03-Learning a Neural Network

CSE 849 Deep Learning

Spring 2025

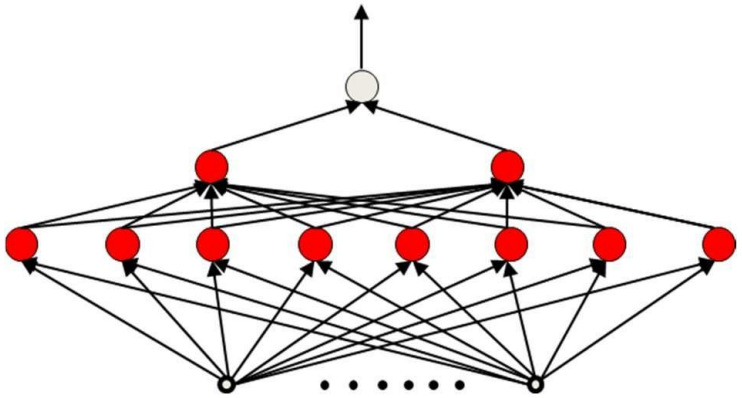
Zijun Cui

The slides have been revised based on CMU's 11-785 Introduction to Deep Learning course by Bhiksha Raj

Outline

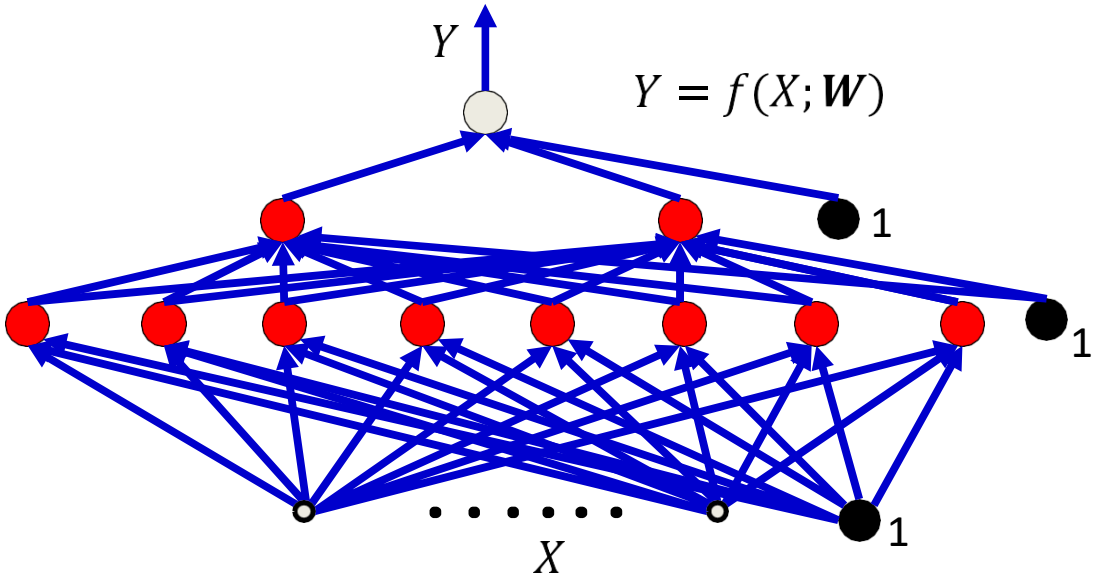
- Define the learning of a NN
- Understand Input/Output/Loss
- Forward and Backward Pass -- conceptually

The structure of the network



- We will assume a *feed-forward* network
  - No loops: Neuron outputs do not feed back to their inputs directly or indirectly
- Part of the design of a network: The architecture
  - How many layers/neurons, which neuron connects to which and how, etc.
- For now, assume the architecture of the network is capable of representing the needed function

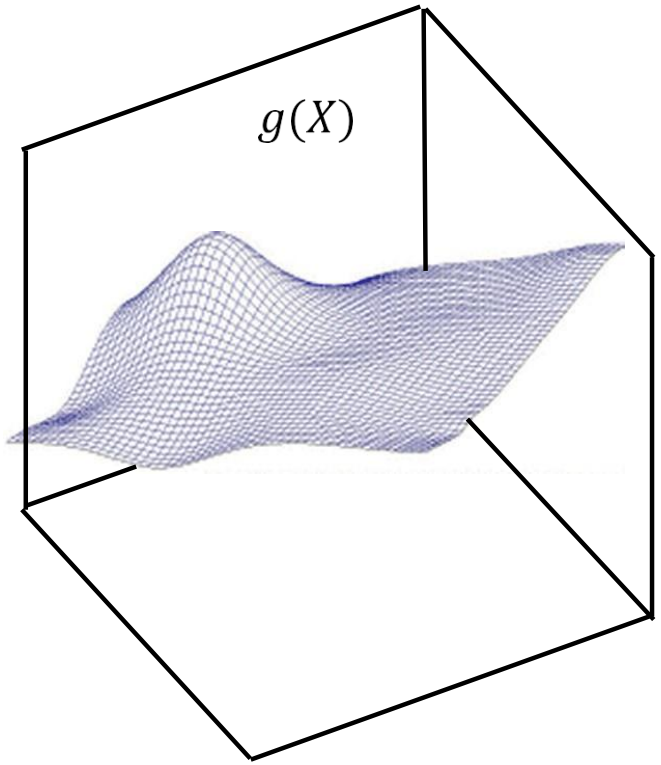
What we learn: The parameters of the network



- The parameters W of the network: The weights and biases
  - The weights associated with the blue arrows in the picture
- The network is a function f() with parameters W which must be set to the appropriate values to get the desired behavior from the net

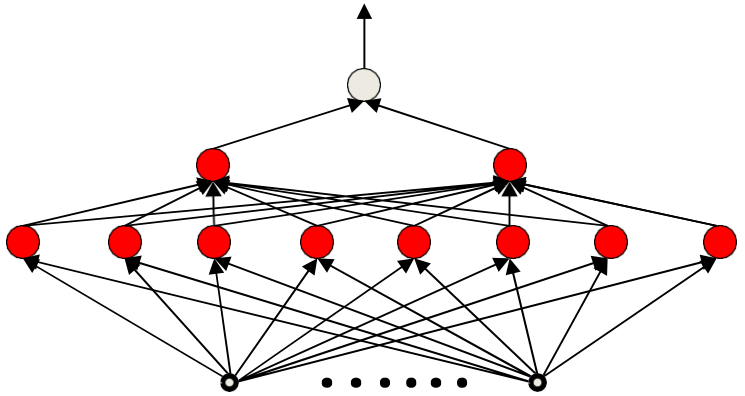
- **Given:** the architecture of the network
- ***Learning the network :*** **Determining the values of these parameters such that the network computes the desired function**

The MLP can represent anything



- The MLP *can be constructed* to represent anything
- Such as the function g(X)

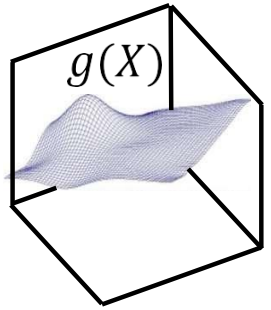
- But *how* do we construct it?

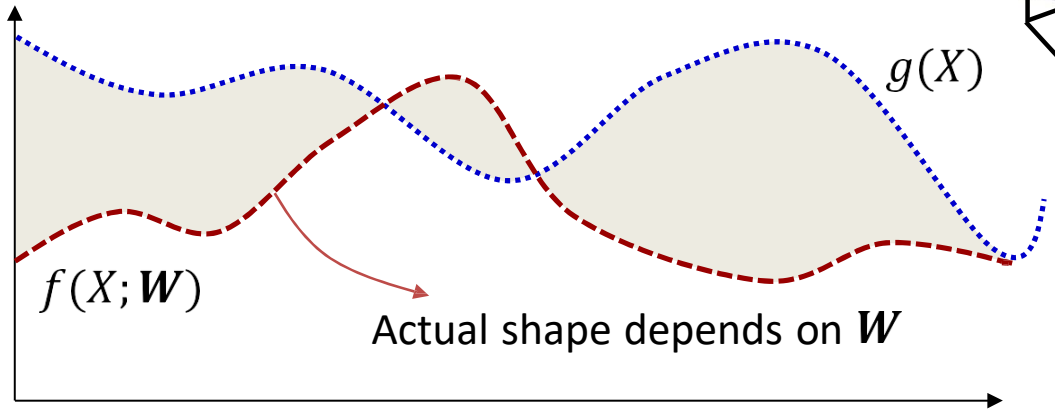


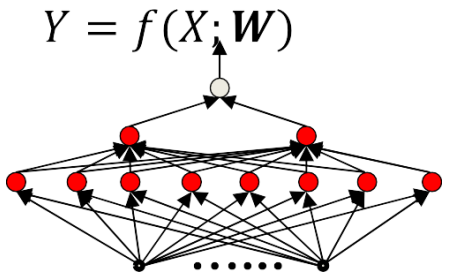
- Automatic estimation of an MLP
- By learning, we mean:

given the function g(X), we *derive* the parameters of the neural network to model it, through computation

How to learn a network?







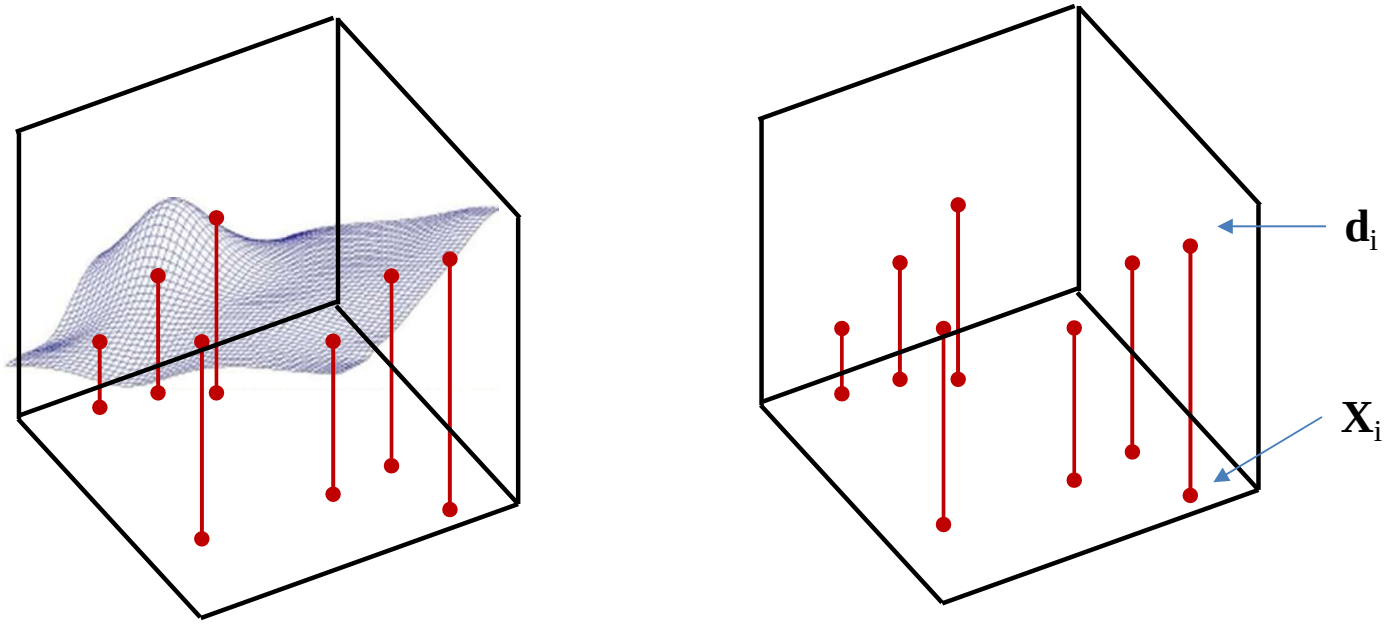
- Solution: Estimate parameters to minimize the error between the target function $g(X)$ and the network function $f(X;W)$
  - Find the parameter W that minimizes the shaded area
- Shaded area:

$$\int_{-\infty}^{\infty} error\big(f(X;\boldsymbol{W}), g(X)\big)dX$$

Issue:

