Unconstrained Minimization of function (Example)

- Minimize

$$f(x_1, x_2, x_3) = (x_1)^2 + x_1(1 - x_2) + (x_2)^2 - x_2 x_3 + (x_3)^2 + x_3$$

- Gradient

$$\nabla_X f^T = \begin{bmatrix} 2x_1 + 1 - x_2 \\ -x_1 + 2x_2 - x_3 \\ -x_2 + 2x_3 + 1 \end{bmatrix}$$

Unconstrained Minimization of function (Example)

- Set the gradient to null

$$\nabla_X f = 0 \Rightarrow \begin{bmatrix} 2x_1 + 1 - x_2 \\ -x_1 + 2x_2 - x_3 \\ -x_2 + 2x_3 + 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

- Solving the 3 equations system with 3 unknowns

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix}$$

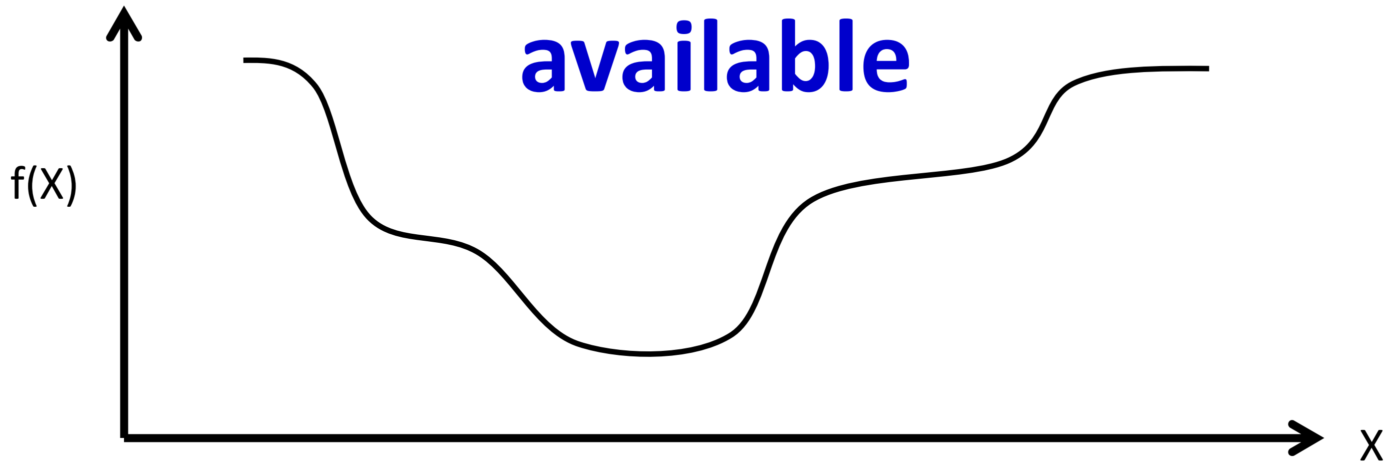
Unconstrained Minimization of function (Example)

- Compute the Hessian matrix $\nabla_X^2 f = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix}$
- Evaluate the eigenvalues of the Hessian matrix

$$\lambda_1 = 3.414, \;\; \lambda_2 = 0.586, \;\; \lambda_3 = 2$$

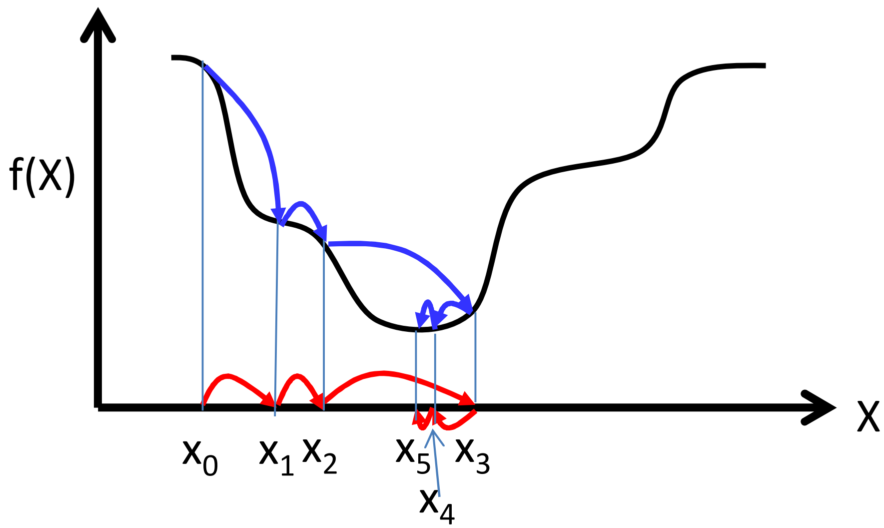
- All the eigenvalues are positives => the Hessian matrix is positive definite
- The point $x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix}$ is a minimum

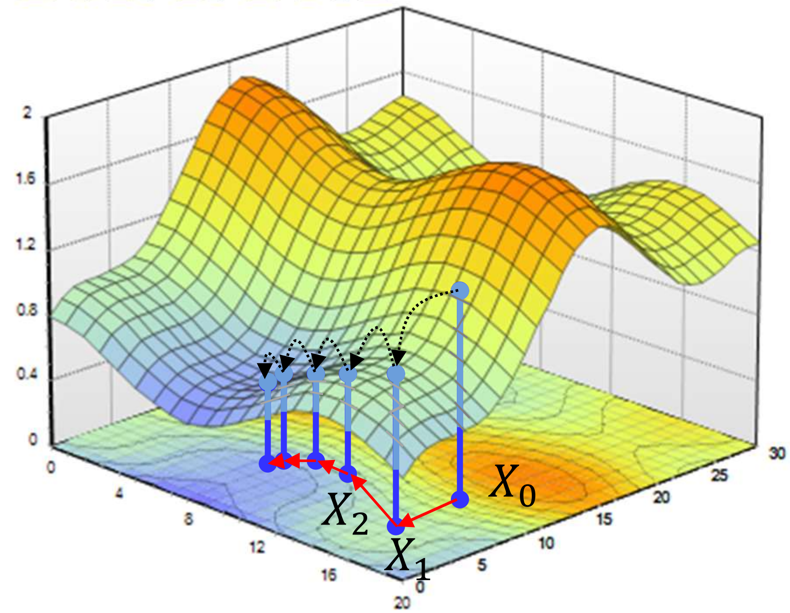
Closed Form Solutions are not always available



- Often it is not possible to simply solve $\nabla_X f(X) = 0$
  - The function to minimize/maximize may have an intractable form
- In these situations, iterative solutions are used
  - Begin with a “guess” for the optimal X and refine it iteratively until the correct value is obtained

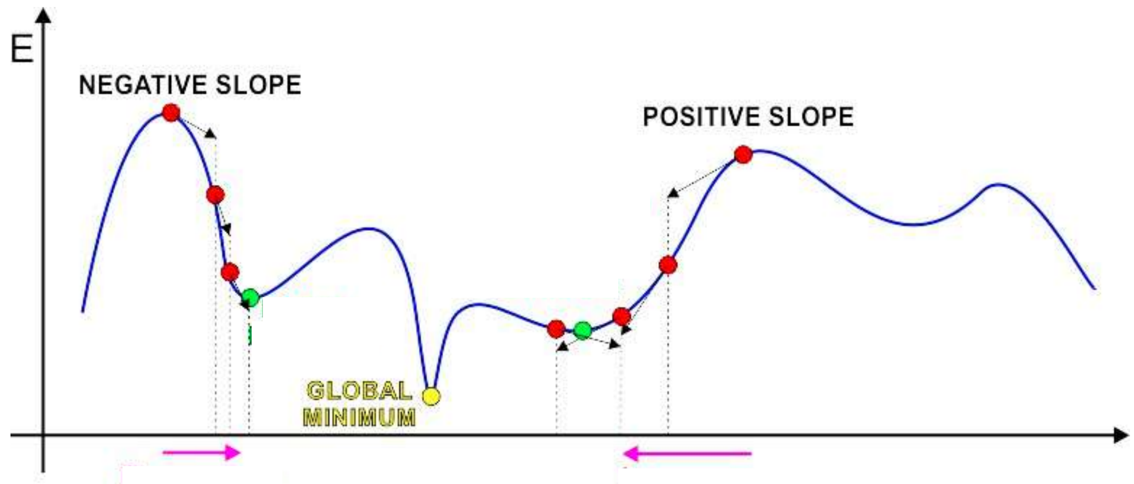
Iterative solutions





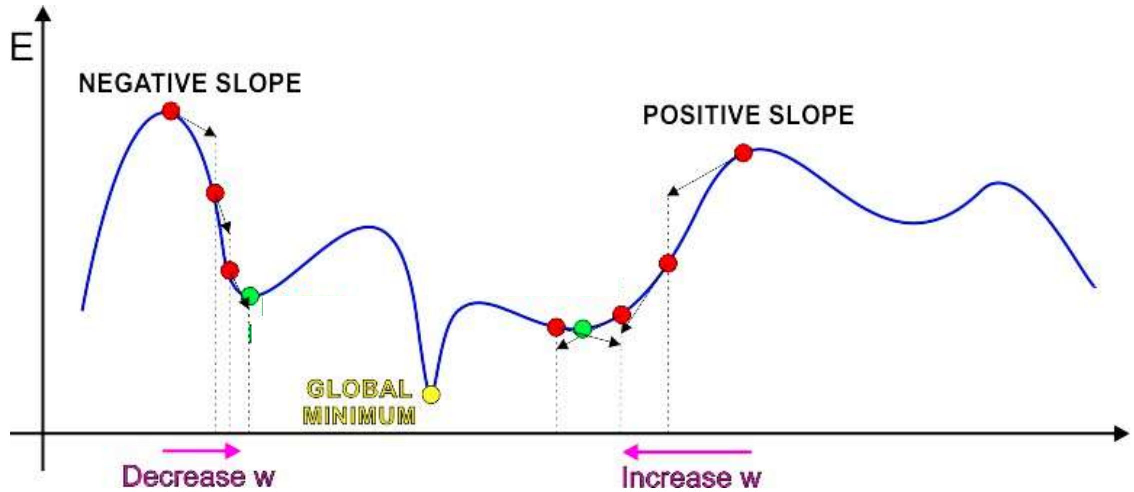
- Iterative solutions
  - Start from an initial guess X_0 for the optimal X
  - Update the guess towards a (hopefully) “better” value of $\mathrm{f}(X)$
  - Stop when $\mathrm{f}(X)$ no longer decreases
- Problems:
  - Which direction to step in
  - How big must the steps be

The Approach of Gradient Descent



- Iterative solution:
  – Start at some point
  – Find direction in which to shift this point to decrease error
    - This can be found from the derivative of the function
      – A negative derivative → moving right decreases error
      – A positive derivative → moving left decreases error
  – Shift point in this direction

The Approach of Gradient Descent



- Iterative solution: Trivial algorithm
  - Initialize x^0
  - While $f'(x^k) \neq 0$

$$x^{k+1} = x^k - \eta^k f'(x^k)$$

- η^k is the "step size"

Gradient descent/ascent (multivariate)

- The gradient descent/ascent method to find the minimum or maximum of a function f iteratively
  - To find a *maximum* move *in the direction of the gradient*

$$x^{k+1} = x^k + \eta^k \nabla_x f\left(x^k\right)^T$$

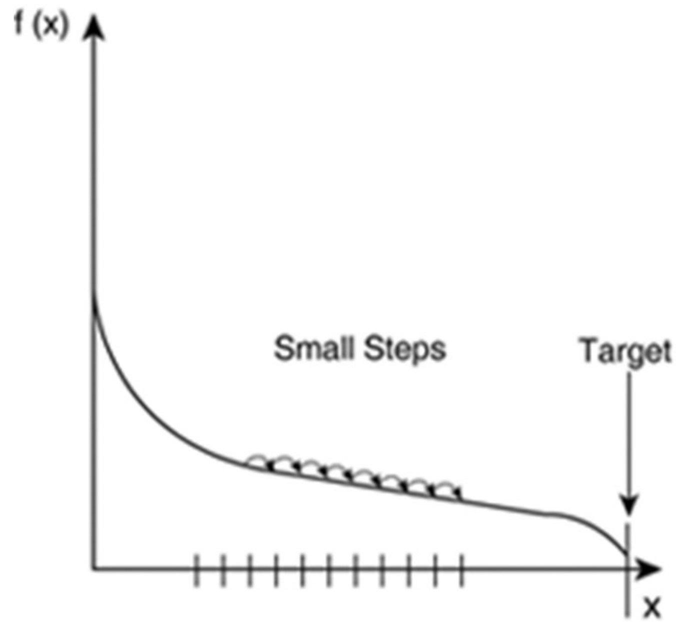
  - To find a *minimum* move *exactly opposite the direction of the gradient*

$$x^{k+1} = x^k - \eta^k \nabla_x f\left(x^k\right)^T$$

- Many solutions to choosing step size η^k

1. Fixed step size

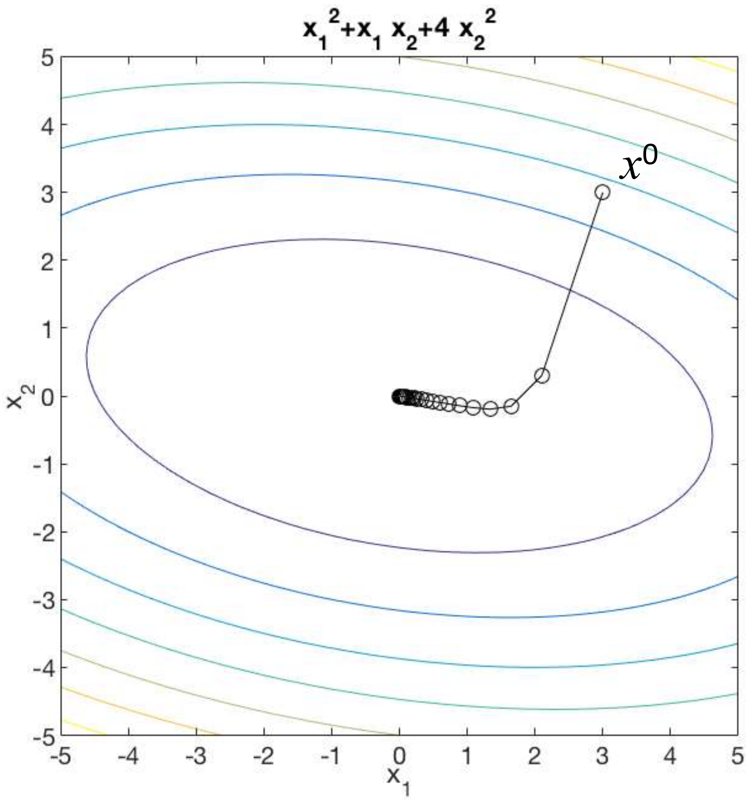
- Fixed step size
  - Use fixed value for η^k



