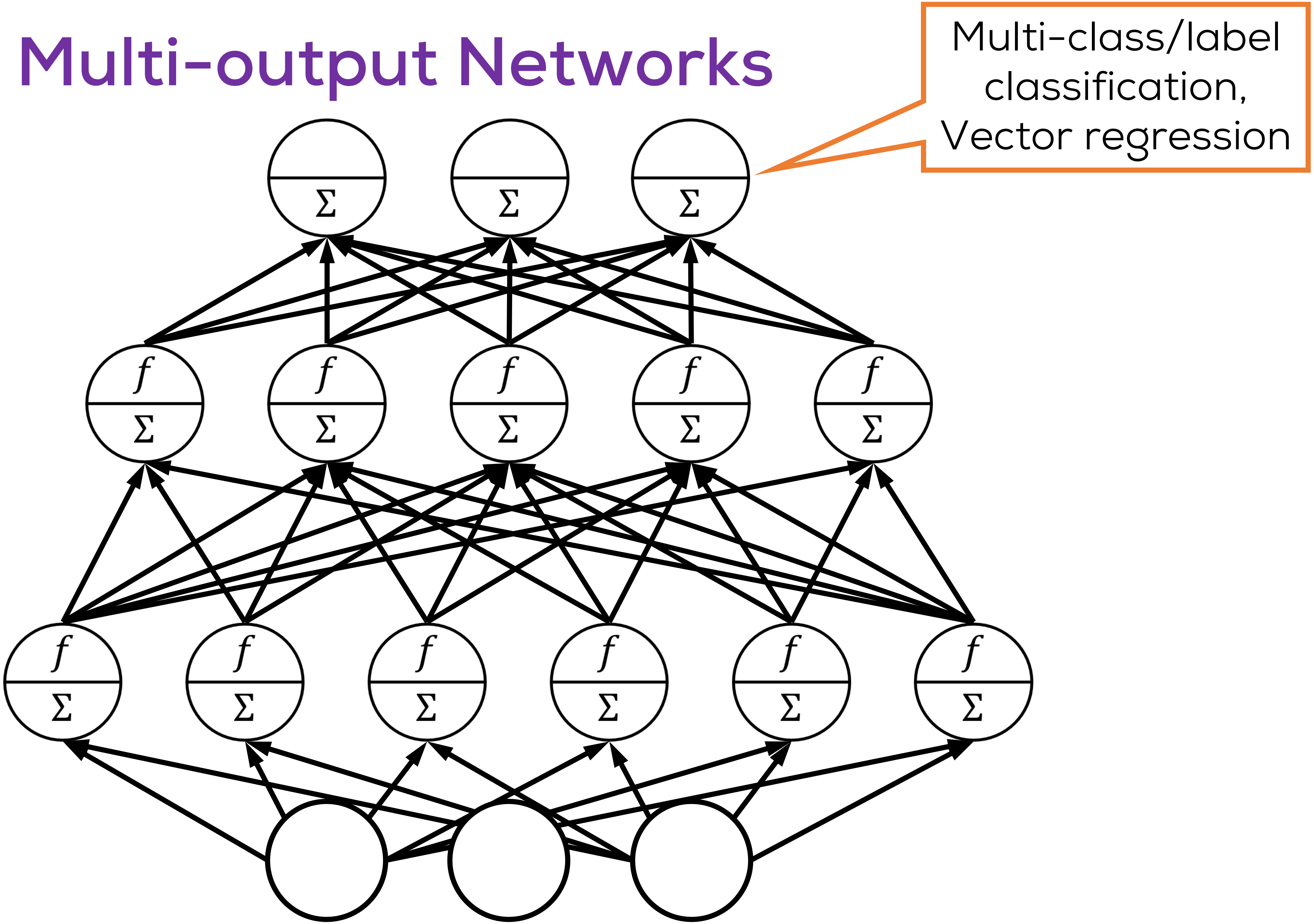
Softplus

Multi-output Networks

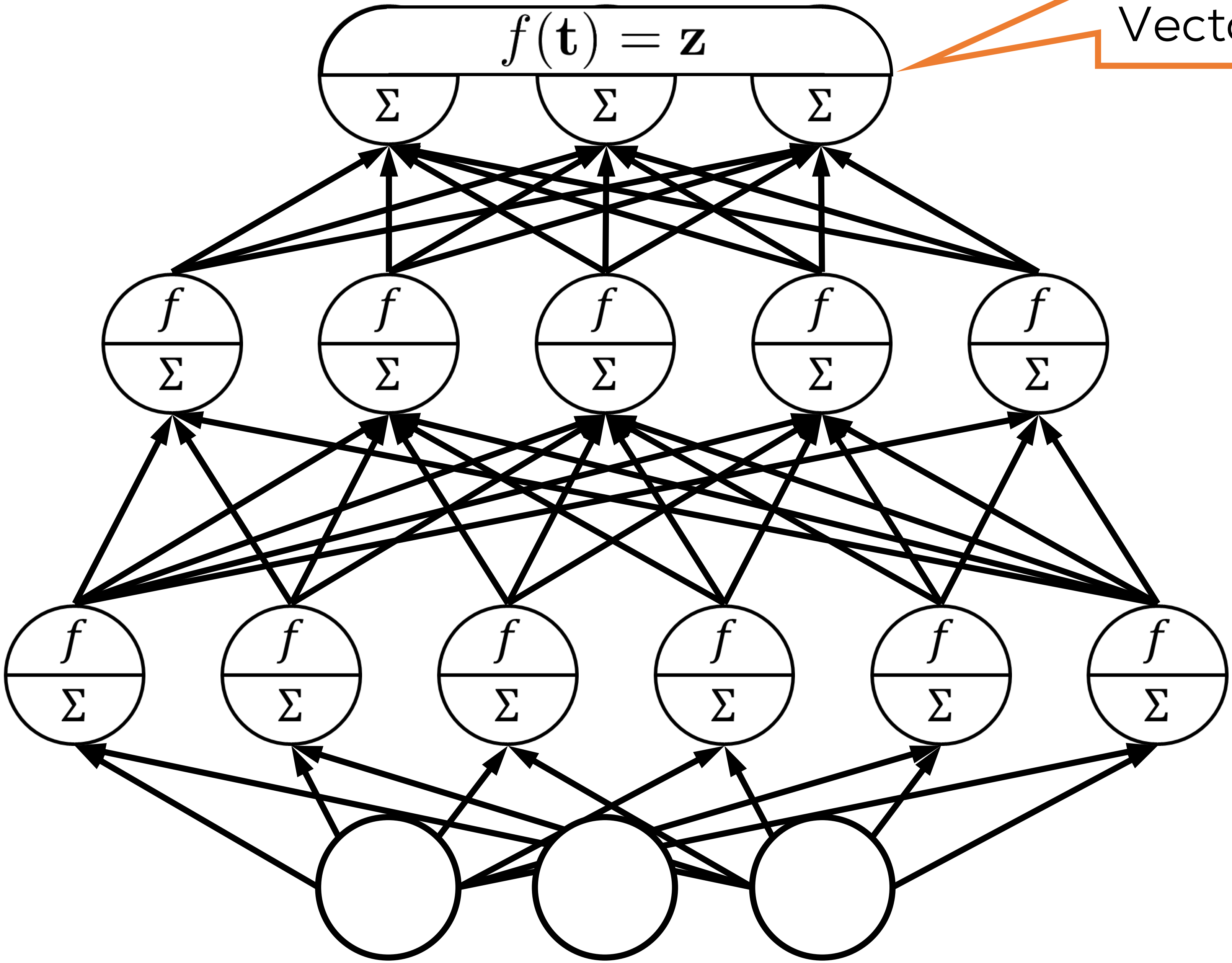
Multi-output Networks