- g(X) must be fully specify in order to compute
  - Fully specify: know g(X) for every X
- **In practice we will not have such specification**

Sampling the function



We must ***learn*** the *entire* function from these few examples

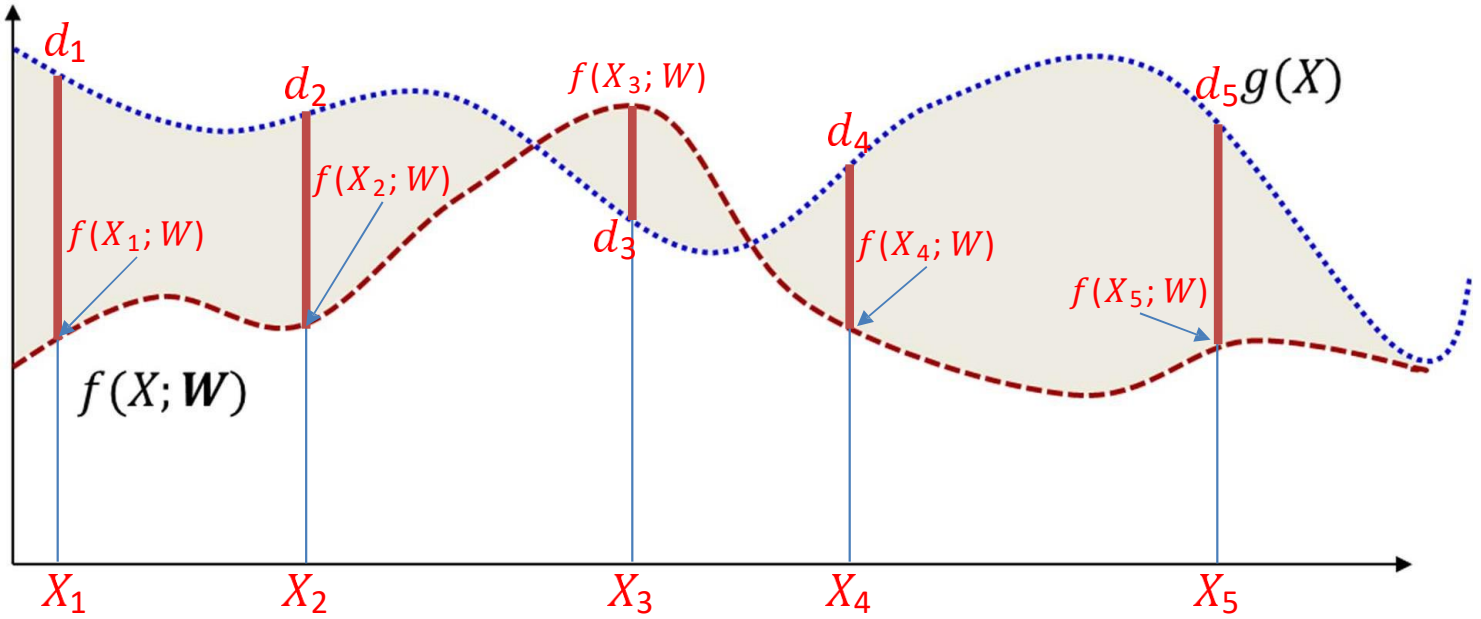
– The "training" samples

- *Sample* $g(X)$
  - Basically, get input-output pairs for a number of samples of input X_i
    - Many samples (X_i, d_i), where $d_i = g(X_i) + noise$

- Very easy to do in most problems: just gather training data
  - E.g. set of images and their class labels
  - E.g. speech recordings and their transcription

Today's Star

The Empirical error



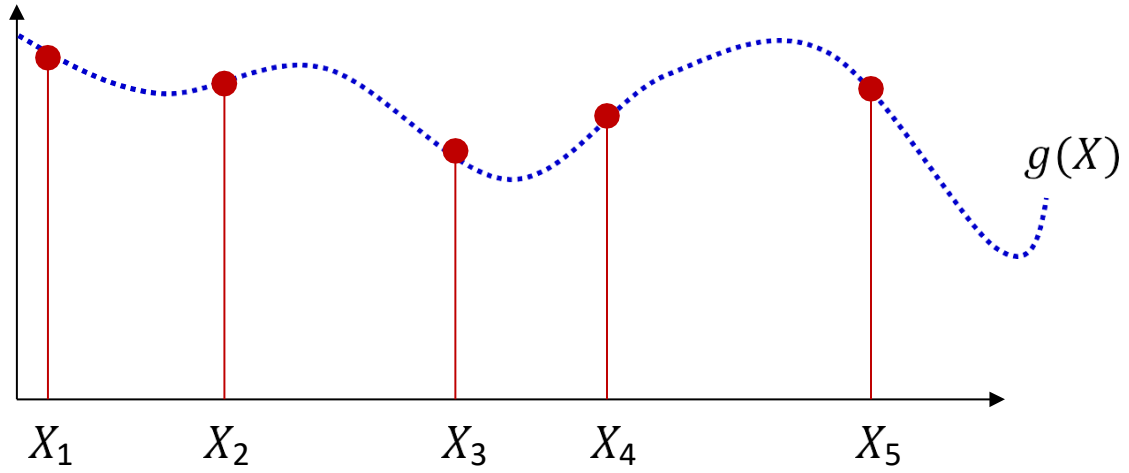
- The empirical estimate of the error is the *average* error over the training samples

$$EmpiricalError(\boldsymbol{W}) = \frac{1}{N}\sum_{i=1}^{N} error(f(X_i;\boldsymbol{W}), d_i)$$

- Estimate network parameters to minimize this average error instead

$$\widehat{\boldsymbol{W}} = \underset{W}{\operatorname{argmin}}\, EmpiricalError(\boldsymbol{W})$$

Learning the function from training samples



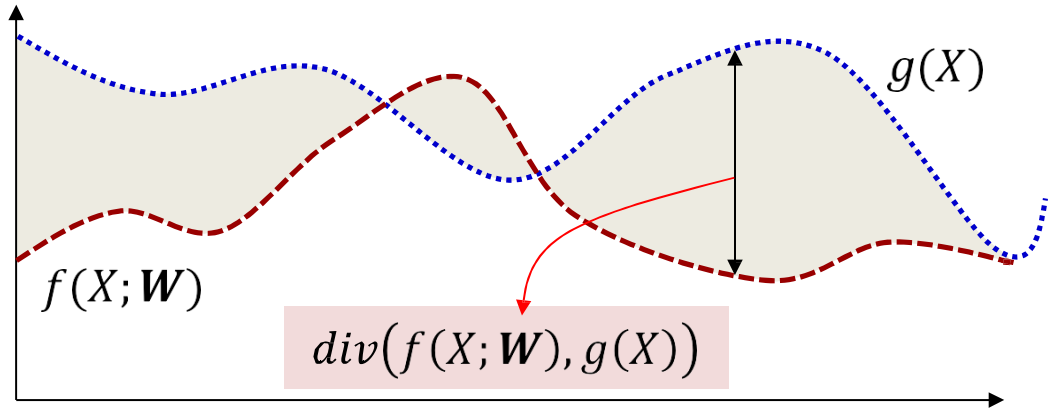
- Aim: Find the network parameters that "fit" the training points exactly

$$\boldsymbol{W}\text{: } EmpiricalError(\boldsymbol{W}) = 0$$

- Assumptions??
  - Assuming network architecture is sufficient for such a fit
  - Assuming unique output d at any $\mathbf{X}$

- And hopefully the resulting function is also correct where we *don't* have training samples

Let's make everything concrete



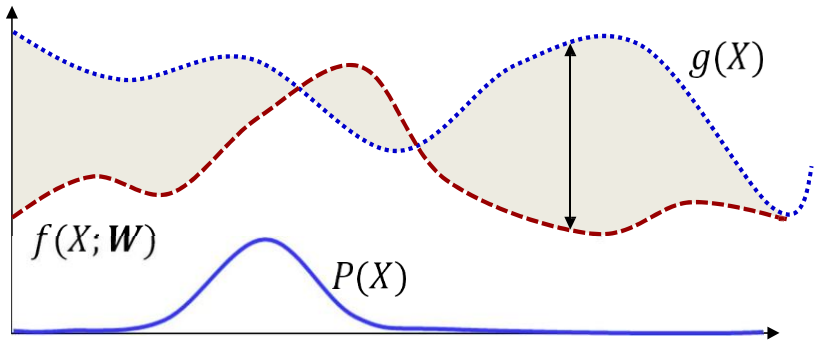
- Define the error function:
  - The divergence function quantifies mismatch between the network output and target function
  - For classification, this is usually not the classification error but a proxy to it
  - div() is a divergence function that goes to zero when

$$g(X) = f(X;W)$$

- Optimize parameters W:

$$\widehat{\boldsymbol{W}} = \underset{W}{\operatorname{argmin}} \int_X div\big(f(X;W),g(X)\big)dX$$

The Empirical risk



- More generally: assume X is a random variable following distribution P(X)

$$\widehat{W} = \underset{W}{\operatorname{argmin}} \int_X div\big(f(X;W), g(X)\big)P(X)dX$$

$$= \underset{W}{\operatorname{argmin}}\, E\big[div\big(f(X;W), g(X)\big)\big]$$

Today's Star

- The *expected* divergence (or risk) is the average divergence over the entire input space

$$E\big[div\big(f(X;W), g(X)\big)\big] = \int_X div\big(f(X;W), g(X)\big)P(X)dX$$

- The *empirical estimate* of the expected risk is the *average* divergence over the samples

$$E\big[div\big(f(X;W), g(X)\big)\big] \approx \frac{1}{N}\sum_{i=1}^{N} div(f(X_i;W), d_i)$$

Note: By sampling (collecting training data), we already take data distribution into account implicitly

Training the network: Empirical Risk Minimization (ERM)

- Given a training set of input-output pairs $(\boldsymbol{X}_1, \boldsymbol{d}_1), (\boldsymbol{X}_2, \boldsymbol{d}_2), \ldots, (\boldsymbol{X}_N, \boldsymbol{d}_N)$
  - Quantification of error on the i[th] instance: $div(f(X_i; W), d_i)$
  - Empirical average divergence (Empirical Risk) on all training data:

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

- Estimate the parameters to minimize the empirical estimate of expected divergence (empiricial risk)

$$\widehat{\boldsymbol{W}} = \underset{W}{\operatorname{argmin}}\, Loss(W)$$

  - I.e. minimize the *empirical risk* over the drawn samples

Note: Loss is only an approximation of the true risk!