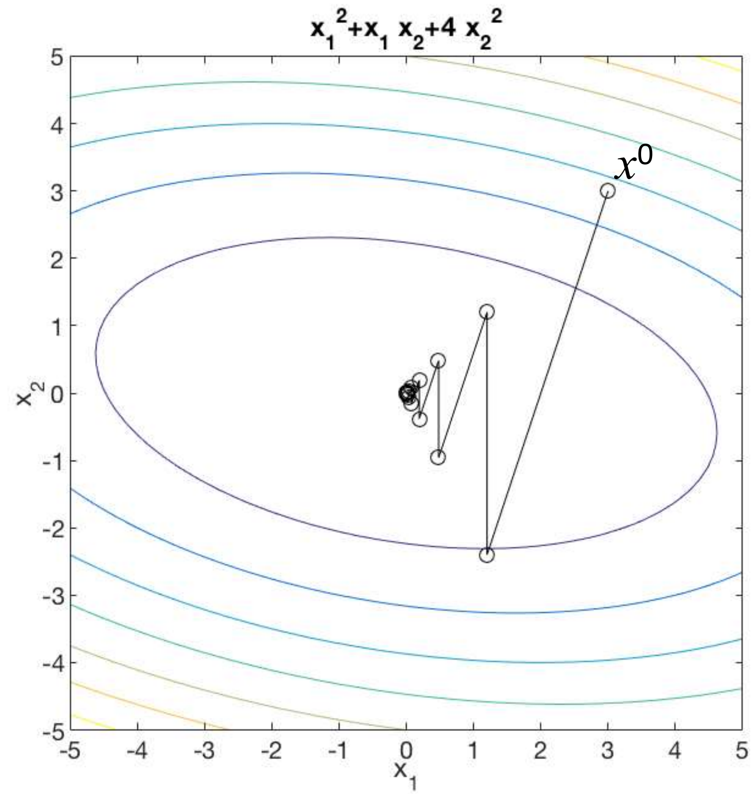
Influence of step size example (constant step size)

$$f(x_1, x_2) = (x_1)^2 + x_1 x_2 + 4(x_2)^2 \qquad x^{initial} = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$$

$\eta = 0.1$



$\eta = 0.2$



What is the optimal step size?

- Step size is critical for fast optimization
- Will revisit this topic later
- For now, simply assume a potentially-iteration-dependent step size

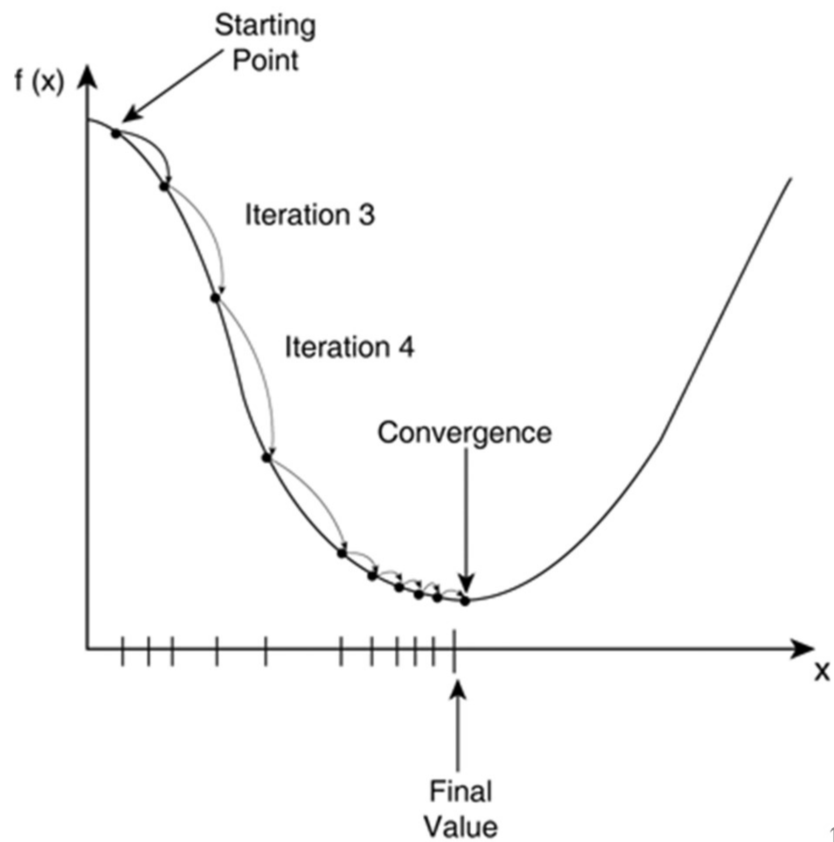
Gradient descent convergence criteria

- The gradient descent algorithm converges when one of the following criteria is satisfied

$$\left| f(x^{k+1}) - f(x^k) \right| < \varepsilon_1$$

- Or

$$\left\| \nabla_x f(x^k) \right\| < \varepsilon_2$$



Overall Gradient Descent Algorithm

- Initialize:
  - x^0
  - $k = 0$

- do
  - $x^{k+1} = x^k - \eta^k \nabla_x f\left(x^k\right)^T$
  - $k = k + 1$
- while $\left|f\left(x^{k+1}\right) - f\left(x^k\right)\right| > \varepsilon$

Problem Statement

- Given a training set of input-output pairs $(\boldsymbol{X}_1, \boldsymbol{d}_1), (\boldsymbol{X}_2, \boldsymbol{d}_2), \ldots, (\boldsymbol{X}_T, \boldsymbol{d}_T)$

- Minimize the following function

$$Err(W) = \frac{1}{T}\sum_i div(f(X_i; W), d_i)$$

w.r.t W

- This is problem of function minimization
  - An instance of optimization

Problem Setup: Things to define

- Given a training set of input-output pairs $(\boldsymbol{X}_1, \boldsymbol{d}_1), (\boldsymbol{X}_2, \boldsymbol{d}_2), \ldots, (\boldsymbol{X}_T, \boldsymbol{d}_T)$

- What are these input-output pairs?

$$Err(W) = \frac{1}{T}\sum_i div(f(X_i; W), d_i)$$

Problem Setup: Things to define

- Given a training set of input-output pairs $(\boldsymbol{X}_1, \boldsymbol{d}_1), (\boldsymbol{X}_2, \boldsymbol{d}_2), \dots, (\boldsymbol{X}_T, \boldsymbol{d}_T)$
- What are these input-output pairs?

$$Err(W) = \frac{1}{T}\sum_i div(f(X_i; W), d_i)$$

What is f() and what are its parameters W?

Problem Setup: Things to define

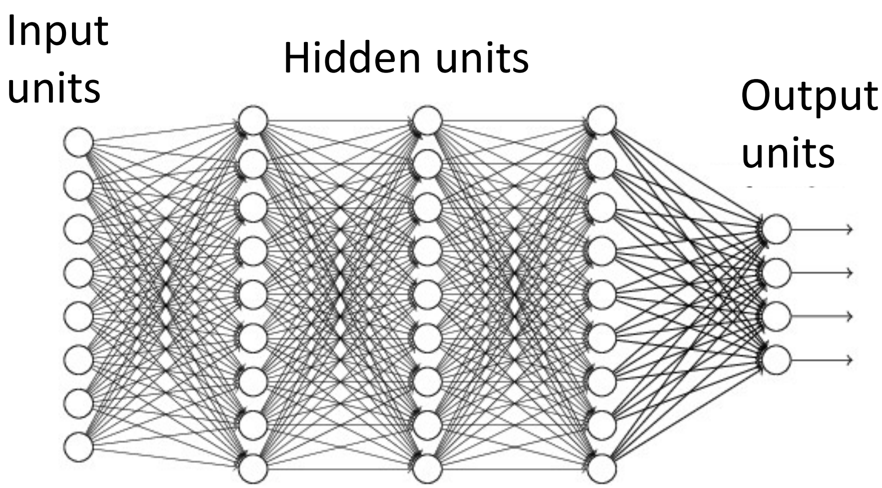
- Given a training set of input-output pairs $(\boldsymbol{X}_1, \boldsymbol{d}_1), (\boldsymbol{X}_2, \boldsymbol{d}_2), \ldots, (\boldsymbol{X}_T, \boldsymbol{d}_T)$
- What are these input-output pairs?

$$Err(W) = \frac{1}{T}\sum_i div(f(X_i; W), d_i)$$

What is the divergence div()?

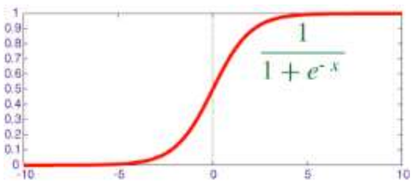
What is f() and what are its parameters W?

What is f()? Typical network



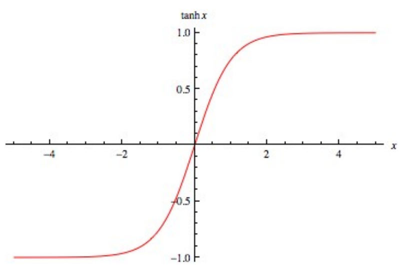
- Multi-layer perceptron
- A *directed* network with a set of inputs and outputs
  - No loops
- Generic terminology
  - We will refer to the inputs as the *input units*
    - **No neurons here – the “input units” are just the inputs**
  - We refer to the outputs as the output units
  - Intermediate units are “hidden” units

Activations and their derivatives



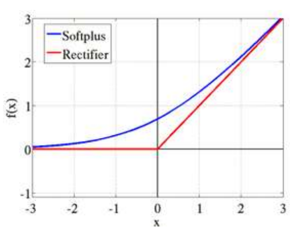
$$f(z) = \frac{1}{1+\exp(-z)}$$

$$f'(z) = f(z)(1-f(z))$$



$$f(z) = \tanh(z)$$

$$f'(z) = (1 - f^2(z))$$



$$f(z) = \begin{cases} 0, & z < 0 \\ z, & z \geq 0 \end{cases}$$

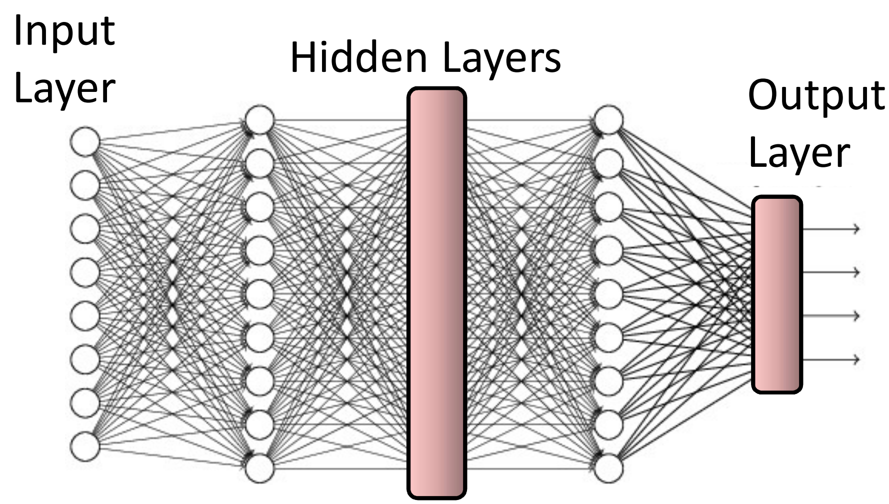
[*] $$f'(z) = \begin{cases} 1, z \geq 0 \\ 0, z < 0 \end{cases}$$