Multi-output Networks

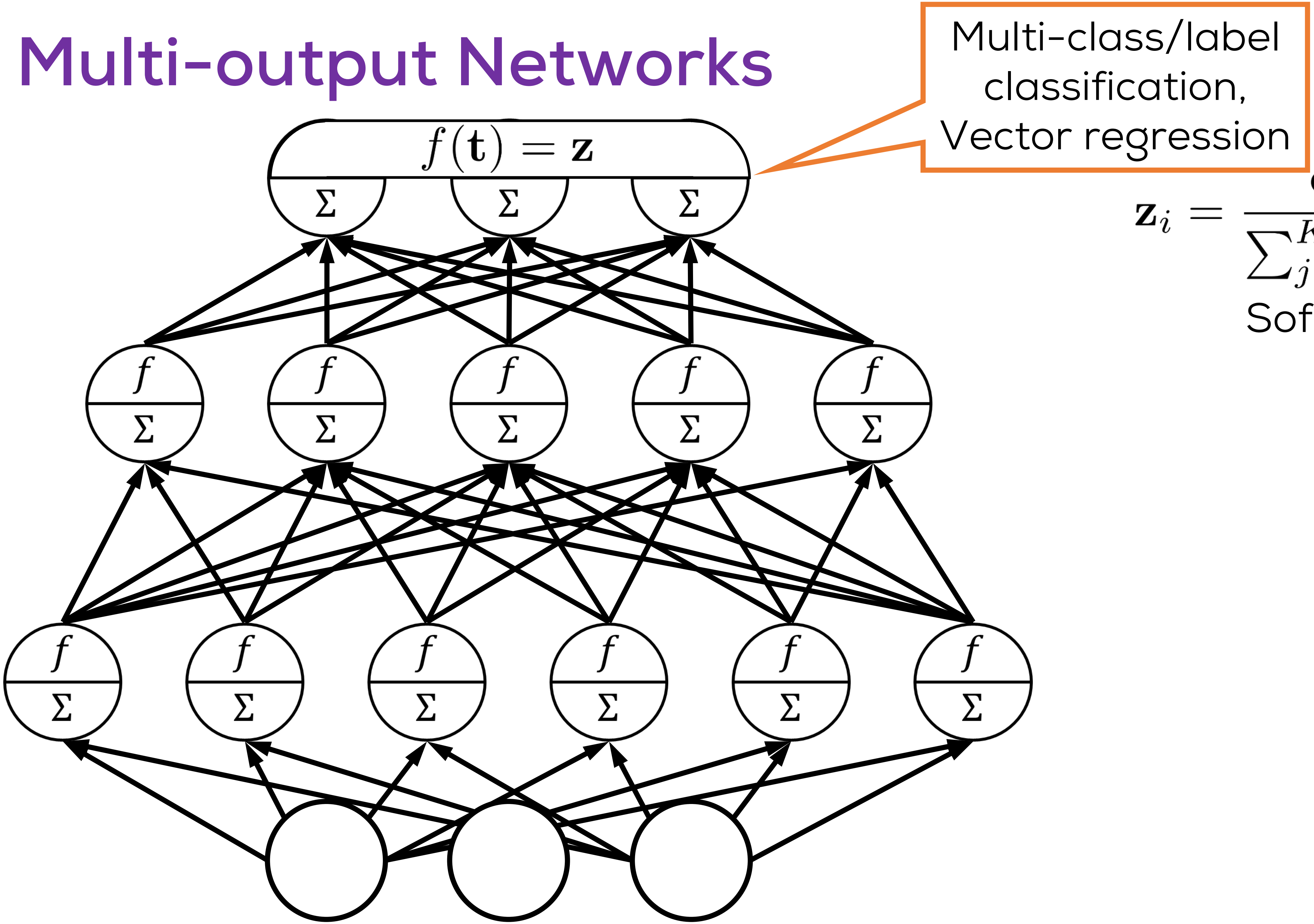


Multi-output Networks



Multi-class/label classification, Vector regression