Problem Statement

- Given a training set of input-output pairs $(\boldsymbol{X}_1, \boldsymbol{d}_1), (\boldsymbol{X}_2, \boldsymbol{d}_2), \ldots, (\boldsymbol{X}_N, \boldsymbol{d}_N)$
- Minimize the following function

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

w.r.t W

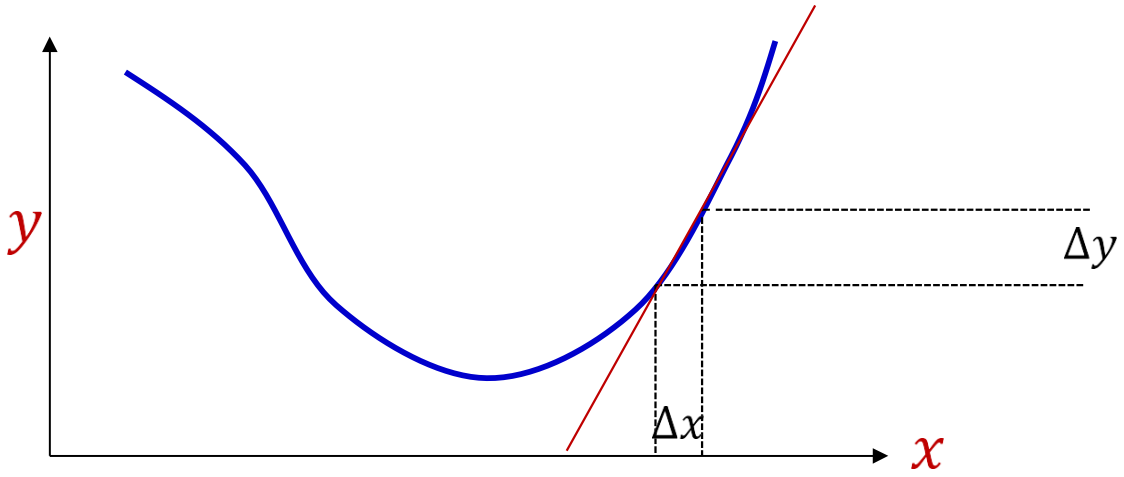
- This is problem of function minimization
  - An instance of optimization

Problem Setup:
Things to define
1: What is f() and what are its parameters W?
2: What are these input-output pairs?
3: What is the divergence div()?

A quick intro to function optimization

with an initial discussion of derivatives

A brief note on derivatives..



- It will be positive where a small increase in x results in an *increase* of f(x)
  - positive slope
- It will be negative where a small increase in x results in a *decrease* of f(x)
  - negative slope
- It will be 0 where the function is locally flat (neither increasing nor decreasing)

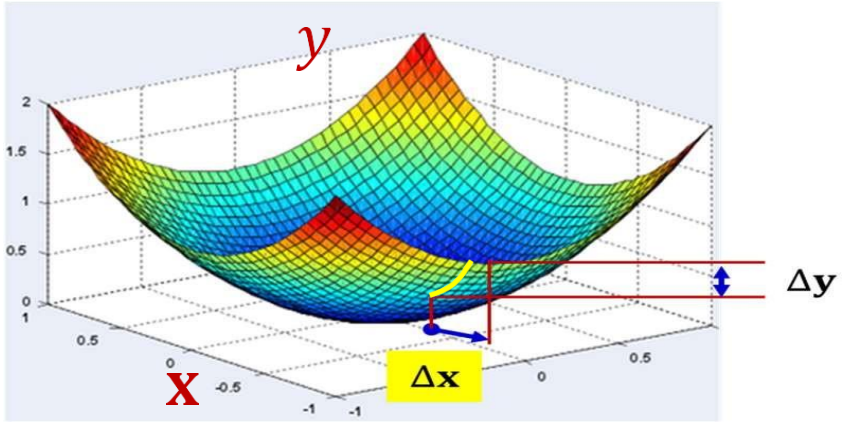
- For any $y = f(x)$, expressed as a multiplier α to a tiny increment Δx to obtain the increments Δy to the output

$$\Delta y = \alpha \Delta x$$

derivative

- When x and y are scalar
  - Often represented as $\frac{dy}{dx}$
  - Or alternately (and more reasonably) as $f'(x)$

Multivariate scalar function:
Scalar function of *vector* argument



$$\Delta y = \alpha \Delta \mathbf{x}$$

- Giving us that α is a row vector: $\alpha = [\alpha_1 \quad \cdots \quad \alpha_D]$

$$\Delta y = \alpha_1 \Delta x_1 + \alpha_2 \Delta x_2 + \cdots + \alpha_D \Delta x_D$$

- The *partial* derivative α_i gives us how y increments when *only* x_i is incremented
- Often represented as $\frac{\partial y}{\partial x_i}$

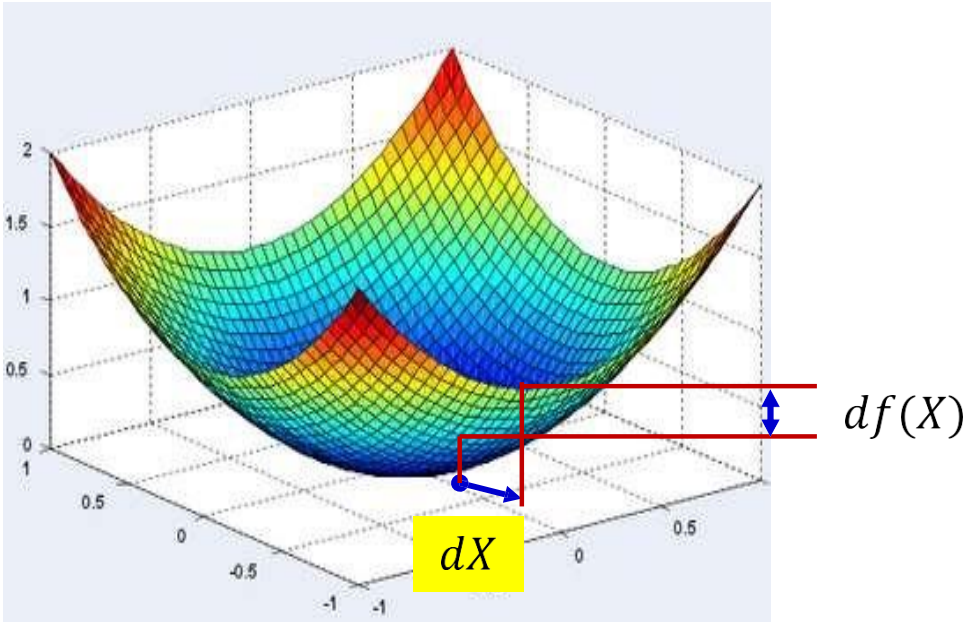
$$\Delta y = \frac{\partial y}{\partial x_1}\Delta x_1 + \frac{\partial y}{\partial x_2}\Delta x_2 + \cdots + \frac{\partial y}{\partial x_D}\Delta x_D$$

$$\Delta y = \nabla_{\mathbf{x}} y \Delta \mathbf{x}$$

where

$$\nabla_{\mathbf{x}} y = \left[\frac{\partial y}{\partial x_1} \quad \cdots \quad \frac{\partial y}{\partial x_D}\right]$$

Gradient of a scalar function of a vector



- The *derivative* $\nabla_X f(X)$ of a scalar function $f(X)$ of a multi-variate input X is a multiplicative factor that gives us the change in $f(X)$ for tiny variations in X

$$df(X) = \nabla_X f(X) dX$$

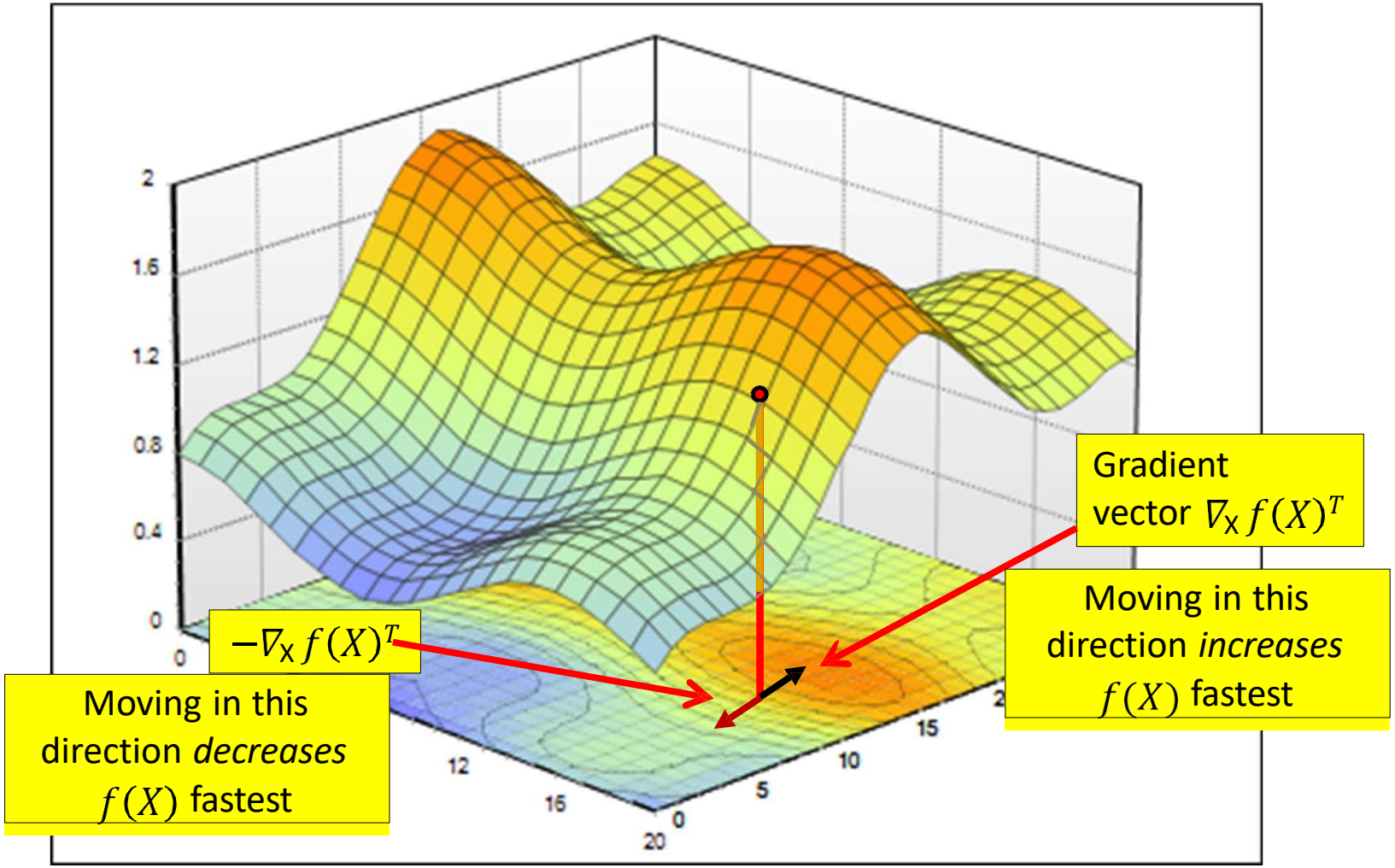
This is a vector inner product

$$\nabla_X f(x) = [\frac{\partial f(x)}{\partial x_1} \quad \frac{\partial f(x)}{\partial x_2} \quad \frac{\partial f(x)}{\partial x_3} \quad \ldots \frac{\partial f(x)}{\partial x_N}]$$