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

$$f'(z) = \frac{1}{1+\exp(-z)}$$

- Some popular activation functions and their derivatives

Vector Activations



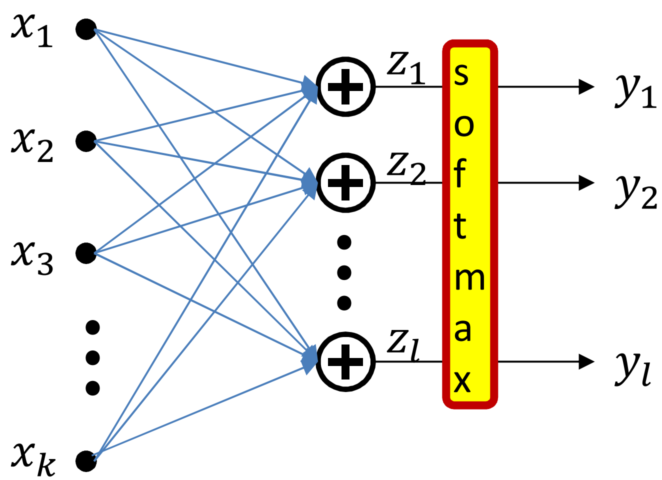
- We can also have neurons that have *multiple coupled* outputs

$$[y_1, y_2, \dots, y_l] = f(x_1, x_2, \dots, x_k; W)$$

  – Function $f()$ operates on set of inputs to produce set of outputs

  – Modifying a single parameter in W will affect *all* outputs

Vector activation example: Softmax



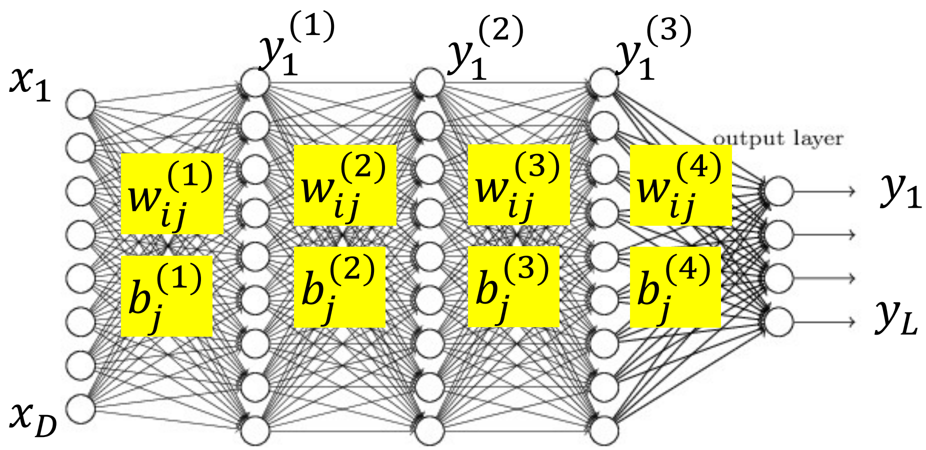
- Example: Softmax *vector* activation

$$z_i = \sum_j w_{ji} x_j + b_i$$

$$y = \frac{exp(z_i)}{\sum_j exp(z_j)}$$

Parameters are weights w_{ji} and bias b_i

Notation



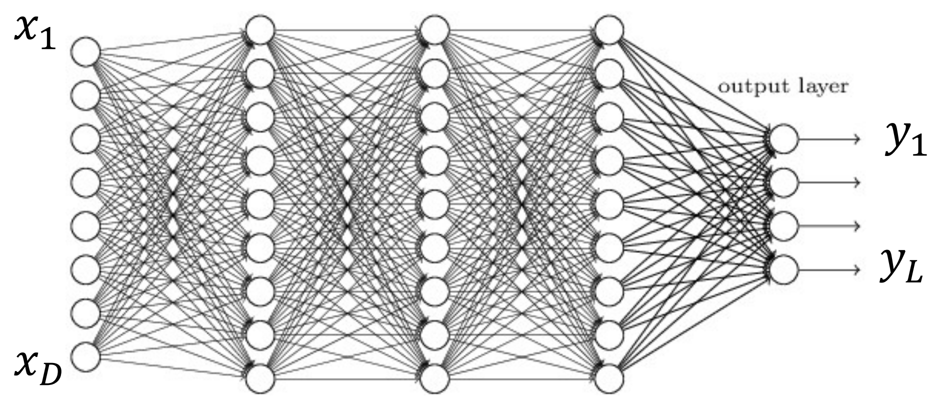
- The input layer is the 0th layer
- We will represent the output of the i-th perceptron of the kth layer as $y_i^{(k)}$
  - **Input to network:** $y_i^{(0)} = x_i$
  - **Output of network:** $y_i = y_i^{(N)}$
- We will represent the weight of the connection between the i-th unit of the k-1th layer and the jth unit of the k-th layer as $w_{ij}^{(k)}$
  - The bias to the jth unit of the k-th layer is $b_j^{(k)}$

Problem Setup: Things to define

- Given a training set of input-output pairs $(\boldsymbol{X}_1, \boldsymbol{d}_1), (\boldsymbol{X}_2, \boldsymbol{d}_2), \ldots, (\boldsymbol{X}_T, \boldsymbol{d}_T)$
- What are these input-output pairs?

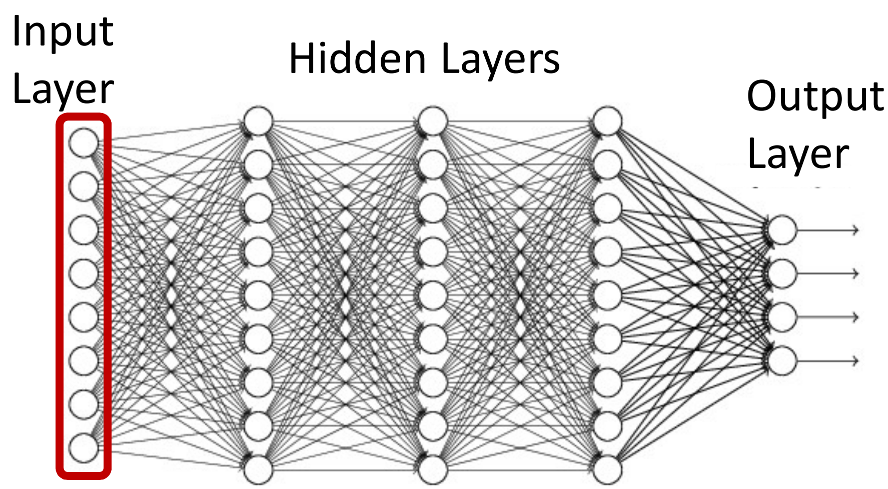
$$Err(W) = \frac{1}{T} \sum_i div(f(X_i; W), d_i)$$

Vector notation



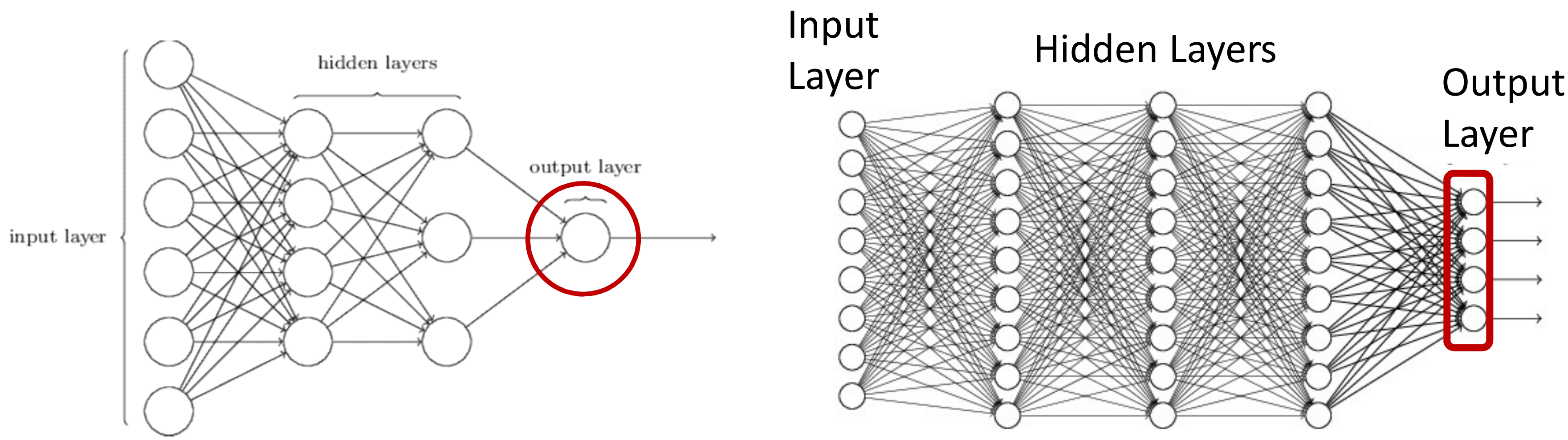
- Given a training set of input-output pairs $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- $X_n = [x_{n1}, x_{n2}, \dots, x_{nD}]$ is the nth input vector
- $d_n = [d_{n1}, d_{n2}, \dots, d_{nL}]$ is the nth desired output
- $Y_n = [y_{n1}, y_{n2}, \dots, y_{nL}]$ is the nth vector of *actual* outputs of the network
- We will sometimes drop the first subscript when referring to a *specific* instance

Representing the input



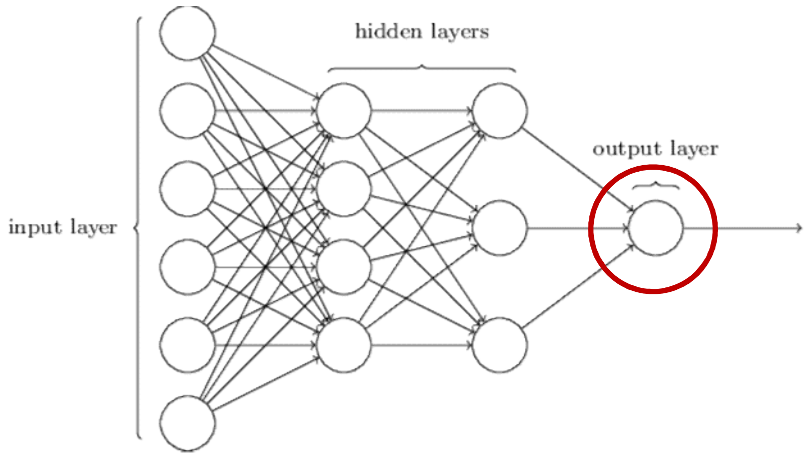
- Vectors of numbers
  - (or may even be just a scalar, if input layer is of size 1)
  - E.g. vector of pixel values
  - E.g. vector of speech features
  - E.g. real-valued vector representing text
    - We will see how this happens later in the course
  - Other real valued vectors

Representing the output



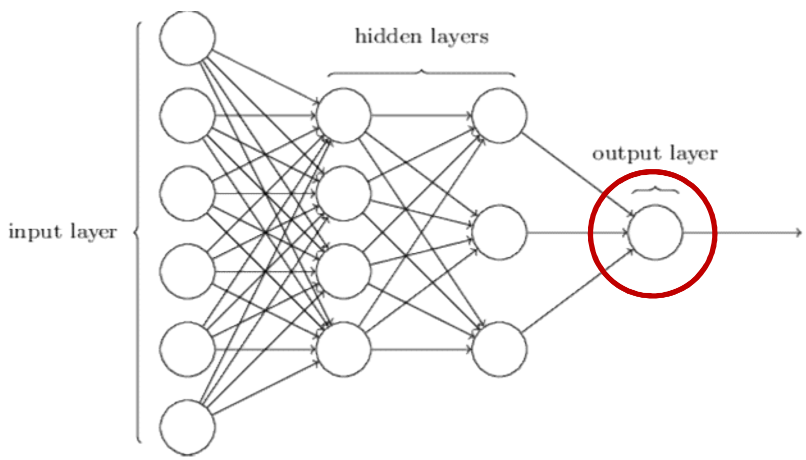
- If the desired *output* is real-valued, no special tricks are necessary
  - Scalar Output : single output neuron
    - d = scalar (real value)
  - Vector Output : as many output neurons as the dimension of the desired output
    - $d = [d_1\ d_2\ ..\ d_L]$ (vector of real values)

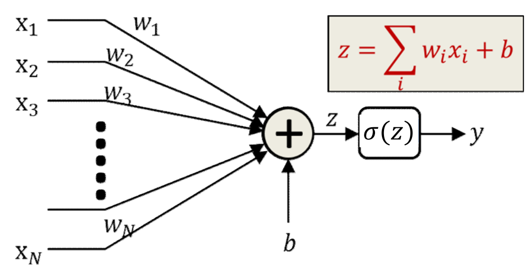
Representing the output

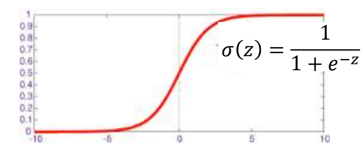


- If the desired output is binary (is this a cat or not), use a simple 1/0 representation of the desired output
  - 1 = Yes it’s a cat
  - 0 = No it’s not a cat.

Representing the output

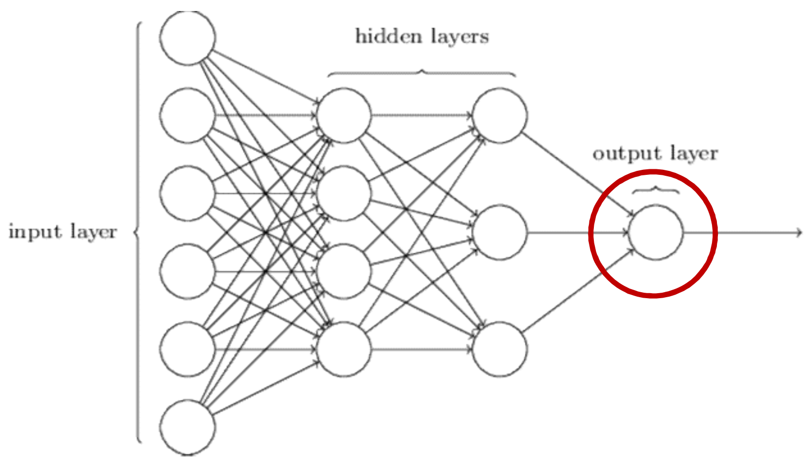


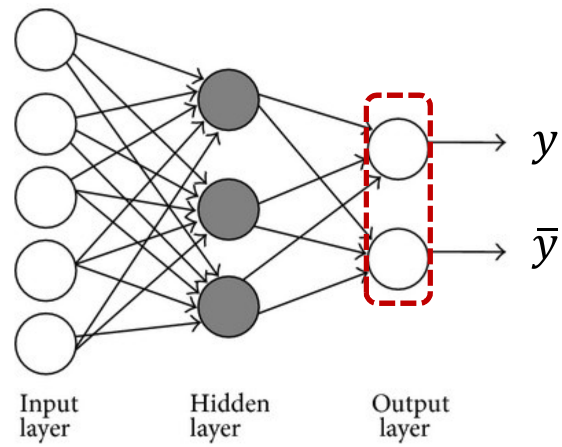




- If the desired output is binary (is this a cat or not), use a simple 1/0 representation of the desired output
- Output activation: Typically a sigmoid
  - Viewed as the *probability* $P(Y = 1|X)$ of class value 1
    - Indicating the fact that for actual data, in general a feature value X may occur for both classes, but with different probabilities
    - Is differentiable

Representing the output





- If the desired output is binary (is this a cat or not), use a simple 1/0 representation of the desired output
  - 1 = Yes it's a cat
  - 0 = No it's not a cat.