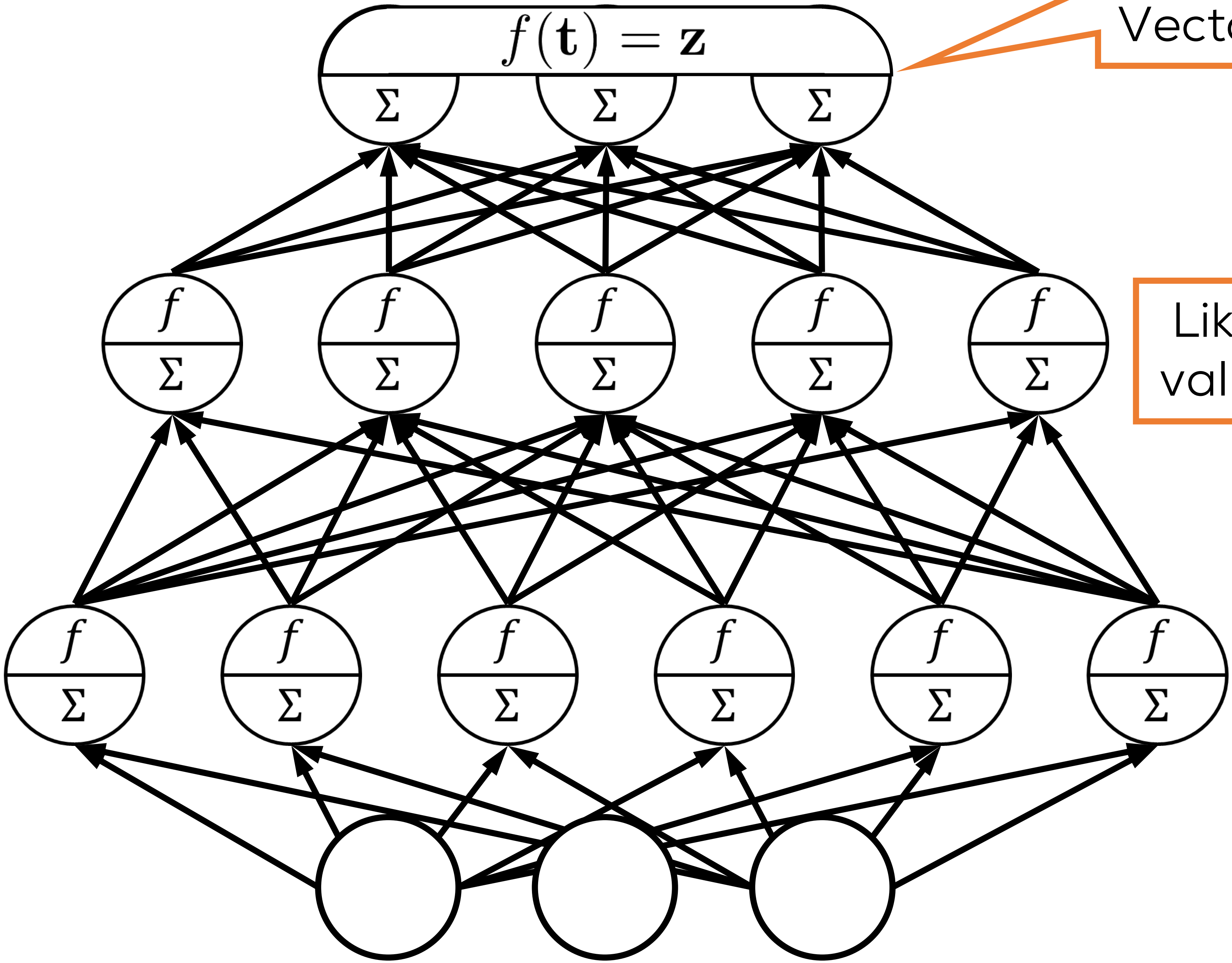
Multi-output Networks



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