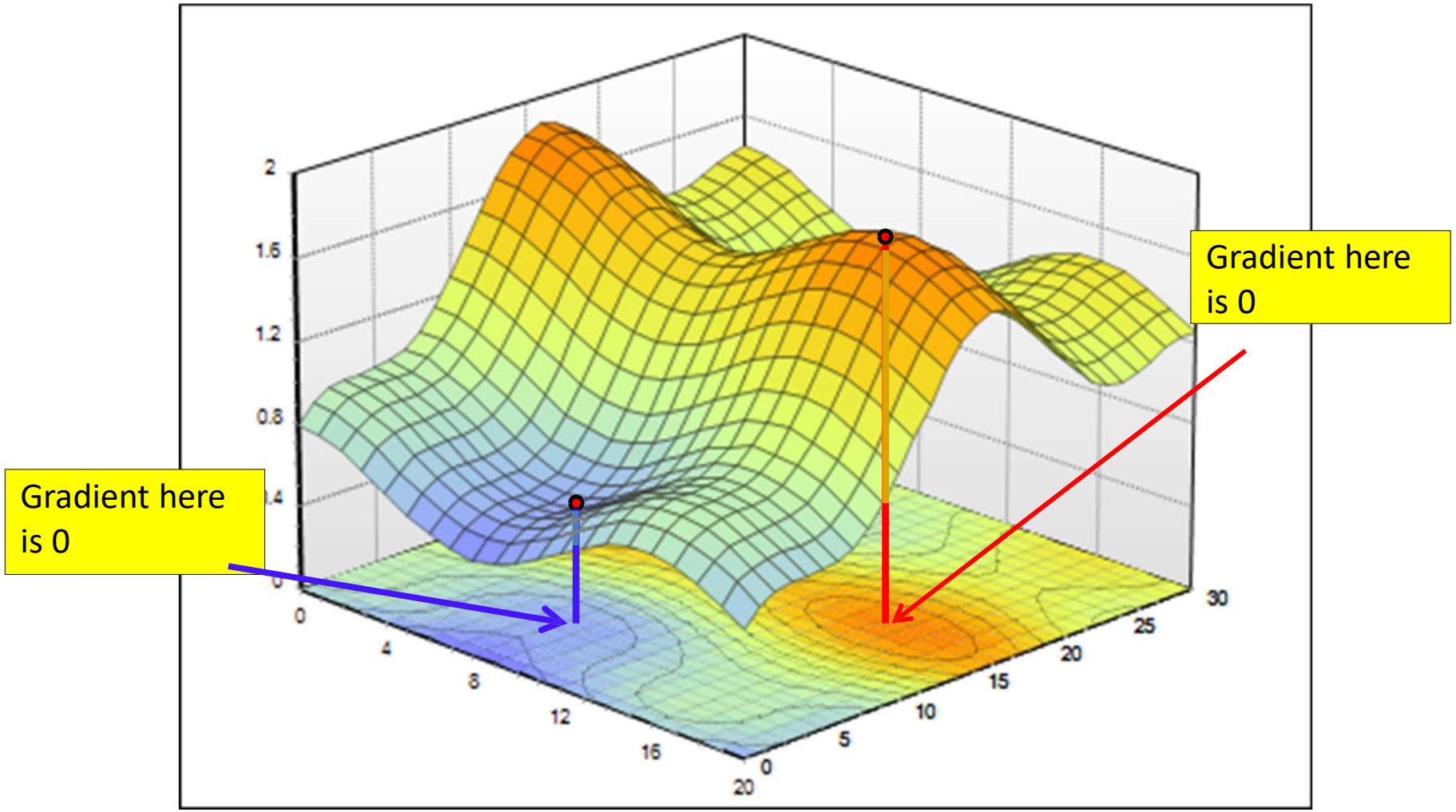
- The ***gradient*** is the transpose of the derivative $\nabla_X f(X)^T$
  - A column vector of the same dimensionality as X

The gradient is the direction of fastest increase in $f(X)$

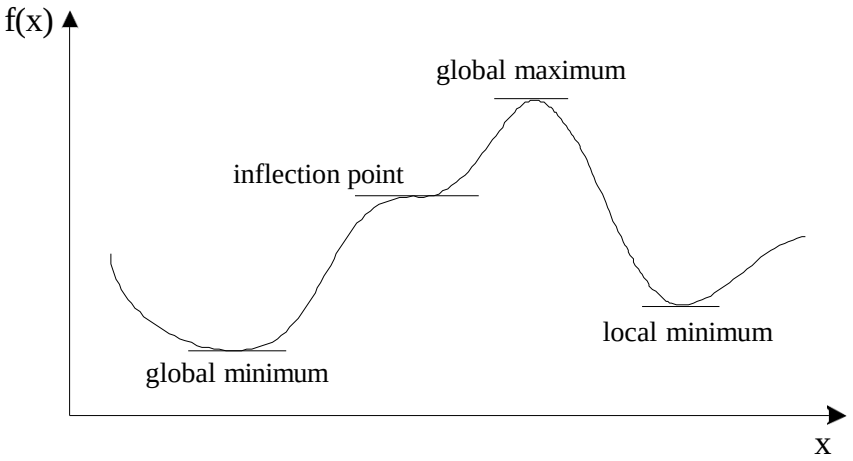
Gradient

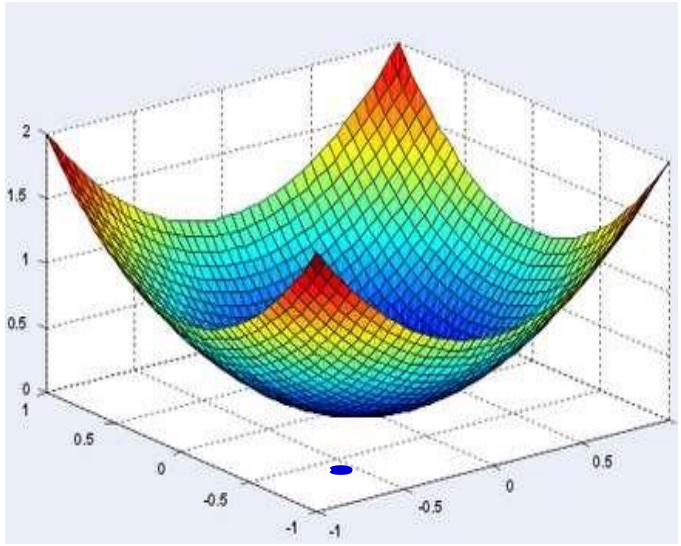

2
1.6
1.2
0.8
0.4
0
0
12
16
20
0
5
10
15
Gradient vector $\nabla_X f(X)^T$
Moving in this direction *increases* $f(X)$ fastest
$-\nabla_X f(X)^T$
Moving in this direction *decreases* $f(X)$ fastest

Gradient

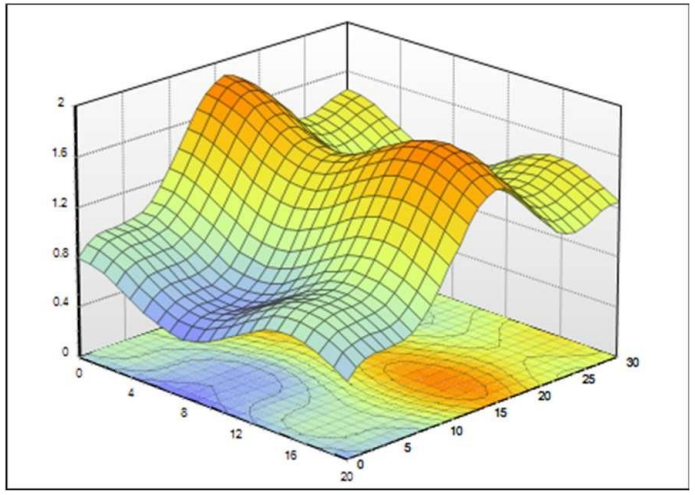

Gradient here is 0
Gradient here is 0
2
1.6
1.2
0.8
0
4
8
12
16
20
0
5
10
15
20
25
30

The problem of optimization

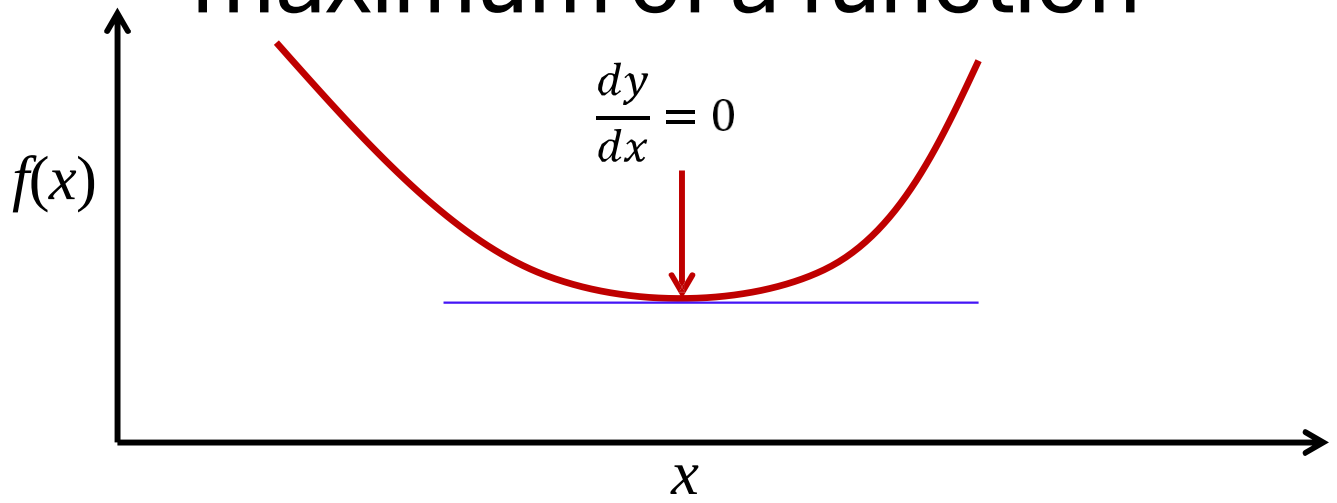




- General problem of optimization: Given a function $\mathbf{f}(\boldsymbol{x})$ of some variable $\boldsymbol{x}$ …
- Find the value of $\boldsymbol{x}$ where $\mathbf{f}(\boldsymbol{x})$ is minimum