- Sometimes represented by *two independent* outputs, one representing the desired output, the other representing the *negation* of the desired output
  - Yes: → [1 0]
  - No: → [0 1]

Multi-class output: One-hot representations

- Consider a network that must distinguish if an input is a cat, a dog, a camel, a hat, or a flower
- We can represent this set as the following vector:

  [cat dog camel hat flower]^T

- For inputs of each of the five classes the desired output is:

  cat: $[1\ 0\ 0\ 0\ 0]^T$

  dog: $[0\ 1\ 0\ 0\ 0]^T$

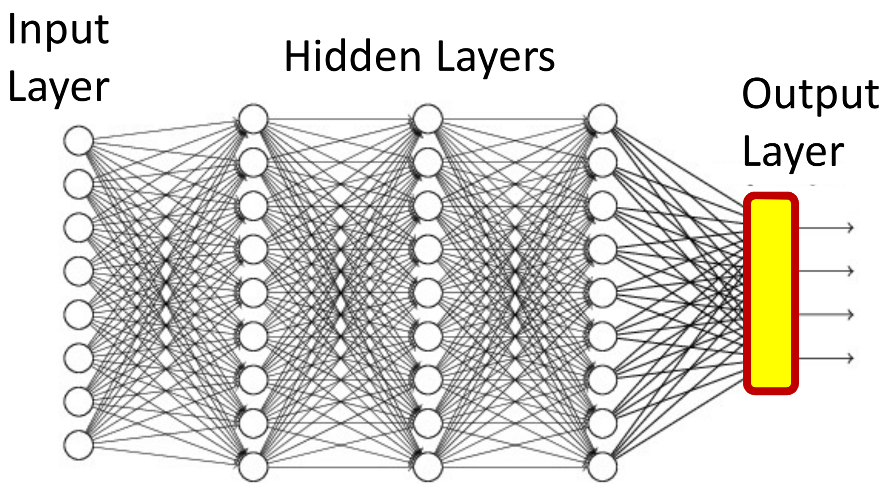
  camel: $[0\ 0\ 1\ 0\ 0]^T$

  hat: $[0\ 0\ 0\ 1\ 0]^T$

  flower: $[0\ 0\ 0\ 0\ 1]^T$

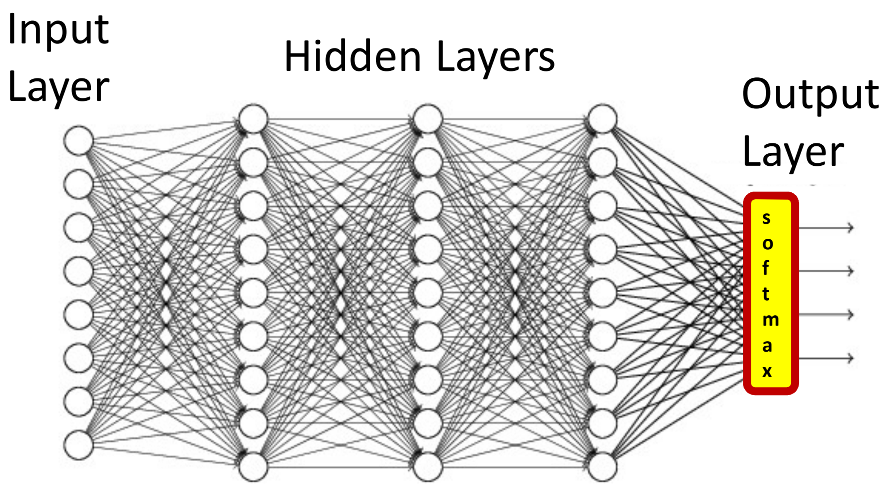
- For an input of any class, we will have a five-dimensional vector output with four zeros and a single 1 at the position of that class
- This is a *one hot vector*

Multi-class networks



- For a multi-class classifier with N classes, the one-hot representation will have N binary outputs
  - An N-dimensional binary vector
- The neural network's output too must ideally be binary (N-1 zeros and a single 1 in the right place)
- More realistically, it will be a probability vector
  - N probability values that sum to 1.

Multi-class classification: Output



- Softmax *vector* activation is often used at the output of multi-class classifier nets

$$z_i = \sum_j w_{ji}^{(n)} y_j^{(n-1)}$$

$$y_i = \frac{exp(z_i)}{\sum_j exp(z_j)}$$

- This can be viewed as the probability $y_i = P(class = i|X)$

Problem Setup: Things to define

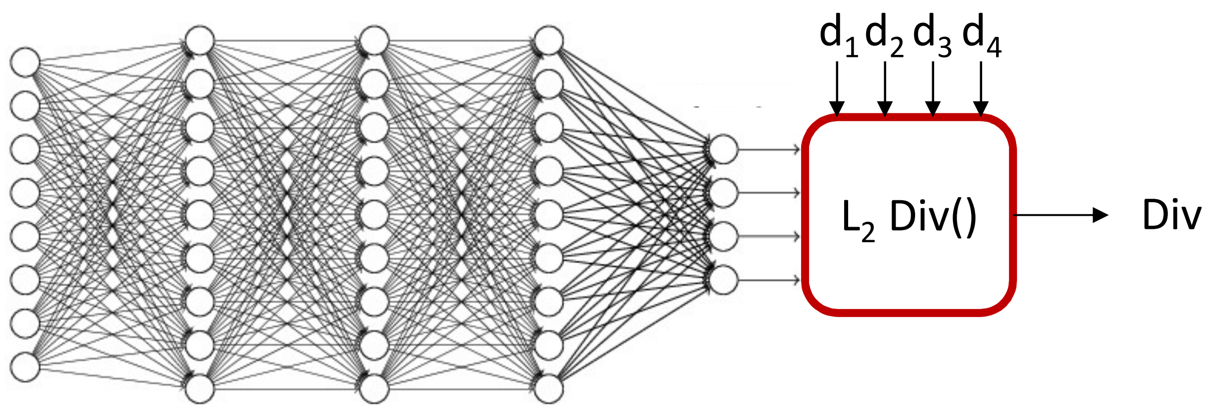
- Given a training set of input-output pairs $(\boldsymbol{X}_1, \boldsymbol{d}_1), (\boldsymbol{X}_2, \boldsymbol{d}_2), \ldots, (\boldsymbol{X}_T, \boldsymbol{d}_T)$

- Minimize the following function

$$Err(W) = \frac{1}{T}\sum_i div(f(X_i; W), d_i)$$

What is the divergence div()?

Examples of divergence functions



- For real-valued output vectors, the (scaled) L_2 divergence is popular

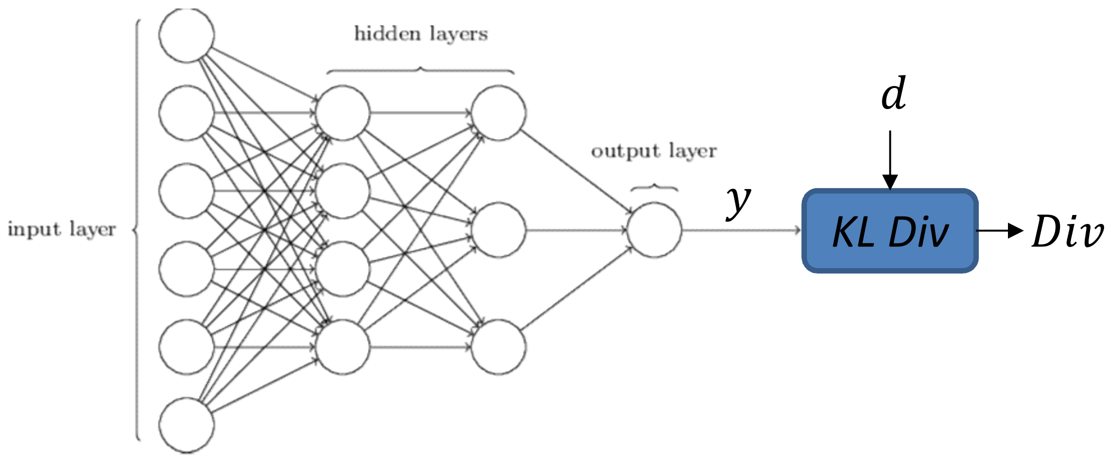
$$Div(Y,d) = \frac{1}{2}\|Y-d\|^2 = \frac{1}{2}\sum_i (y_i - d_i)^2$$

  - Squared Euclidean distance between true and desired output
  - Note: this is differentiable

$$\frac{dDiv(Y,d)}{dy_i} = (y_i - d_i)$$

$$\nabla_Y Div(Y,d) = [y_1 - d_1, y_2 - d_2, \dots]$$

For binary classifier



- For binary classifier with scalar output, Y ∈ $(0,1)$, d is $0/1$, the cross entropy between the probability distribution $[Y, 1-Y]$ and the ideal output probability $[d, 1-d]$ is popular

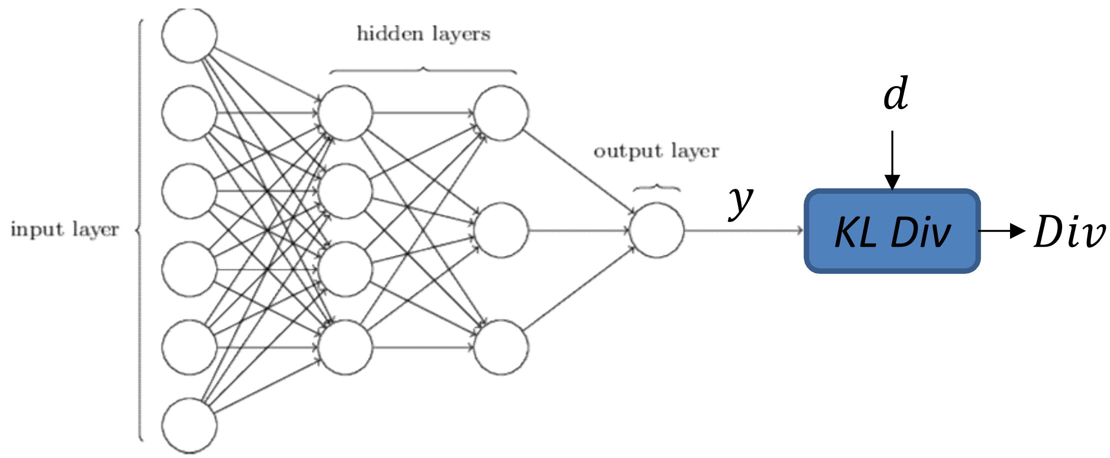
$$Div(Y,d) = -dlogY - (1-d)\log(1-Y)$$

  - Minimum when d $= Y$

- Derivative

$$\frac{dDiv(Y,d)}{dY} = \begin{cases} -\frac{1}{Y} & if \;\; d = 1 \\ \frac{1}{1-Y} & if \; d = 0 \end{cases}$$

For binary classifier



- For binary classifier with scalar output, $Y \in (0,1)$, d is $0/1$, the cross entropy between the probability distribution $[Y, 1 - Y]$ and the ideal output probability $[d, 1 - d]$ is popular

$$Div(Y, d) = -d log Y - (1 - d)\log(1 - Y)$$

  – Minimum when d $= Y$

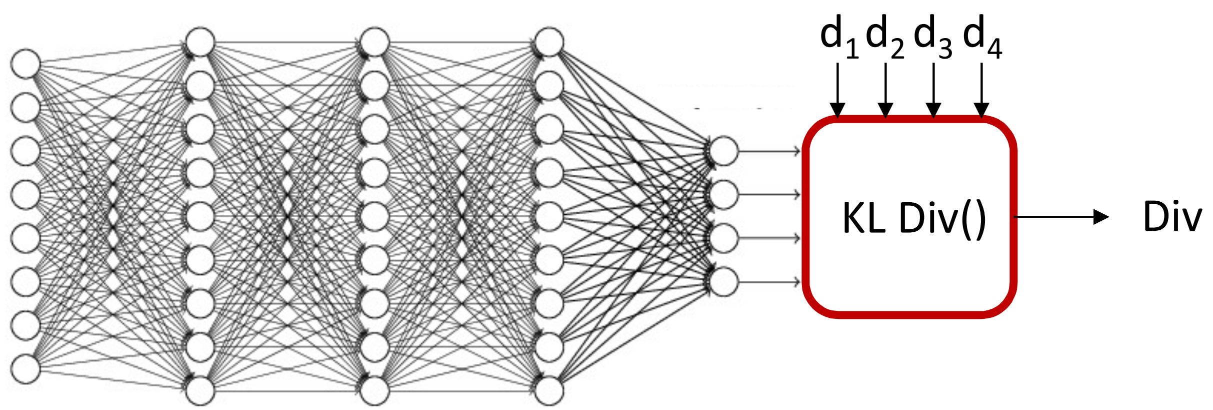
- Derivative

$$\frac{dDiv(Y,d)}{dY} = \begin{cases} -\frac{1}{Y} & if \ \ d = 1 \\ \frac{1}{1-Y} & if \ d = 0 \end{cases}$$