Softmax

Multi-output Networks



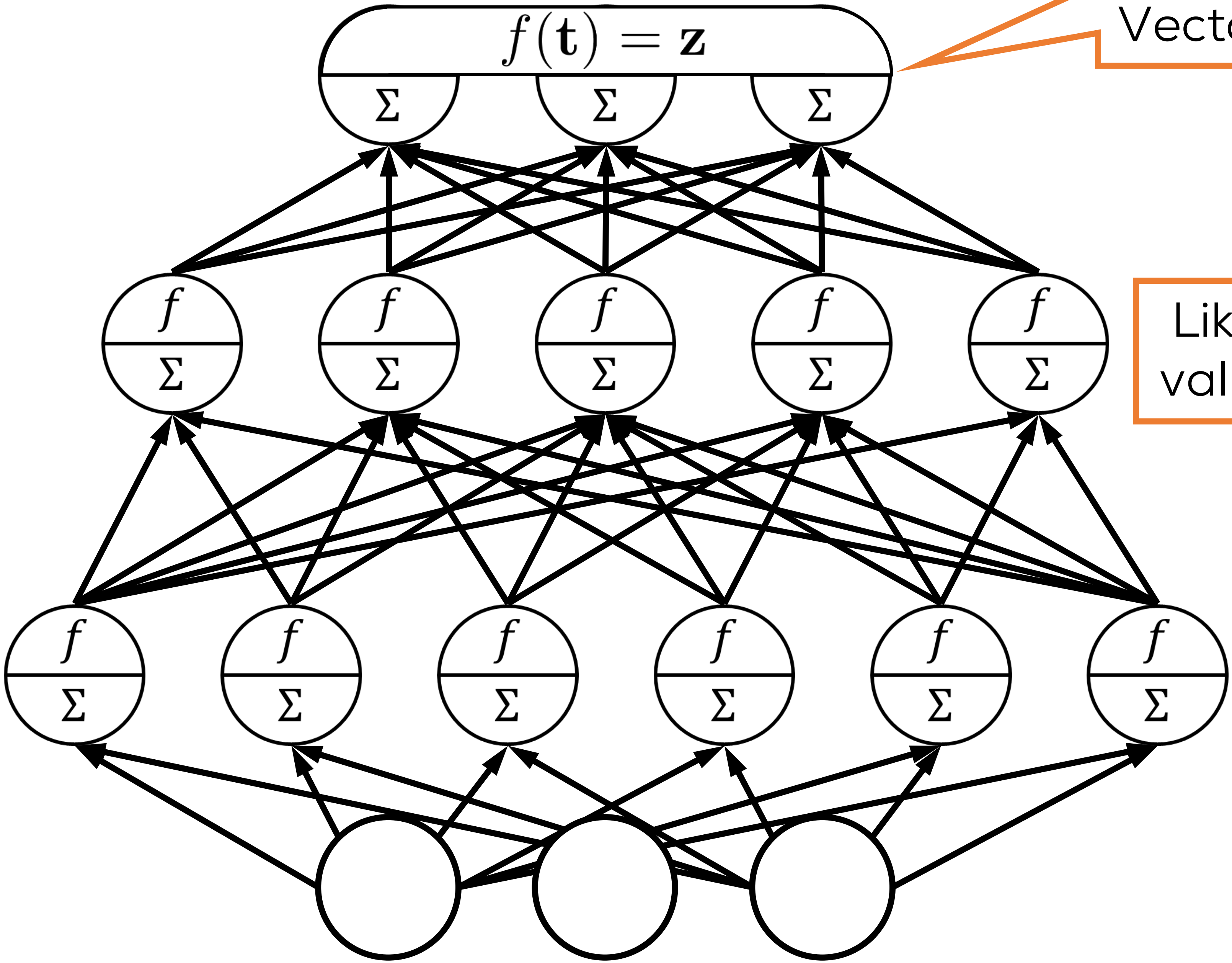
Multi-class/label classification, Vector regression