Solution: Finding the minimum or maximum of a function



- Find the value x at which $f'(x) = 0$: Solve

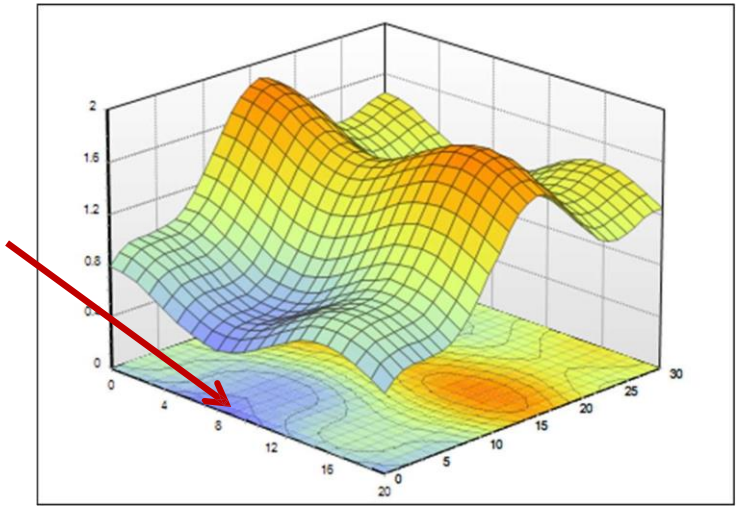
$$\frac{df(x)}{dx} = 0$$

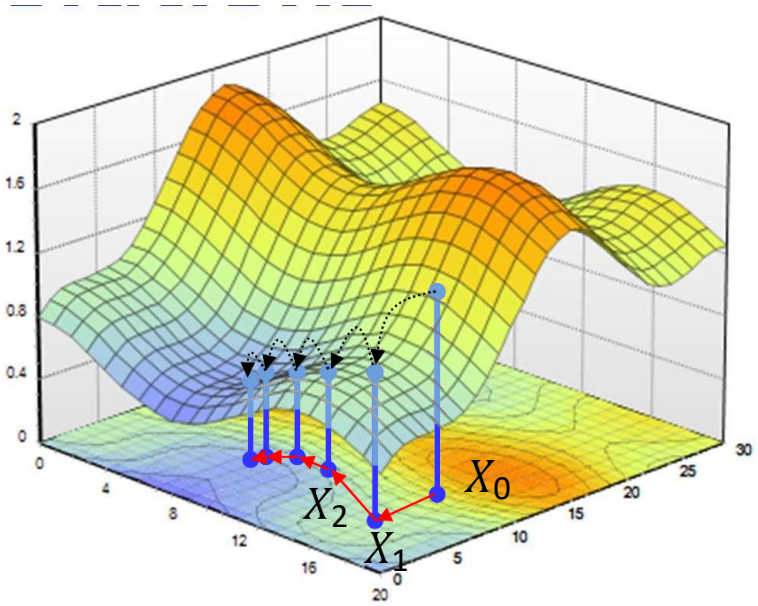
- The solution x_{soln} is a ***turning point***
- Check the double derivative at x_{soln} : compute

$$f''(x_{soln}) = \frac{df'(x_{soln})}{dx}$$

- If $f''(x_{soln}) > 0$ x_{soln} is a minimum
- If $f''(x_{soln}) < 0$ x_{soln} is a maximum

What about functions of multiple variables?





- The optimum point is still "turning" point
  - Shifting in any direction will increase the value (minimum point)
  - For smooth functions, at the minimum/maximum, the gradient is 0

- If it is not possible to simply solve $\nabla_X f(x)$
  - The function to minimize/maximize may have an intractable form
- In these situations, iterative solutions are used

Gradient descent/ascent

- 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

- Returning to our problem from our detour..

Problem Statement

- Given a training set of input-output pairs $(\boldsymbol{X}_1, \boldsymbol{d}_1), (\boldsymbol{X}_2, \boldsymbol{d}_2), \ldots, (\boldsymbol{X}_N, \boldsymbol{d}_N)$
- Minimize the following function

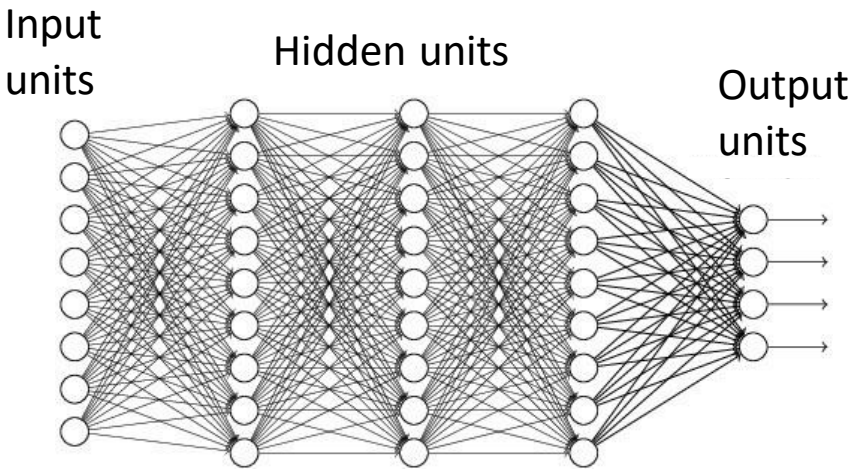
$$Loss(W) = \frac{1}{N}\sum_i div(f(X_i; W), d_i)$$

w.r.t W

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

Problem Setup:
Things to define
1: What is f() and what are its parameters W?
2: What are these input-output pairs?
3: What is the divergence div()?

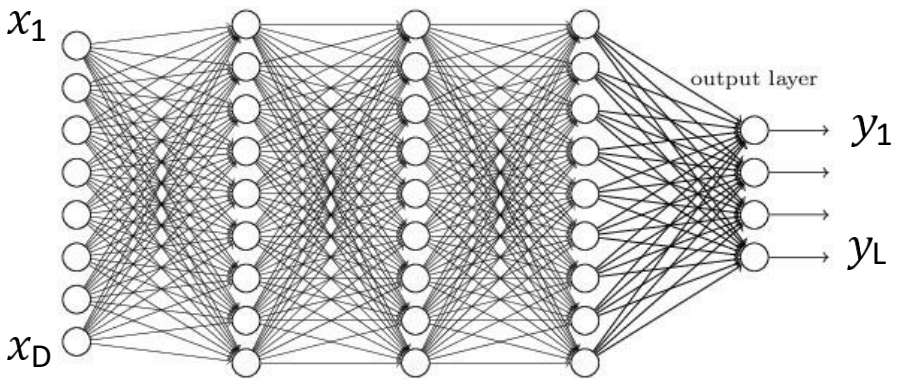
What is f()? Typical network



- Multi-layer perceptron
- A *directed* network with a set of inputs and outputs
  - No loops

- We assume a "layered" network for simplicity
  - Each "layer" of neurons only gets inputs from the earlier layer(s) and outputs signals only to later layer(s)
  - We will refer to the inputs as the ***input layer***
    - No neurons here – the "layer" simply refers to inputs
  - We refer to the outputs as the ***output layer***
  - Intermediate layers are ***"hidden" layers***

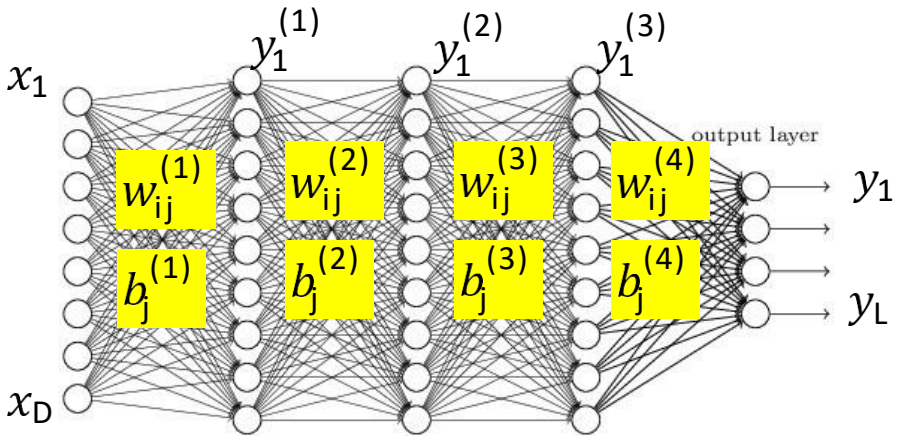
Notation



- Given a training set of input-output pairs $(X_1, d_1), (X_2, d_2), \ldots, (X_T, d_T)$
- $X_n = [x_{n1}, x_{n2}, \ldots, x_{nD}]^T$ is the *n*th input vector
- $d_n = [d_{n1}, d_{n2}, \ldots, d_{nL}]^T$ is the *n*th Ground Truth annotation
- $Y_n = [y_{n1}, y_{n2}, \ldots, y_{nL}]^T$ is the *n*th vector of *actual* outputs of the network
  - Function of input X_n and network parameters

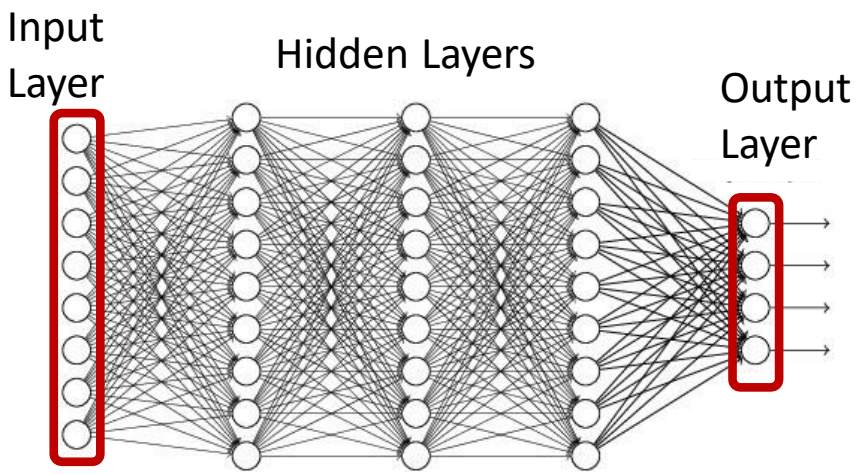
When referring to a *specific* instance, we will drop the first subscript