Note: when $y = d$ the derivative is *not* 0

Even though $div() = 0$ (minimum) *when y = d*

For multi-class classification



- Desired output d is a one hot vector $[0\ 0 \ldots 1\ \ \ldots 0\ 0\ 0\ ]$ with the 1 in the c-th position (for class c)
- Actual output will be probability distribution $[y_1, y_2, \ldots]$
- The cross-entropy between the desired one-hot output and actual output:

$$Div(Y,d) = -\sum_i d_i \log y_i = -\log y_c$$

- Derivative

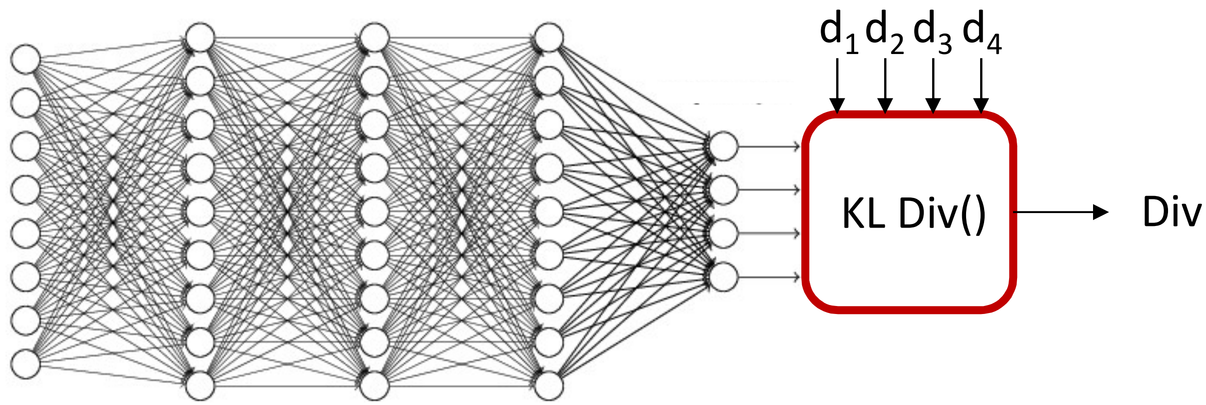
$$\frac{dDiv(Y,d)}{dY_i} = \begin{cases} -\frac{1}{y_c} & for\ the\ c-th\ component \\ 0 & for\ remaining\ component \end{cases}$$

$$\nabla_Y Div(Y,d) = \left[0\ 0\ \ldots \frac{-1}{y_c} \ldots 0\ 0\right]$$

If $y_c < 1$, the slope is negative w.r.t. y_c

Indicates *increasing* y_c will *reduce* divergence

For multi-class classification



- Desired output d is a one hot vector $[0\ 0 \ldots 1\ \ldots 0\ 0\ 0\,]$ with the 1 in the c-th position (for class c)
- Actual output will be probability distribution $[y_1, y_2, \ldots]$
- The cross-entropy between the desired one-hot output and actual output:

$$Div(Y, d) = -\sum_i d_i \log y_i = -\log y_c$$

- Derivative

$$\frac{dDiv(Y,d)}{dY_i} = \begin{cases} -\dfrac{1}{y_c} & for\ the\ c-th\ component \\ 0 & for\ remaining\ component \end{cases}$$

$$\nabla_Y Div(Y, d) = \left[0\ 0\ \ldots \frac{-1}{y_c} \ldots 0\ 0\right]$$

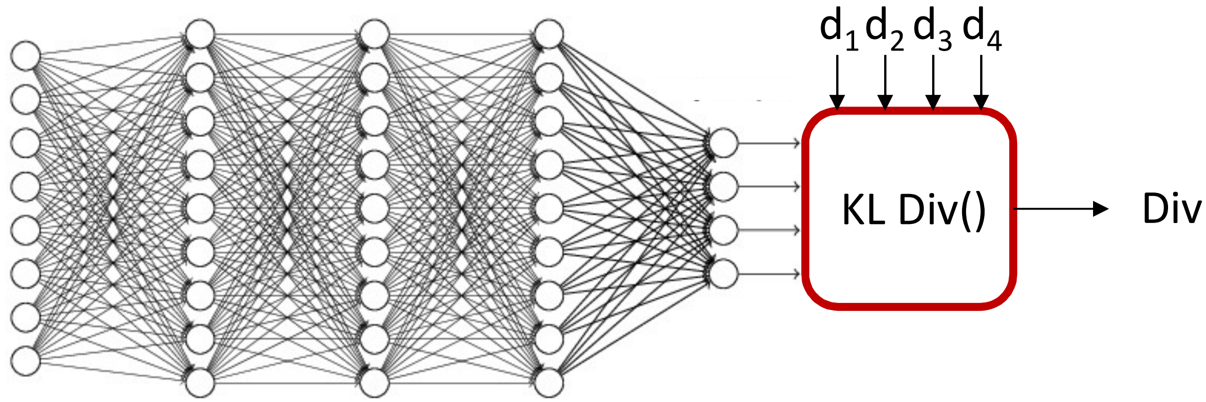
If $y_c < 1$, the slope is negative w.r.t. y_c

Indicates *increasing* y_c will *reduce* divergence

Note: when $y = d$ the derivative is *not* 0

Even though $div() = 0$ (minimum) *when y = d*

For multi-class classification



- It is sometimes useful to set the target output to $[\epsilon \;\; \epsilon \ldots (1-(K-1)\epsilon) \ldots \epsilon \;\; \epsilon \;\; \epsilon]$ with the value $1-(\mathrm{K}-1)\epsilon$ in the c-th position (for class c) and ϵ elsewhere for some small ϵ
  - "Label smoothing" -- aids gradient descent
- The cross-entropy remains:

$$Div(Y,d) = -\sum_i d_i \log y_i$$

- Derivative

$$\frac{dDiv(Y,d)}{dY_i} = \begin{cases} -\dfrac{1-(K-1)\epsilon}{y_c} & for\ the\ c-th\ component \\ -\dfrac{\epsilon}{y_i} for\ remaining\ components \end{cases}$$

Training Neural Nets through Gradient Descent

Total training error:

$$Err = \frac{1}{T}\sum_t Div(Y_t, d_t)$$

- Gradient descent algorithm:
- Initialize all weights and biases $\left\{w_{ij}^{(k)}\right\}$
  - Using the extended notation: the bias is also a weight
- Do:
  - For every layer k for all i, j, update:
    - $w_{i,j}^{(k)} = w_{i,j}^{(k)} - \eta \frac{dErr}{dw_{i,j}^{(k)}}$
- Until Err has converged

Assuming the bias is also represented as a weight

The derivative

Total training error:

$$Err = \frac{\mathbf{1}}{\boldsymbol{T}} \sum_{\boldsymbol{t}} Div(\boldsymbol{Y_t}, \boldsymbol{d_t})$$

- Computing the derivative

Total derivative:

$$\frac{dErr}{dw_{i,j}^{(k)}} = \frac{\mathbf{1}}{\boldsymbol{T}} \sum_{\boldsymbol{t}} \frac{dDiv(\boldsymbol{Y_t}, \boldsymbol{d_t})}{dw_{i,j}^{(k)}}$$

Training by gradient descent

- Initialize all weights $\left\{w_{ij}^{(k)}\right\}$
- Do:
  - For all $i, j, k,$ initialize $\frac{dEr}{dw_{i,j}^{(k)}} = 0$
  - For all $t = 1{:}T$
    - For every layer k for all i, j:
      - Compute $\frac{d\boldsymbol{Div}(\boldsymbol{Y_t}, \boldsymbol{d_t})}{dw_{i,j}^{(k)}}$
      - $\frac{dErr}{dw_{i,j}^{(k)}} += \frac{d\boldsymbol{Div}(\boldsymbol{Y_t}, \boldsymbol{d_t})}{dw_{i,j}^{(k)}}$
  - For every layer k for all i, j:

$$w_{i,j}^{(k)} = w_{i,j}^{(k)} - \frac{\eta}{T} \frac{dErr}{dw_{i,j}^{(k)}}$$

- Until Err has converged

The derivative

Total training error:

$$Err = \frac{\mathbf{1}}{\boldsymbol{T}} \sum_{\boldsymbol{t}} Div(\boldsymbol{Y}_{\boldsymbol{t}}, \boldsymbol{d}_{\boldsymbol{t}})$$

Total derivative:

$$\frac{d\boldsymbol{Err}}{dw_{i,j}^{(k)}} = \frac{\mathbf{1}}{\boldsymbol{T}} \sum_{\boldsymbol{t}} \frac{dDiv(\boldsymbol{Y}_{\boldsymbol{t}}, \boldsymbol{d}_{\boldsymbol{t}})}{dw_{i,j}^{(k)}}$$

- So we must first figure out how to compute the derivative of divergences of individual training inputs

Calculus Refresher: Chain rule

For any nested function $y = f(g(x))$

$$\boxed{\frac{dy}{dx} = \frac{\partial f}{\partial g(x)} \frac{dg(x)}{dx}}$$

Check – we can confirm that : $\Delta y = \dfrac{dy}{dx} \Delta x$

$z = g(x) \Rightarrow \Delta z = \dfrac{dg(x)}{dx} \Delta x$

$y = f(z) \Rightarrow \quad \Delta y = \dfrac{df}{dz} \Delta z = \dfrac{df}{dz} \dfrac{dg(x)}{dx} \Delta x$ ✓

Calculus Refresher: Distributed Chain rule

$$y = f\left(g_1(x), g_1(x), \dots, g_M(x)\right)$$

$$\boxed{\frac{dy}{dx} = \frac{\partial f}{\partial g_1(x)}\frac{dg_1(x)}{dx} + \frac{\partial f}{\partial g_2(x)}\frac{dg_2(x)}{dx} + \cdots + \frac{\partial f}{\partial g_M(x)}\frac{dg_M(x)}{dx}}$$

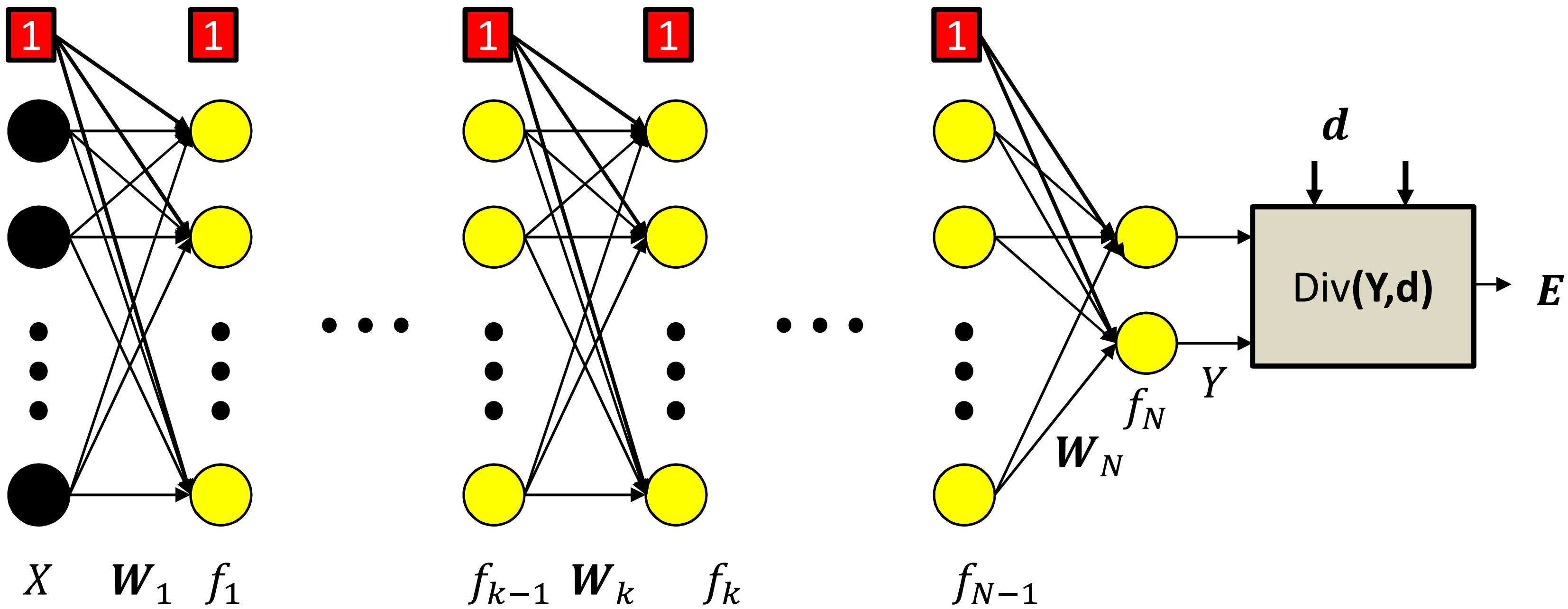
Check: $\Delta y = \frac{dy}{dx}\Delta x$

$$\Delta y = \frac{\partial f}{\partial g_1(x)}\Delta g_1(x) + \frac{\partial f}{\partial g_2(x)}\Delta g_2(x) + \cdots + \frac{\partial f}{\partial g_M(x)}\Delta g_M(x)$$

$$\boxed{\Delta y = \frac{\partial f}{\partial g_1(x)}\frac{dg_1(x)}{dx}\Delta x + \frac{\partial f}{\partial g_2(x)}\frac{dg_2(x)}{dx}\Delta x + \cdots + \frac{\partial f}{\partial g_M(x)}\frac{dg_M(x)}{dx}\Delta x}$$