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

Softmax

Like sigmoid, converts real values to probability values

Multi-output Networks



Multi-class/label classification, Vector regression

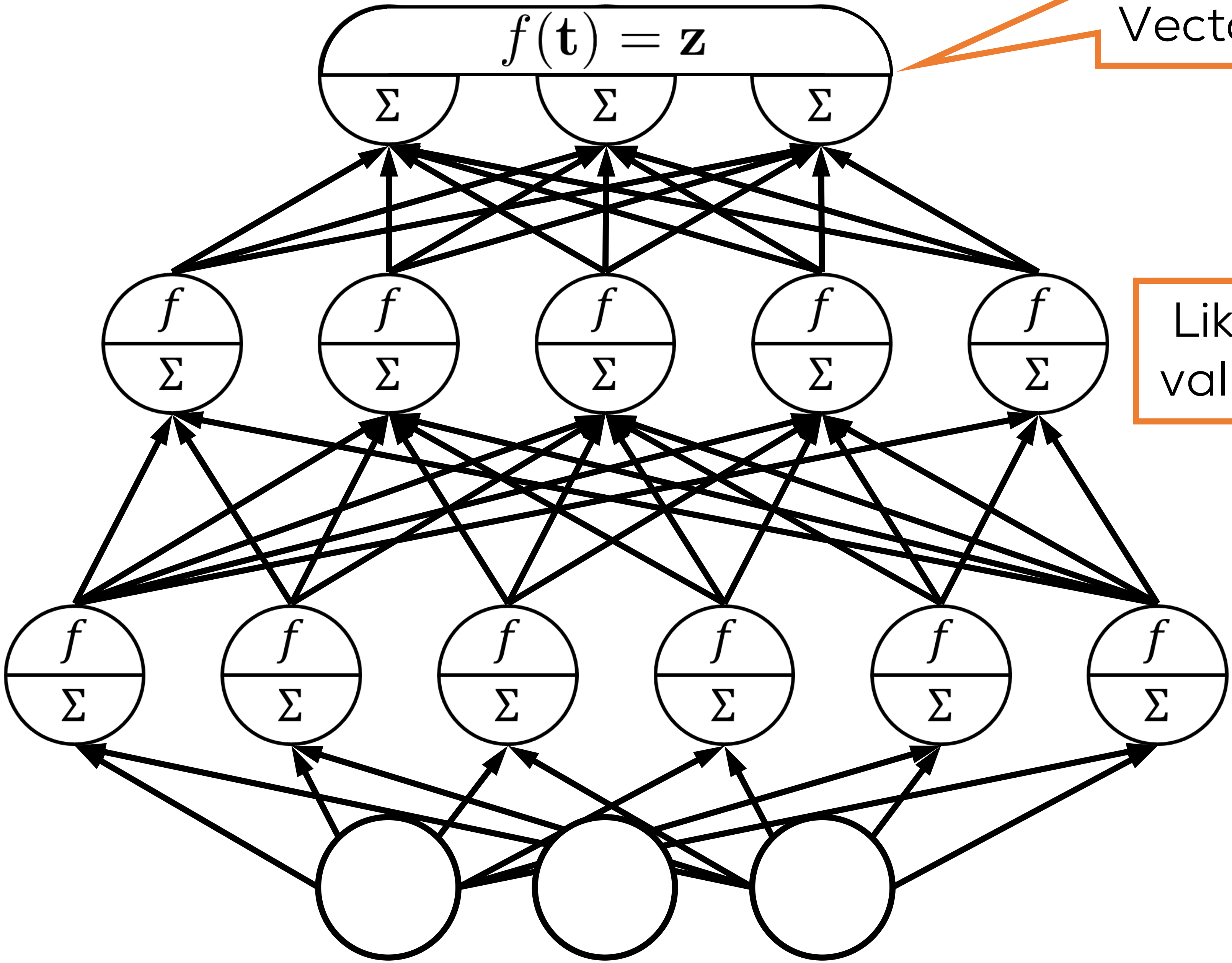
$$\mathbf{z}_i = \frac{\exp(\mathbf{t}_i)}{\sum_{j=1}^{K} \exp(\mathbf{t}_j)}$$