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 $b_j^{(k)}$

Input and Output

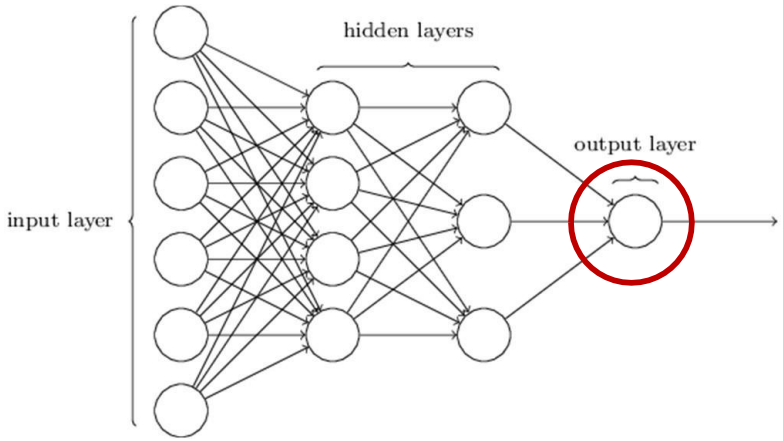


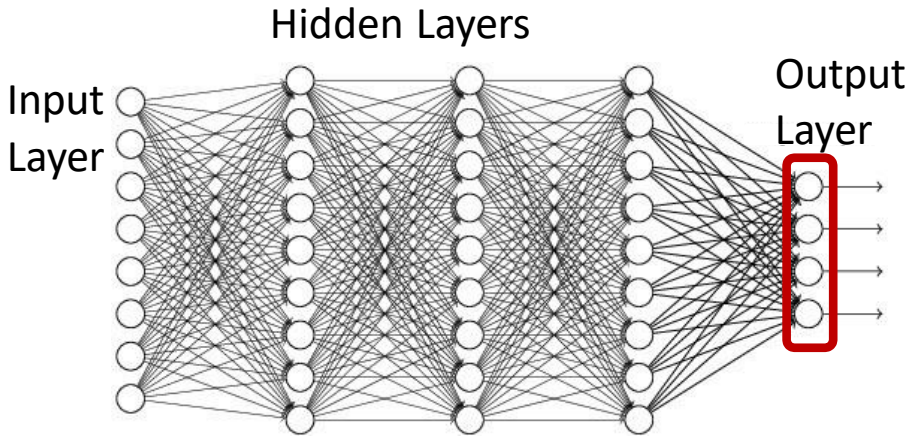
- Inputs are vectors of numbers
  - (or may even be just a scalar, if input layer is of size 1)
  - E.g. vector of pixel values
  - E.g. vector of speech features
  - E.g. real-valued vector representing text

- The output can be
  - Real-valued
  - Binary
  - Categorical

Output: Real-valued

- If the desired *output* is real-valued, no special tricks are necessary





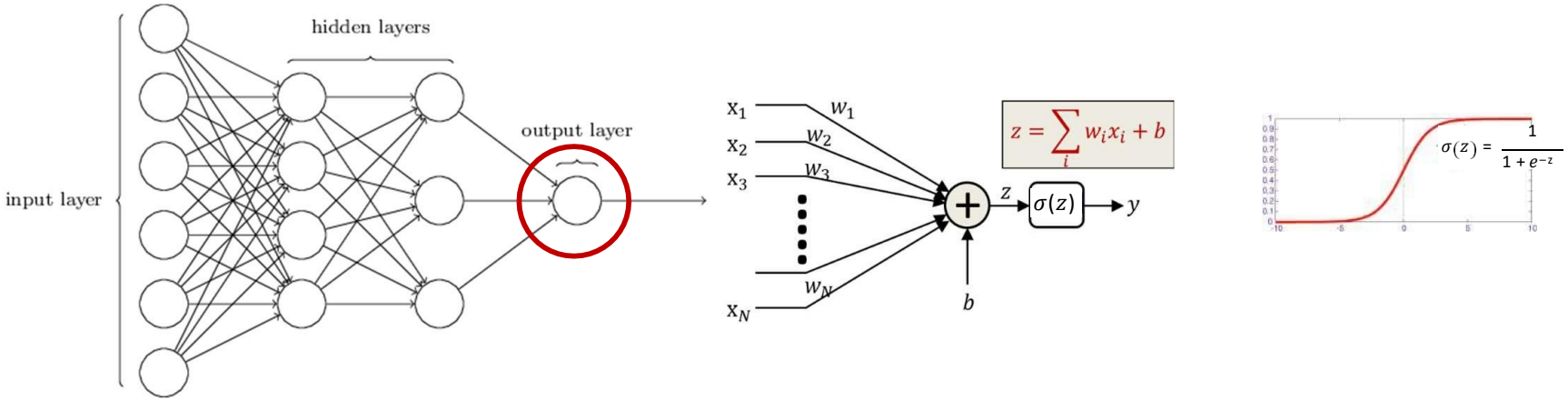
Scalar Output : single output neuron

- d = scalar (real value)

Vector Output : as many output neurons as the dimension of the desired output

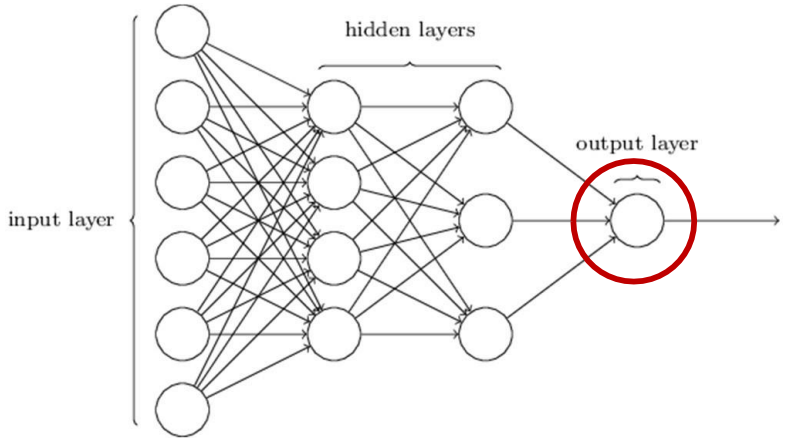
- d = [d_1 d_2 .. d_L] (vector of real values)

Output: Binary



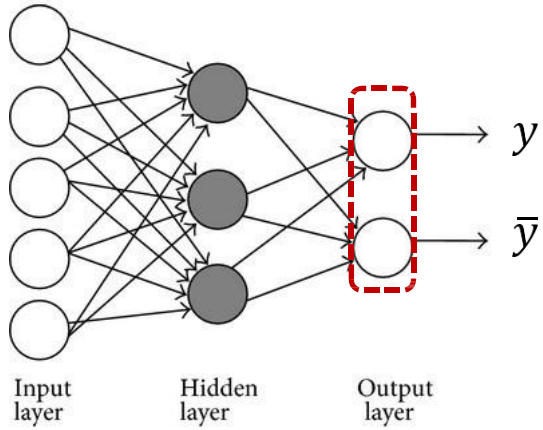
- If the desired output is binary use a simple 1/0 representation of the desired output
- Require output activation: Typically, a sigmoid
  - Viewed as the *probability p(Y=1|X)* of class value 1
    - Indicating the fact that for actual data, in general a feature value X may occur for both classes, but with different probabilities
    - Is differentiable

Output: Binary



- 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.

Alternatively:



- Sometimes represented by *two* outputs, one representing the desired output, the other representing the *negation* of the desired output
  - Yes: [1 0]
  - No: [0 1]
- The output explicitly becomes a 2-output softmax

Multi-class output: One-hot representations

- Multi-class: 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, with the classes arranged in a chosen order:

  [cat dog camel hat flower]^T
- For inputs of each of the five classes the desired output is:

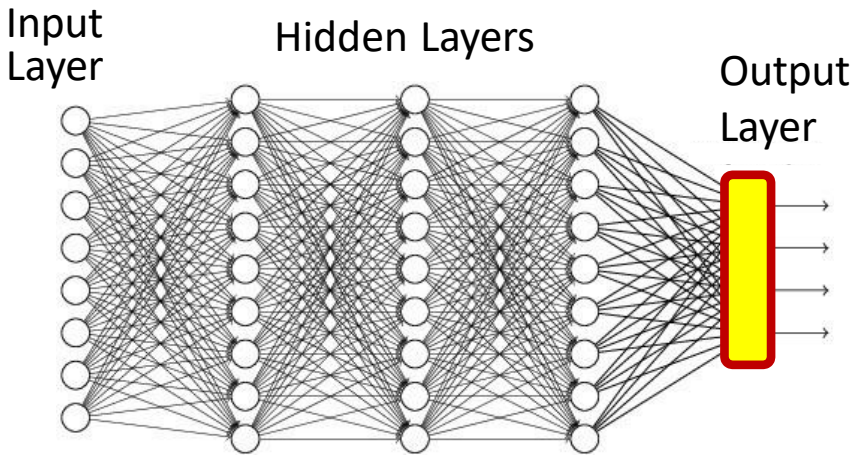
  cat: [1 0 0 0 0] ^{T}

  dog: [0 1 0 0 0] ^{T}

  camel: [0 0 1 0 0] ^{T}

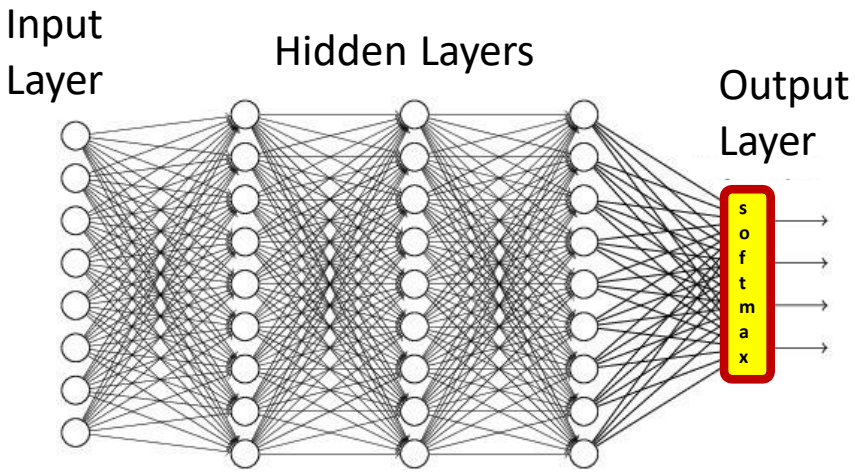
  hat: [0 0 0 1 0] ^{T}

  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***



- For a multi-class classifier with N classes, the desired output is an N-dimensional binary vector (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 output: Softmax



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

$$y_i = \frac{exp(z_i)}{\sum_j exp(z_j)} \quad \text{with} \quad z_i = \sum_j w_{ji}^{(n)} y_j^{(n-1)}$$

Softmax is a vector activation

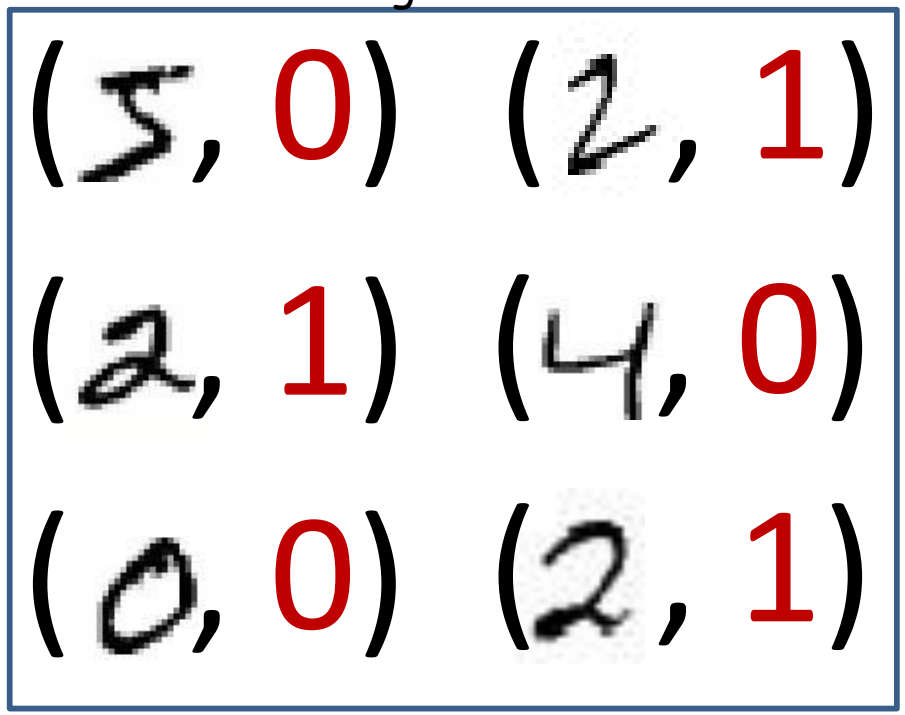
- This can be viewed as the probability $y_i = P(class = i|X)$