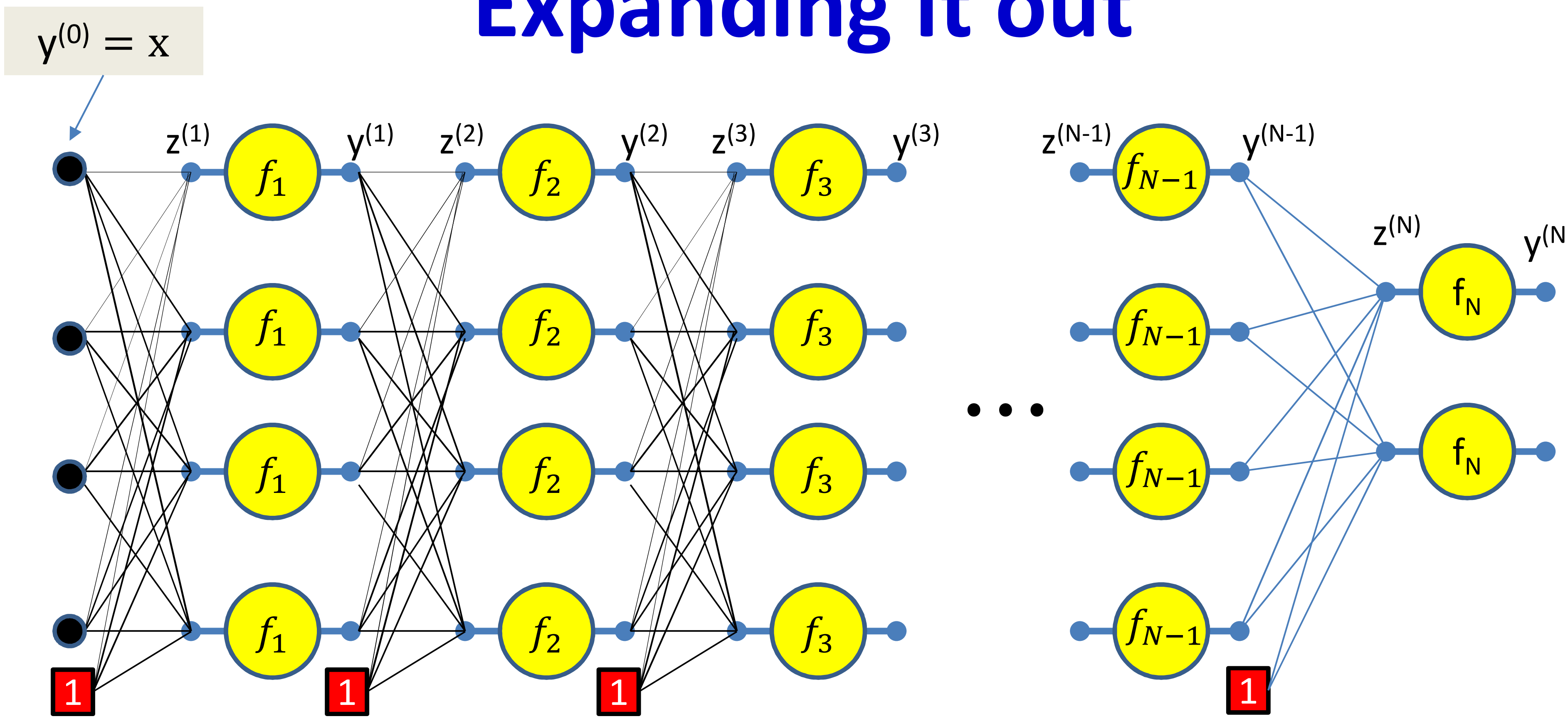
$$\boxed{\Delta y = \left(\frac{\partial f}{\partial g_1(x)}\frac{dg_1(x)}{dx} + \frac{\partial f}{\partial g_2(x)}\frac{dg_2(x)}{dx} + \cdots + \frac{\partial f}{\partial g_M(x)}\frac{dg_M(x)}{dx}\right)\Delta x}$$ ✓

BP: Scalar Formulation



- The network again

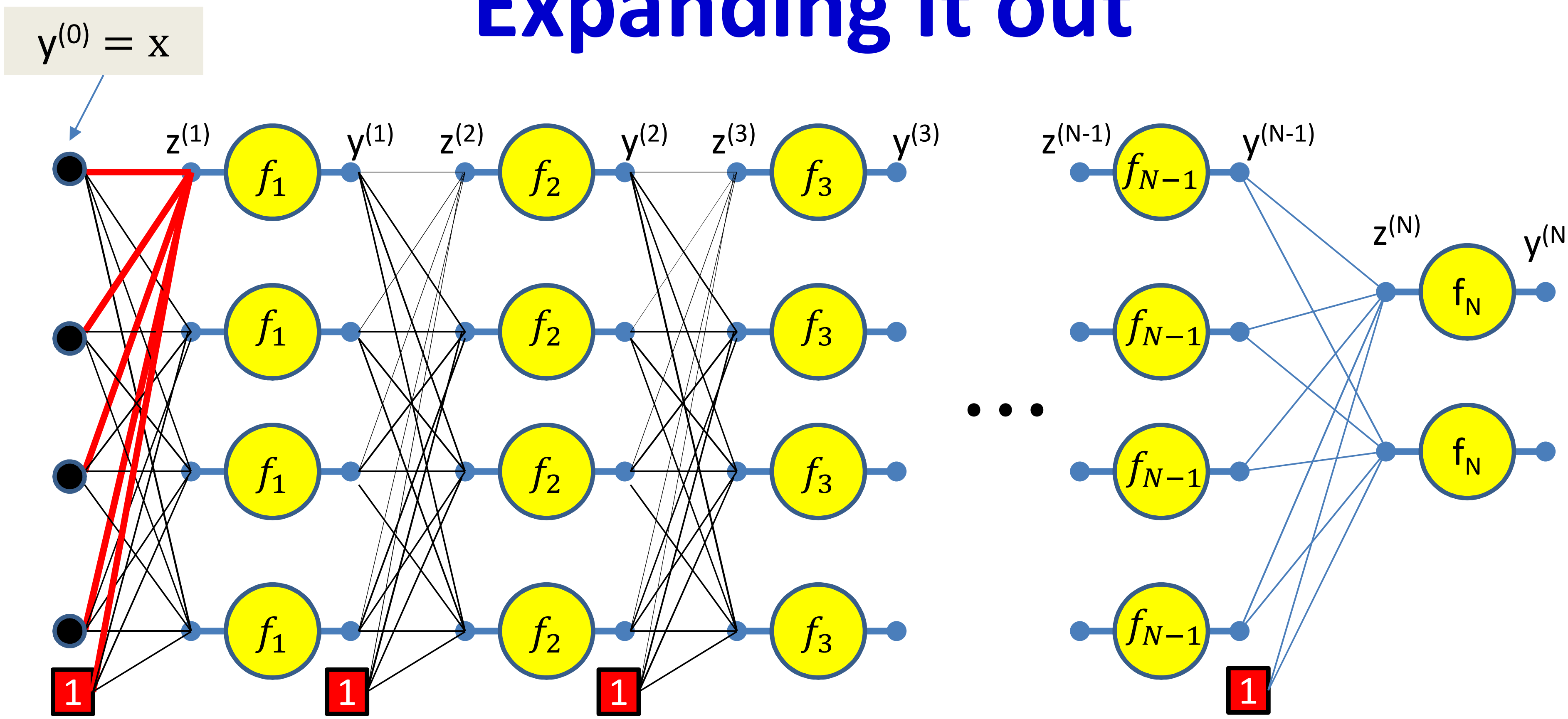
Expanding it out



Setting $y_i^{(0)} = x_i$ for notational convenience

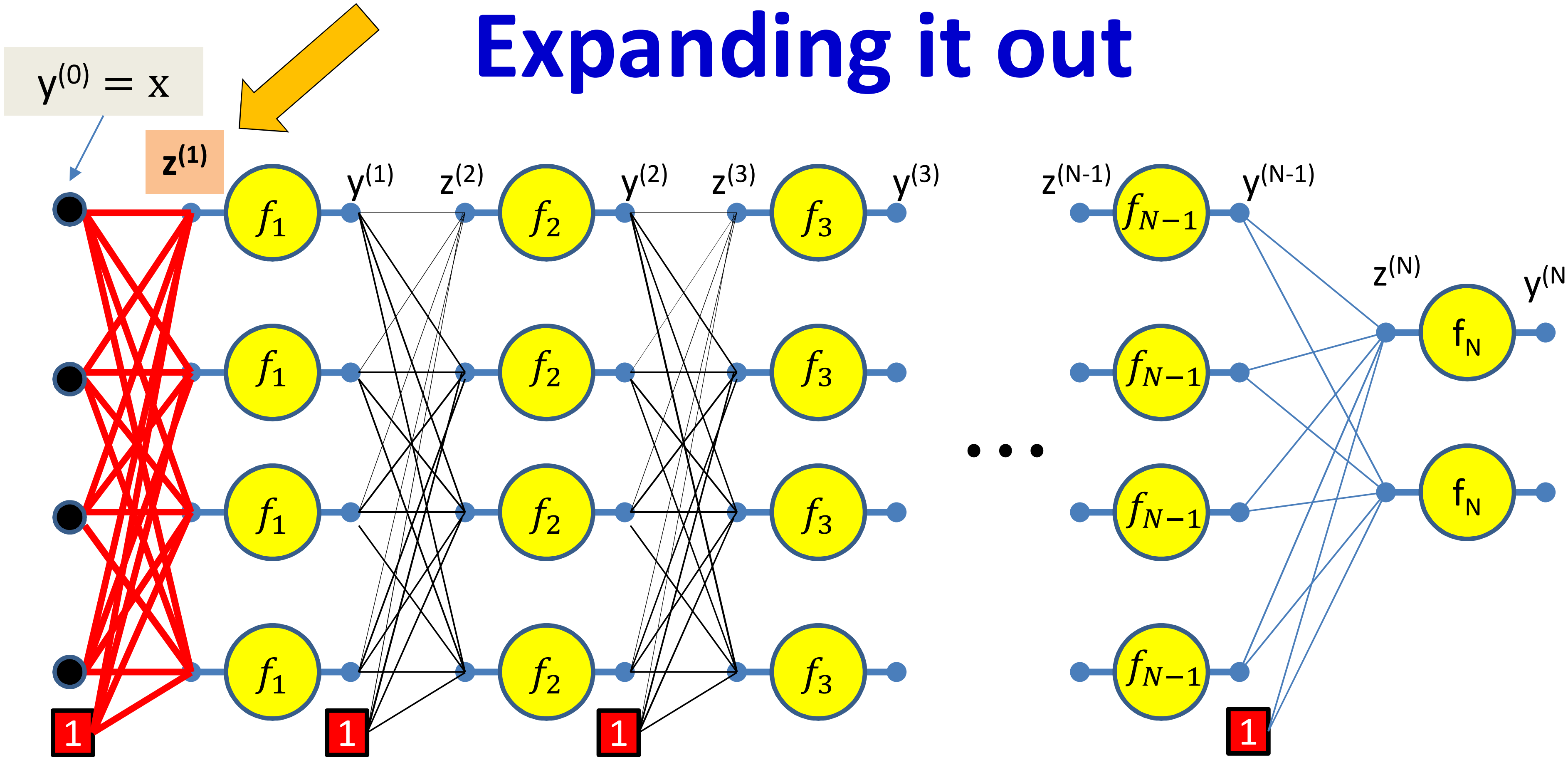
Assuming $w_{0j}^{(k)} = b_j^{(k)}$ and $y_0^{(k)} = 1$ -- assuming the bias is a weight and extending the output of every layer by a constant 1, to account for the biases