Softmax

Like sigmoid, converts real values to probability values

Useful in modelling likelihood maximization problems using NN

Multi-output Networks



Multi-class/label classification, Vector regression

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

Softmax

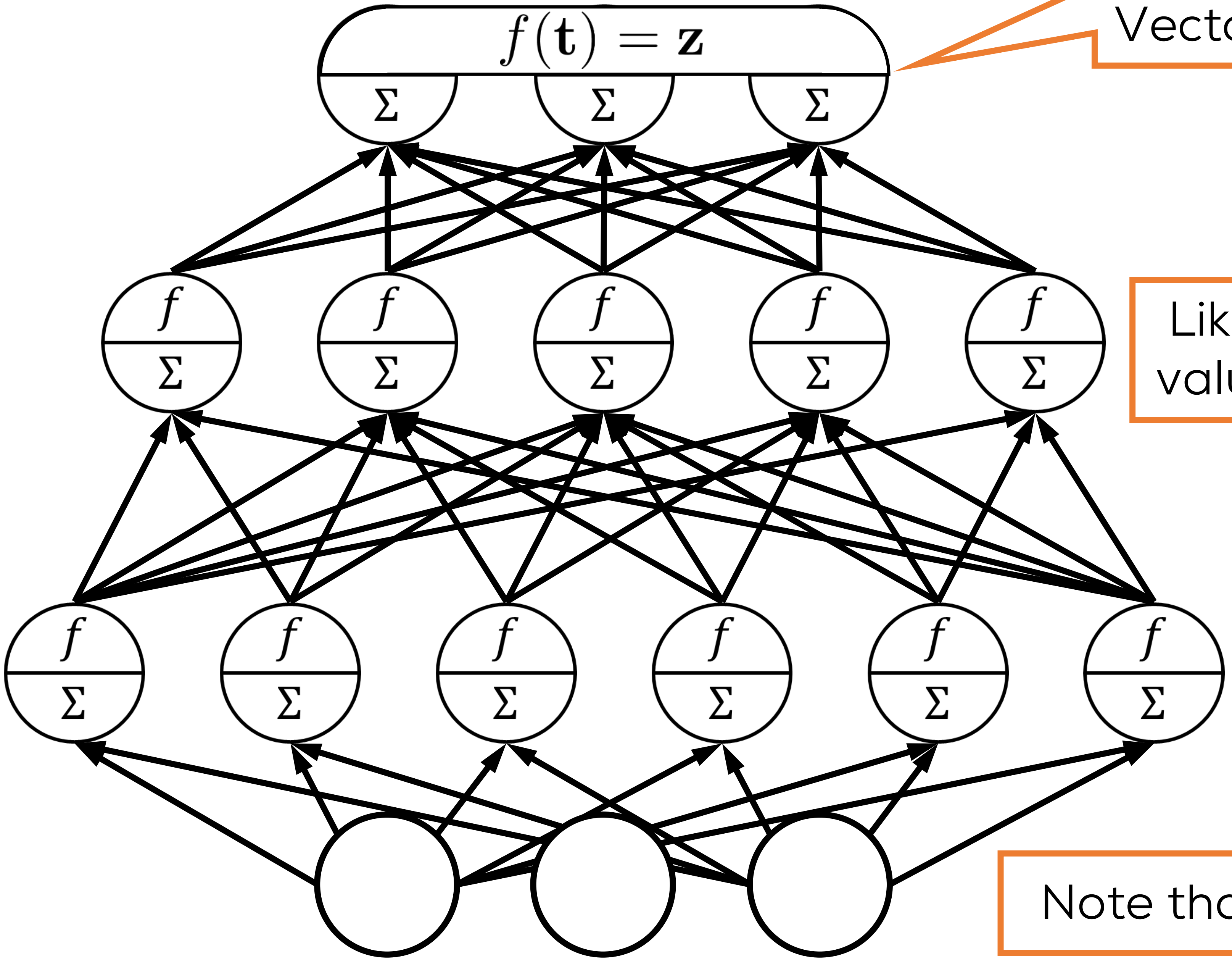
Like sigmoid, converts real values to probability values

Useful in modelling likelihood maximization problems using NN

Normalize before use

$$\tilde{\mathbf{t}}_i = \mathbf{t}_i - \max_j \mathbf{t}_j$$

Multi-output Networks



Multi-class/label classification, Vector regression

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

Softmax

Like sigmoid, converts real values to probability values

Useful in modelling likelihood maximization problems using NN

Normalize before use

$$\tilde{\mathbf{t}}_i = \mathbf{t}_i - \max_j \mathbf{t}_j$$

Note that $\mathrm{f_{SM}}(\mathbf{t}) = \mathrm{f_{SM}}(\tilde{\mathbf{t}})$

Loss/Cost Functions

- Squared loss

$$\ell_{\text{LS}}(\hat{y}, y) = (\hat{y} - y)^2$$

- Absolute difference

$$\ell_{\text{ABS}}(\hat{y}, y) = |\hat{y} - y|$$

- Negative log-likelihood loss

$$y \in [K], \hat{\mathbf{y}} = f_{\text{SM}}(\mathbf{t}), \mathbf{t} \in \mathbb{R}^K$$
$$\ell_{\text{NLL}}(\hat{\mathbf{y}}, y) = -\log\left(\hat{\mathbf{y}}_y\right)$$