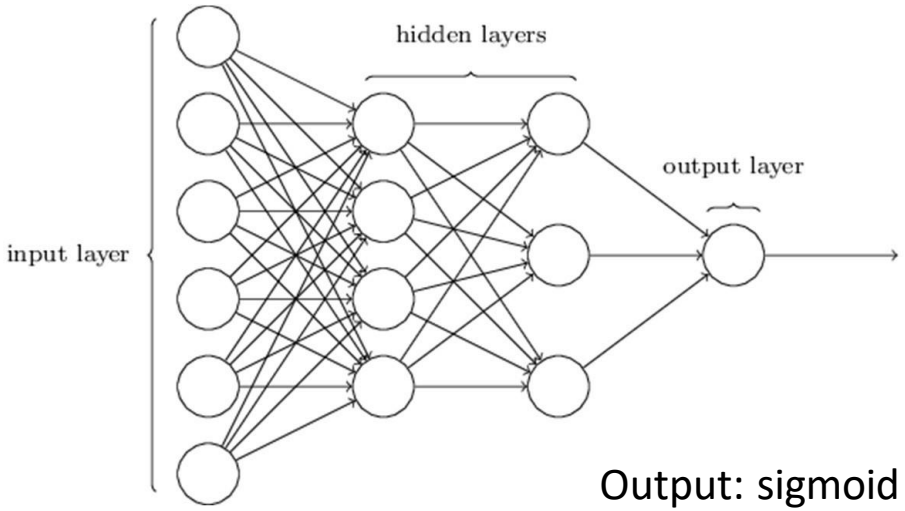
Inputs and outputs: Typical Problem Statement

- We are given a number of “training” data instances
- E.g. images of digits, along with information about which digit the image represents
- Tasks:
  - Binary recognition: Is this a “2” or not
  - Multi-class recognition: Which digit is this?

Typical Problem statement:
binary classification

Training data





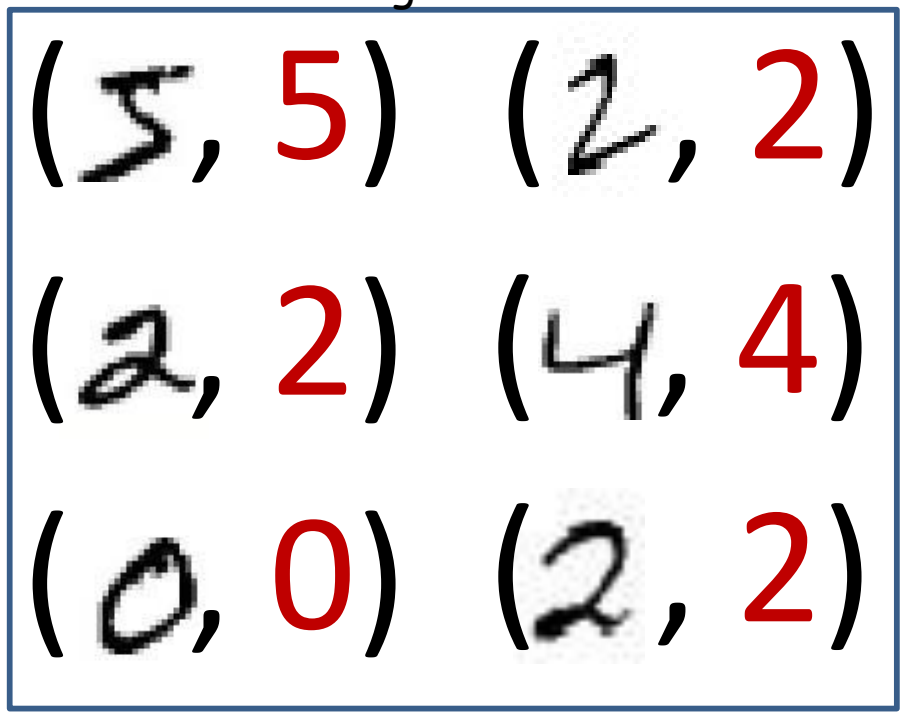
Output: sigmoid

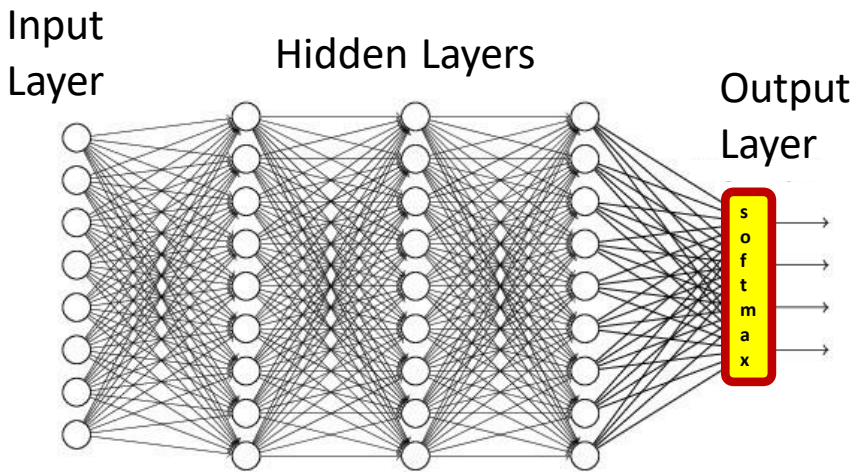
Input: vector of pixel values

- Given, many positive and negative examples (training data),
  - learn all weights such that the network does the desired job

Typical Problem statement: **multiclass classification**

Training data



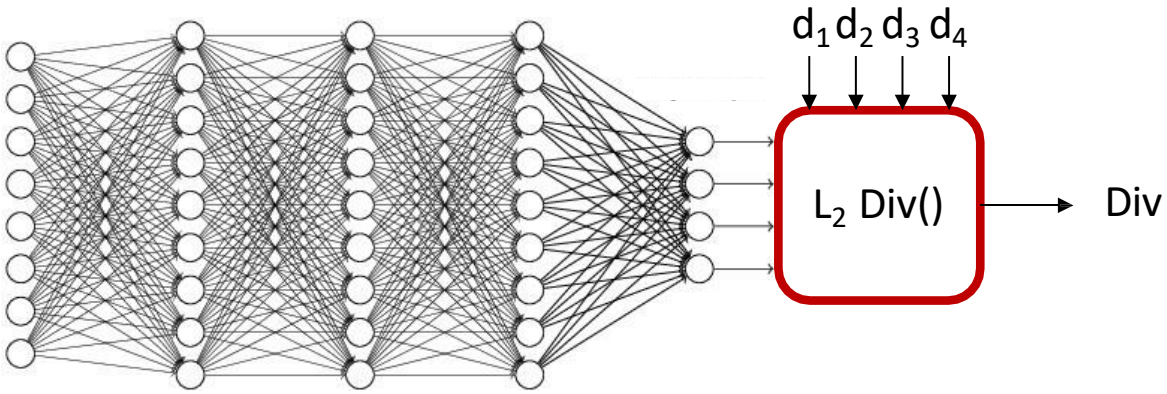


Input: vector of pixel values

Output: Class prob

- Given, many positive and negative examples (training data),
  - learn all weights such that the network does the desired job

Divergence functions: Real-valued Outputs



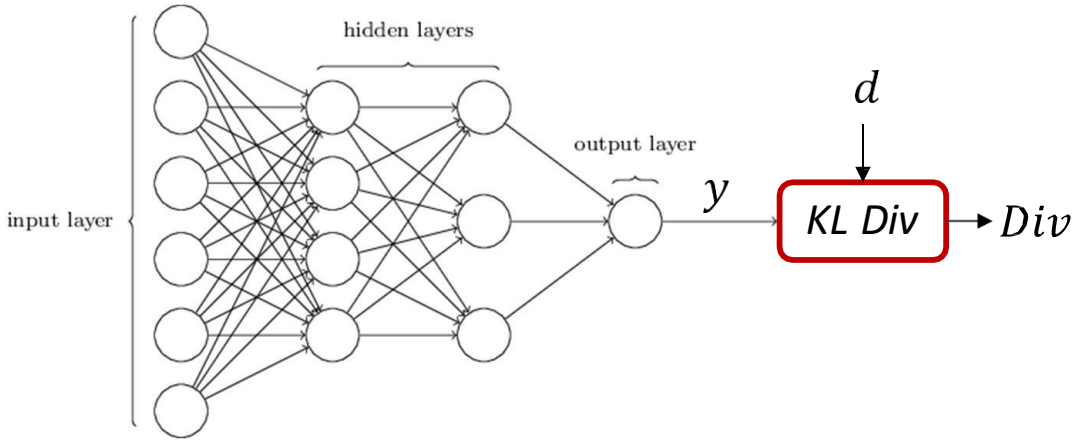
- For real-valued output vectors, the (scaled) L_2 divergence is popular

$$Div(Y,d) = \frac{1}{2}\|Y-d\|^2 = \frac{1}{2}\sum_{\mathrm{i}}(y_{\mathrm{i}}-d_{\mathrm{i}})^2$$

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

$$\frac{dDiv(Y,d)}{dy_{\mathrm{i}}} = (y_{\mathrm{i}}-d_{\mathrm{i}})$$

Divergence functions: Binary Classifier



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

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

also known as Binary Cross-Entropy, a special case of KL when H(d)=0

  - Minimum when $d = Y$

Today's Star

- Derivative

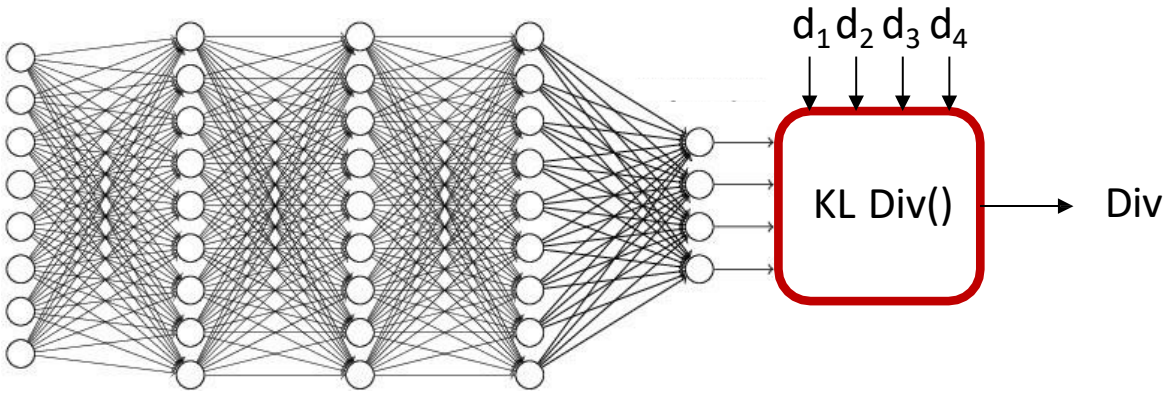
$$\frac{dDiv(Y,d)}{dY} = \begin{cases} -\frac{1}{Y} & if \ d = 1 \\ \frac{1}{1-Y} & if \ d = 0 \end{cases}$$

Note: $div() = 0$ (minimum) *when y = d*

However, when y = d the derivative is *not* 0!