Expanding it out

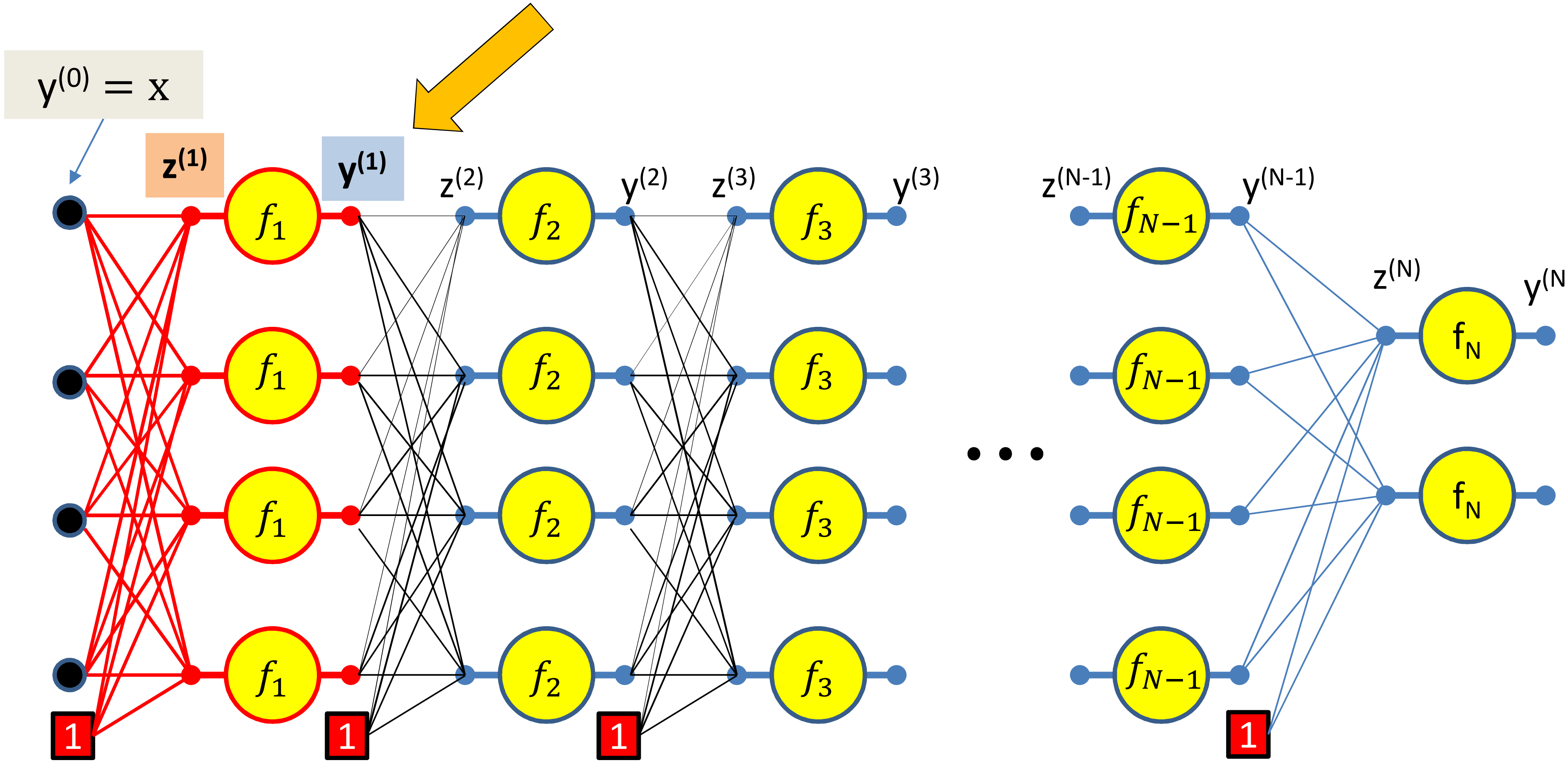


$$z_1^{(1)} = \sum_i w_{i1}^{(1)} y_i^{(0)}$$

Expanding it out

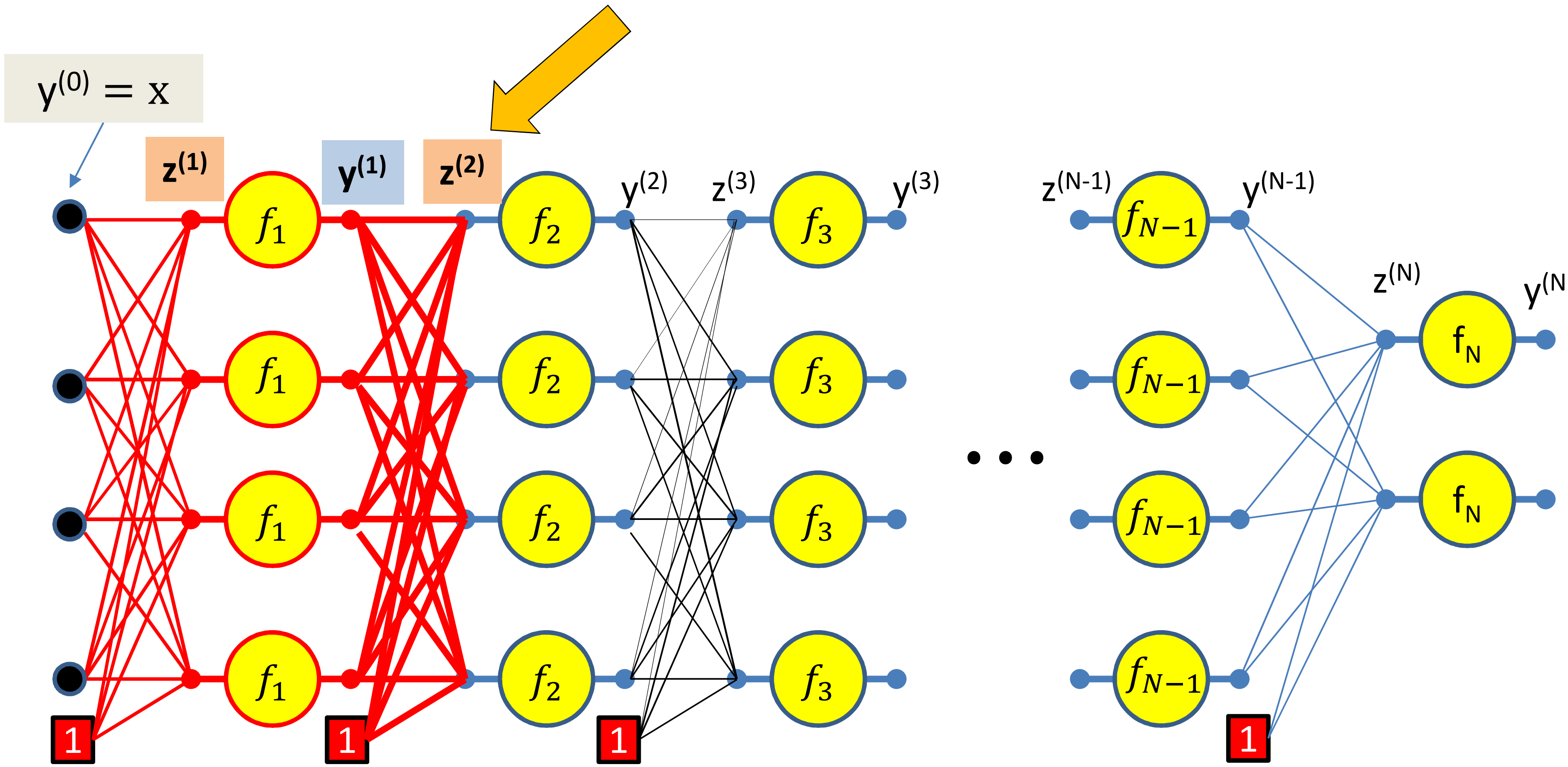


$$z_j^{(1)} = \sum_i w_{ij}^{(1)} y_i^{(0)}$$



$$z_j^{(1)} = \sum_i w_{ij}^{(1)} y_i^{(0)}$$

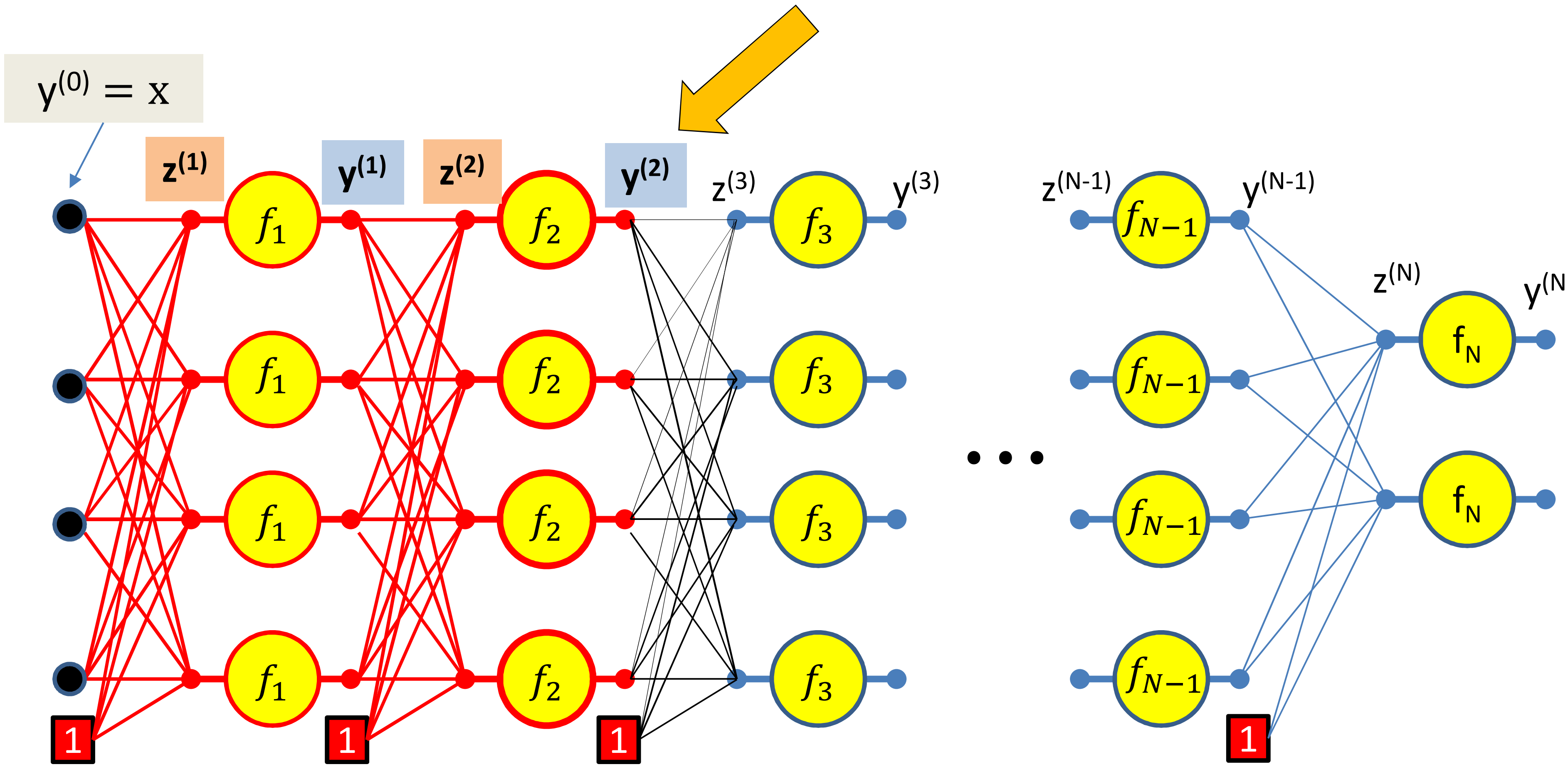
$$\boxed{y_j^{(1)} = f_1\left(z_j^{(1)}\right)}$$