- Cross-entropy

$$\ell_{\text{CE}}(\hat{y}, y) = y \cdot \log \hat{y} + (1 - y) \cdot \log(1 - \hat{y})$$

- Hinge loss

$$\ell_{\text{Hinge}}(\hat{y}, y) = [1 - y\hat{y}]_+$$

Loss/Cost Functions

- Squared loss

$$\ell_{\text{LS}}(\hat{y}, y) = (\hat{y} - y)^2$$

- Absolute difference

$$\ell_{\text{ABS}}(\hat{y}, y) = |\hat{y} - y|$$

- Negative log-likelihood loss

$$y \in [K], \hat{\mathbf{y}} = f_{\text{SM}}(\mathbf{t}), \mathbf{t} \in \mathbb{R}^K$$
$$\ell_{\text{NLL}}(\hat{\mathbf{y}}, y) = -\log\left(\hat{\mathbf{y}}_y\right)$$

- Cross-entropy

$$\ell_{\text{CE}}(\hat{y}, y) = y \cdot \log \hat{y} + (1 - y) \cdot \log(1 - \hat{y})$$

- Hinge loss

$$\ell_{\text{Hinge}}(\hat{y}, y) = [1 - y\hat{y}]_+$$

LS used with identity/ReLU activation

Sigmoid/softmax flatten out quickly so LS doesn't work

NLL, CE used with sigmoid/softmax activations

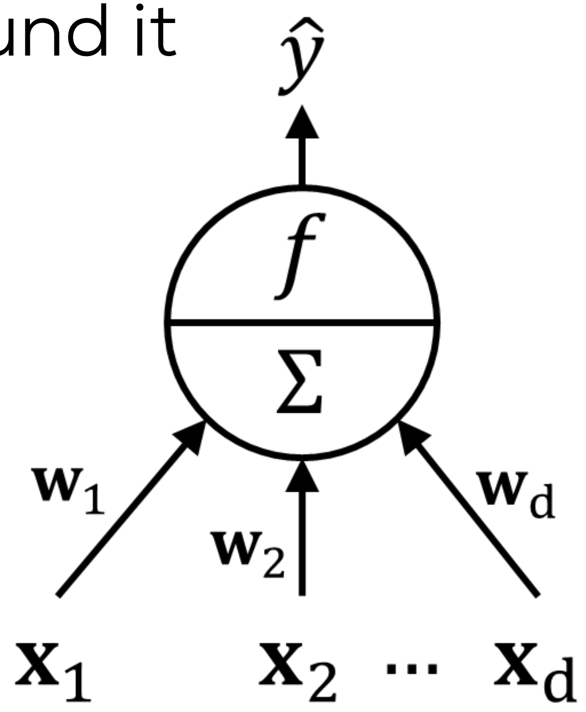
CE/Hinge usually used when $y \in \{0,1\}$ is binary

Training a Perceptron

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

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

Gradient Descent Revisited

Gradient Descent 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

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?

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?

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?

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?

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

- 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

- Batch gradient

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

Chain rule!

Very small batch sizes usually not used for deep networks

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

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

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?

- Simply rule of thumb – naïve but fast
  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
$$\eta_t = \arg\min_{\eta \geq 0} F(\mathbf{w}^t - \eta \cdot \mathbf{g}^t)$$
- Can we do something more adaptive?
- Can we let each coordinate of $\mathbf{w}$ get its own step length?
- $\mathbf{w}^{t+1} \leftarrow \mathbf{w}^t - \eta \cdot H^{-1}\mathbf{g}^t$ where $H = \mathrm{diag}(h_1, h_2, \ldots, h_d)$
- Wait … this looks like the Newton method!
- Indeed this is an approximate Newton method – wait a bit!