$$z_j^{(1)} = \sum_i w_{ij}^{(1)} y_i^{(0)}$$

$$y_j^{(1)} = f_1\left(z_j^{(1)}\right)$$

$$\boxed{z_j^{(2)} = \sum_i w_{ij}^{(2)} y_i^{(1)}}$$

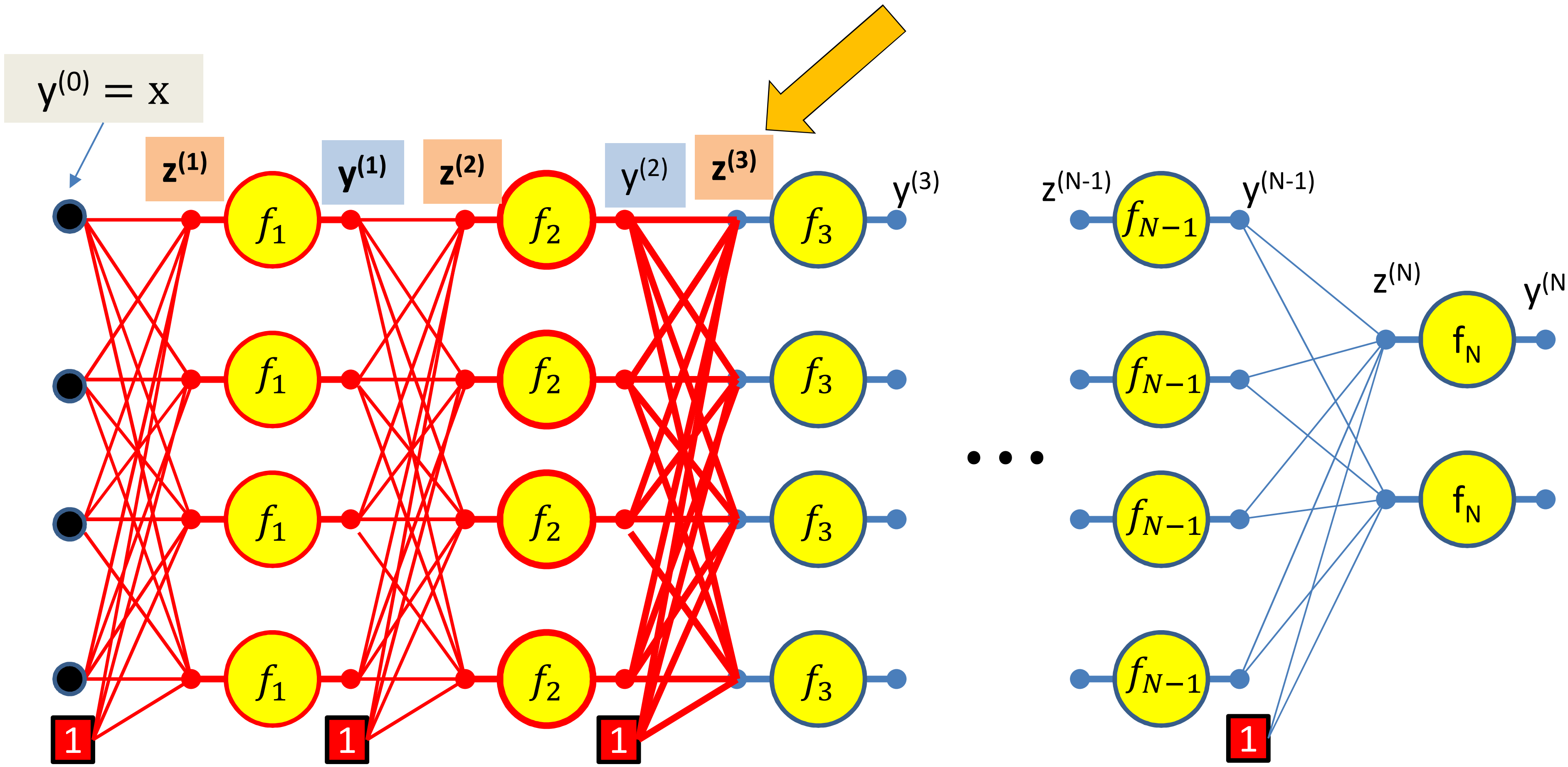


$$z_j^{(1)} = \sum_i w_{ij}^{(1)} y_i^{(0)}$$

$$y_j^{(1)} = f_1\left(z_j^{(1)}\right)$$

$$z_j^{(2)} = \sum_i w_{ij}^{(2)} y_i^{(1)}$$

$$\boxed{y_j^{(2)} = f_2\left(z_j^{(2)}\right)}$$



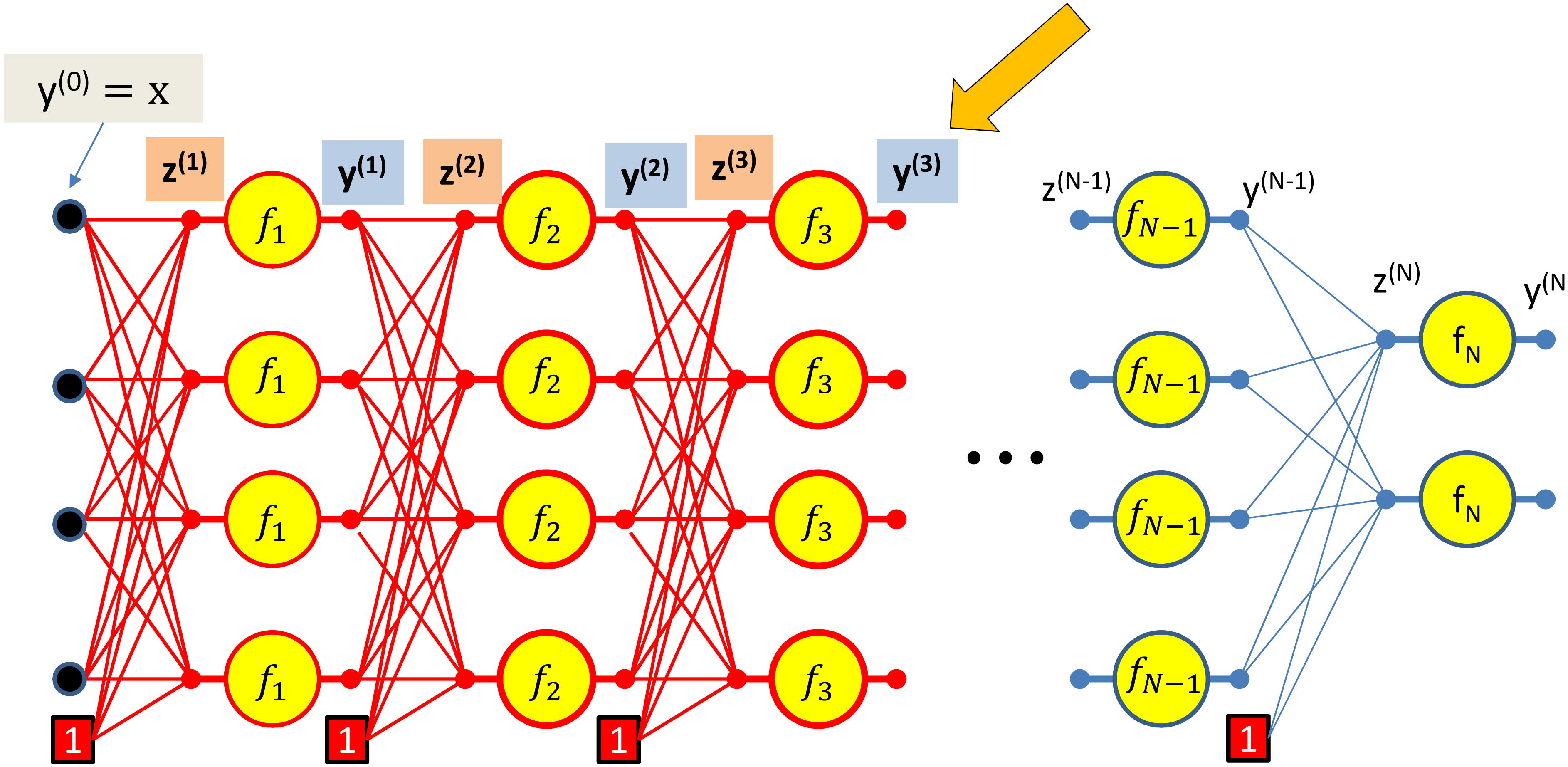
$$z_j^{(1)} = \sum_i w_{ij}^{(1)} y_i^{(0)}$$

$$y_j^{(1)} = f_1\left(z_j^{(1)}\right)$$

$$z_j^{(2)} = \sum_i w_{ij}^{(2)} y_i^{(1)}$$

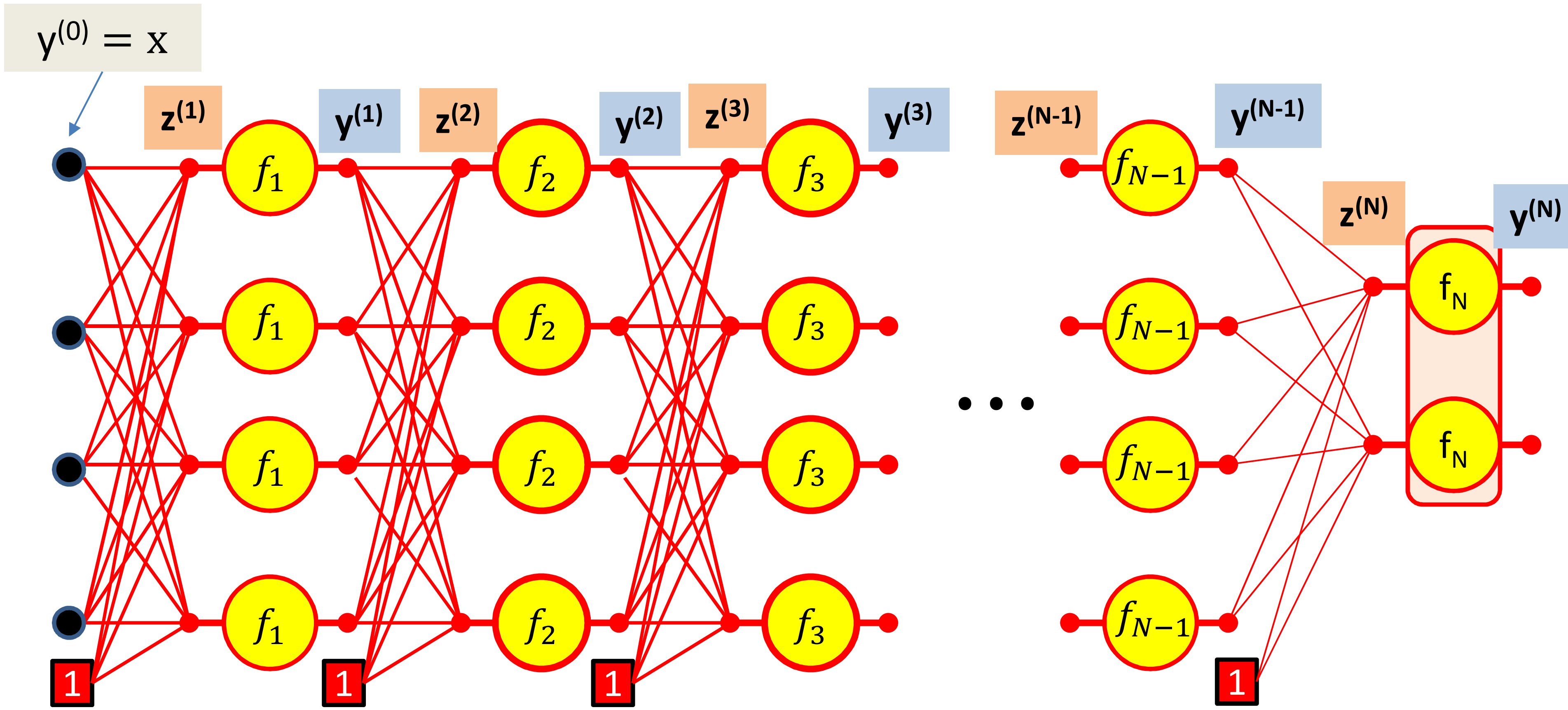
$$y_j^{(2)} = f_2\left(z_j^{(2)}\right)$$

$$\boxed{z_j^{(3)} = \sum_i w_{ij}^{(3)} y_i^{(2)}}$$



$$z_j^{(1)} = \sum_i w_{ij}^{(1)} y_i^{(0)} \qquad y_j^{(1)} = f_1\left(z_j^{(1)}\right) \qquad z_j^{(2)} = \sum_i w_{ij}^{(2)} y_i^{(1)} \qquad y_j^{(2)} = f_2\left(z_j^{(2)}\right)$$

$$z_j^{(3)} = \sum_i w_{ij}^{(3)} y_i^{(2)} \qquad \boxed{y_j^{(3)} = f_3\left(z_j^{(3)}\right)} \qquad \cdots$$



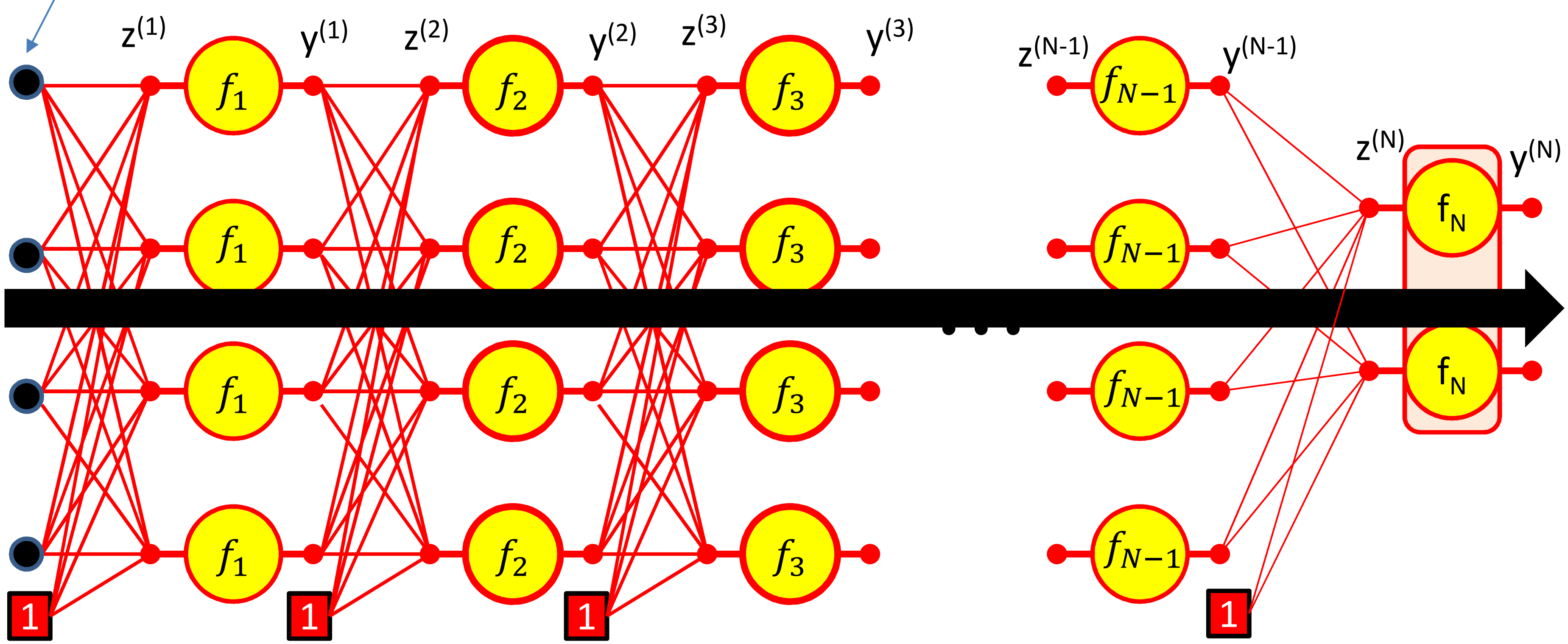
$$y_j^{(N-1)} = f_{N-1}\left(z_j^{(N-1)}\right)$$

$$z_j^{(N)} = \sum_i w_{ij}^{(N)} y_i^{(N-1)}$$

$$\boldsymbol{y}^{(N)} = f_N\left(\boldsymbol{z}^{(N)}\right)$$

Forward Computation

$y^{(0)} = x$



ITERATE FOR k = 1:N

for j = 1:layer-width

$$y_i^{(0)} = x_i$$

$$z_j^{(k)} = \sum_i w_{ij}^{(k)} y_i^{(k-1)}$$

$$y_j^{(k)} = f_k\left(z_j^{(k)}\right)$$

Forward "Pass"

- Input: D dimensional vector $\mathbf{x} = [x_j,\ j = 1 \dots D]$
- Set:
  - $D_0 = D,$ is the width of the 0th (input) layer
  - $y_j^{(0)} = x_j,\ j = 1 \dots D; \qquad y_0^{(k=1\dots N)} = x_0 = 1$
- For layer $k = 1 \dots N$
  - For $j = 1 \dots D_k$ — D_k is the size of the kth layer
    - $z_j^{(k)} = \sum_{i=0}^{D_{k-1}} w_{i,j}^{(k)} y_i^{(k-1)}$
    - $y_j^{(k)} = f_k\left(z_j^{(k)}\right)$
- Output:
  - $Y = y_j^{(N)}, j = 1..D_N$