Momentum Methods

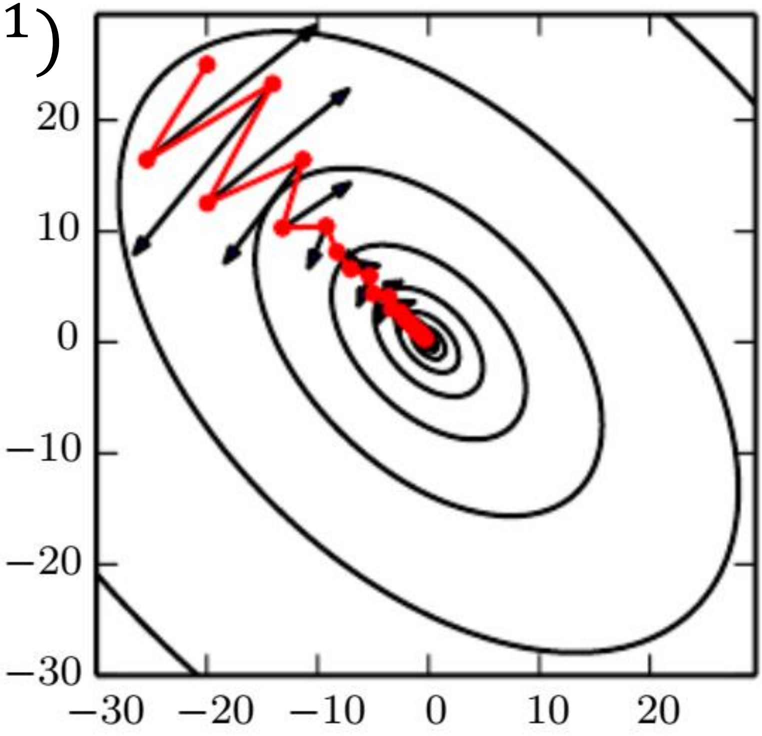
- Introduce a velocity term to push GD along, avoid oscillations

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

- **Nesterov's accelerated gradient (NAG)**
- Does a "look-ahead"

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

- For "smooth" convex problems, NAG ensures ϵ-convergence in just $\mathcal{O}(1/\epsilon)$ steps
  hence the name "accelerated" gradient
- Don't have very deep insights why it works 😐



Adaptive Learning Rates

$$\mathbf{w}^{t+1} \leftarrow \mathbf{w}^t - \eta \cdot (H^t)^{-1}\mathbf{g}^t$$
$$H^t = \mathrm{diag}(h_1^t, h_2^t, \ldots, h_d^t)$$

- Adagrad (Duchi et al. 2011)

$$h_i^t = \sqrt{\epsilon + \sum_{\tau=1}^{t} \left(\mathbf{g}_i^\tau\right)^2}$$

- Note that if all coordinates get roughly similar gradients then we have $h_i^t \approx t$ and Adagrad behaves as if we had set $\eta_t \approx \eta/t$
- However, if some coordinate getting updated very vigorously, $|\mathbf{g}_i^\tau| \gg 0$ for all τ, Adagrad slows it down a bit
- If some coordinate is static $\mathbf{g}_i^\tau \equiv 0$ for all τ, then $h_i^t = \epsilon$, no effect

Adaptive Learning Rates

$$\mathbf{w}^{t+1} \leftarrow \mathbf{w}^{t} - \eta \cdot (H^{t})^{-1}\mathbf{g}^{t}$$
$$H^{t} = \mathrm{diag}(h_1^t, h_2^t, \ldots, h_d^t)$$

- RMSProp (Hinton 2012)

$$h_i^t = \sqrt{\epsilon + v_i^t}$$

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

- Adagrad can be too aggressive in forcing step sizes down
- RMSProp has better performance in non-convex settings
- Hinton suggests $\gamma \approx 0.9$
- May be combined with NAG as well!

Adaptive Learning Rates

$$\mathbf{w}^{t+1} \leftarrow \mathbf{w}^t - \eta \cdot (H^t)^{-1}\mathbf{u}^t$$
$$H^t = \mathrm{diag}(h_1^t, h_2^t, \dots, h_d^t)$$

- Adam (Kingma and Ba 2014)

$$h_i^t = \sqrt{\epsilon + v_i^t}$$
$$\mathbf{u}^t = \gamma_1 \cdot \mathbf{u}^{t-1} + (1-\gamma_1) \cdot \mathbf{g}^t$$
$$v_i^t = \gamma_2 \cdot v_i^{t-1} + (1-\gamma_2) \cdot \left(\mathbf{g}_i^t\right)^2$$

- Keeps track of past gradients as well as squared gradients
- Actually does a bias correction step before using $\mathbf{u}^t$ and H^t
- Details can be found in Deep Learning textbook

How to prevent overfitting?

- Add a regularization term L_2/L_1 to the objective

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

- Gradients calculations change very slightly
- Constraint the weights of the network to satisfy $|\mathbf{w}_i| < r$

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

- 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
  - Effectively trains on multiple sparse networks in parallel
  - While executing a GD update, randomly remove edges or entire nodes from network so they do not participate
  - Can be shown to be equivalent to L_2 reg. in nice settings

Other techniques used to train NNs

- Pre-training
- Batch normalization
- Curriculum learning
- Pre-training
- Conjugate gradient descent
- Normalized gradient descent
- Approximate Newton method L-BFGS
- Some of these developed earlier for convex opt. problems
- Some we have discussed earlier in context of linear models
  Coordinate descent, Model averaging (Ruppert-Polyak method)

Nice discussion in the Deep Learning book

Up Next

- Training multi-layered perceptrons - backpropagation
- Autoencoders
- RNNs
- CNNs
- GANs
- Cannot go into too many details but will cover basics ☺

Please give your Feedback

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