This is due to the logarithmic nature of KL

Divergence functions: Multi-class classification



- Desired output d is a one hot vector [0 0 …1 …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 KL divergence between the desired one-hot output and actual output:

$$KL(Y,d) = \sum_i d_i log \frac{d_i}{y_i} = \sum_i d_i log d_i - \sum_i d_i log y_i = -log y_c$$

- Derivative

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

Note: $div() = 0$ (minimum) *when y = d*

However, when y = d the derivative is *not* 0!

The slope is negative w.r.t., y_c

means *increasing* y_c will *reduce* divergence

KL divergence vs cross entropy

- KL divergence between d and y:

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

- Cross-entropy between d and y:

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

The cross entropy is merely the $KL(Y,d) - H(d)$

- The W that minimizes cross-entropy will minimize the KL divergence
  - $H(d)$ does not depend on the neural net
  - In fact, for one-hot d, $H(d) = 0$ (and $KL(Y,d) = H(Y,d)$)
- We will generally minimize to the cross-entropy loss rather than the KL divergence

For KL divergence,

$KL(Y,d) \geq 0$

$KL(Y,d)$ achieves minimum when Y = d

Its minimum is zero

For cross-entropy,

$H(Y,d) \geq H(d)$

It achieves minimum when Y = d

But: its minimum is not zero!

Note: cross-entropy is NOT a measure of divergence.
Although it achieves minimum when Y = d, its minimum is not Zero.

Understanding the derivatives of KL

Consider binary case:

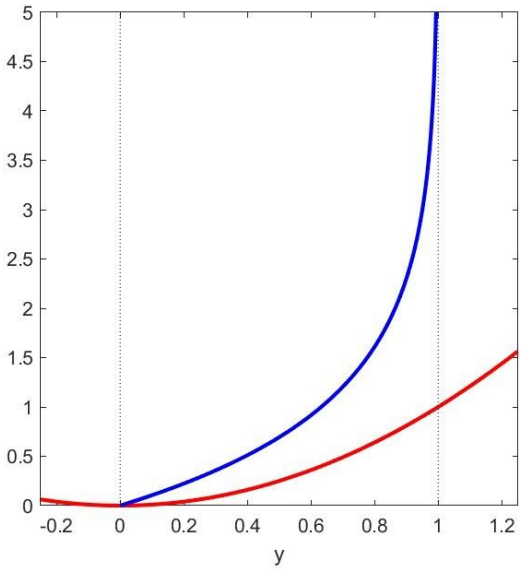
$$KL(Y,d) = -dlogY - (1-d)\log(1-Y)$$

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

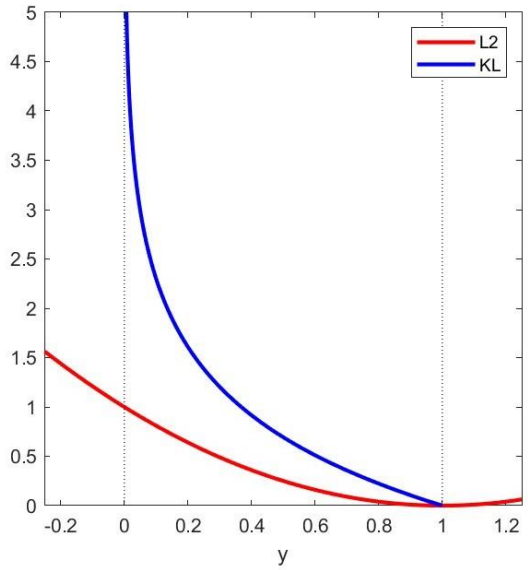
L2 for comparison:

$$L2(Y,d) = (y-d)$$

d=0



d=1



- Both KL and L2 have a minimum when y is the target value of d
- KL rises much more steeply away from d
  - Encouraging faster convergence of gradient descent
- The derivative of KL is *not* equal to 0 at the minimum
  - It is 0 for L2, though

The non-zero gradient ensures that the optimization algorithm remain sensitive to changes and continue to refine Y

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial w}$$

Problem Statement

- Given a training set of input-output pairs $(\boldsymbol{X}_1, \boldsymbol{d}_1), (\boldsymbol{X}_2, \boldsymbol{d}_2), \ldots, (\boldsymbol{X}_N, \boldsymbol{d}_N)$
- Minimize the following function

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

  w.r.t W

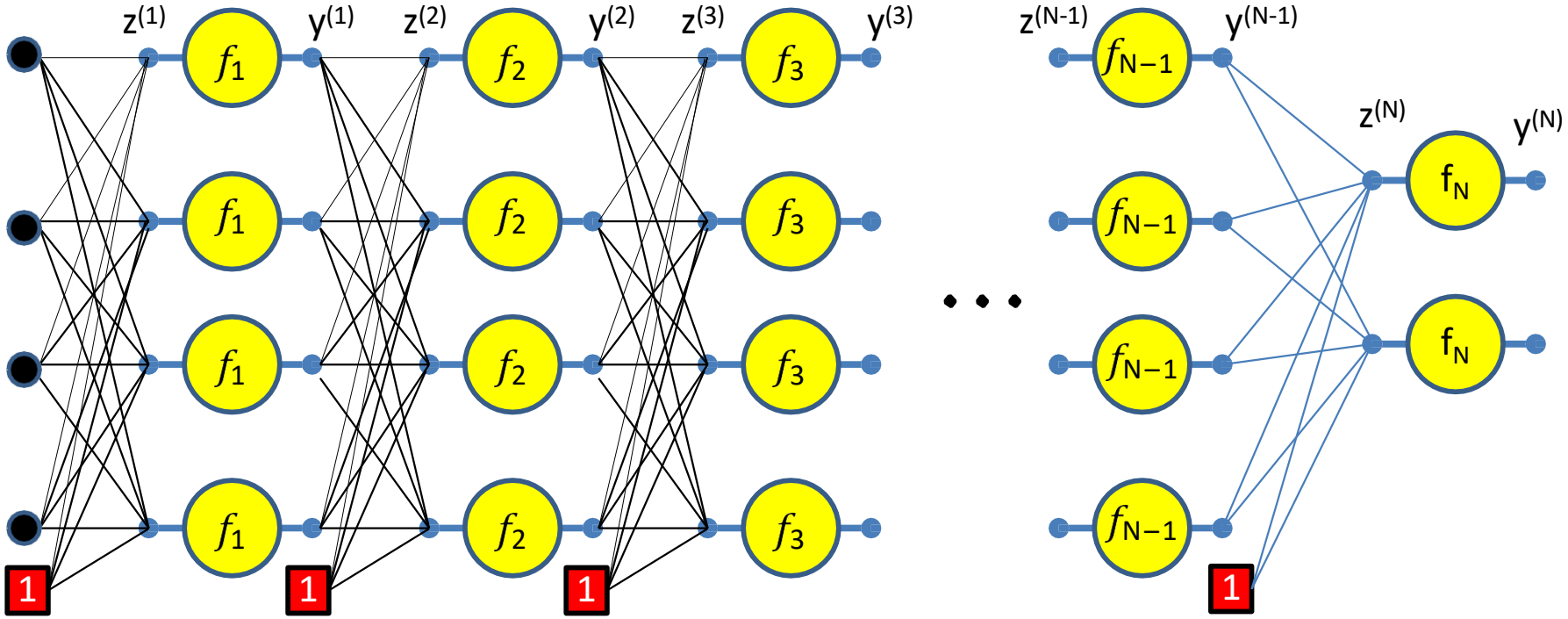
- This is problem of function minimization
  - An instance of optimization

Problem Setup:
Things to define

- ✓ What is f() and what are its parameters W?
- ✓ What are these input-output pairs?
- ✓ What is the divergence div()?

We are ready!
Next, we will start forward pass and backward update

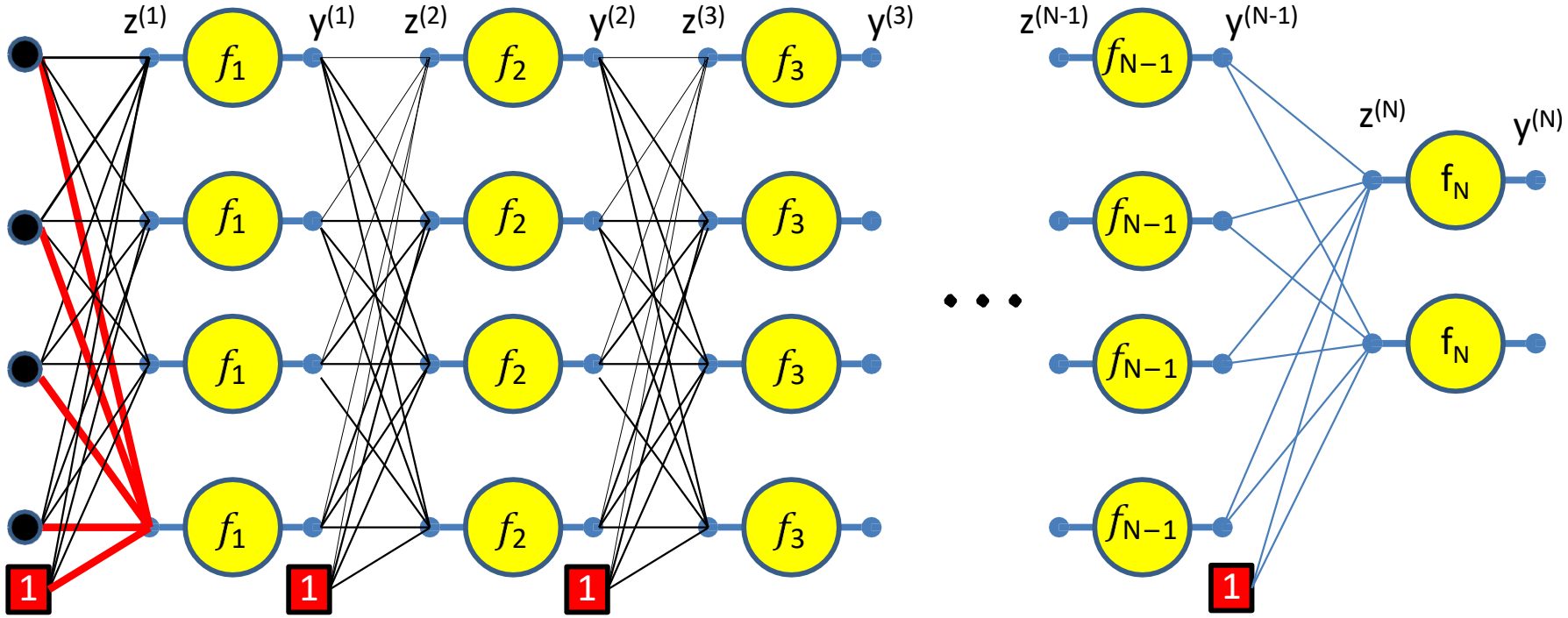
Forward pass



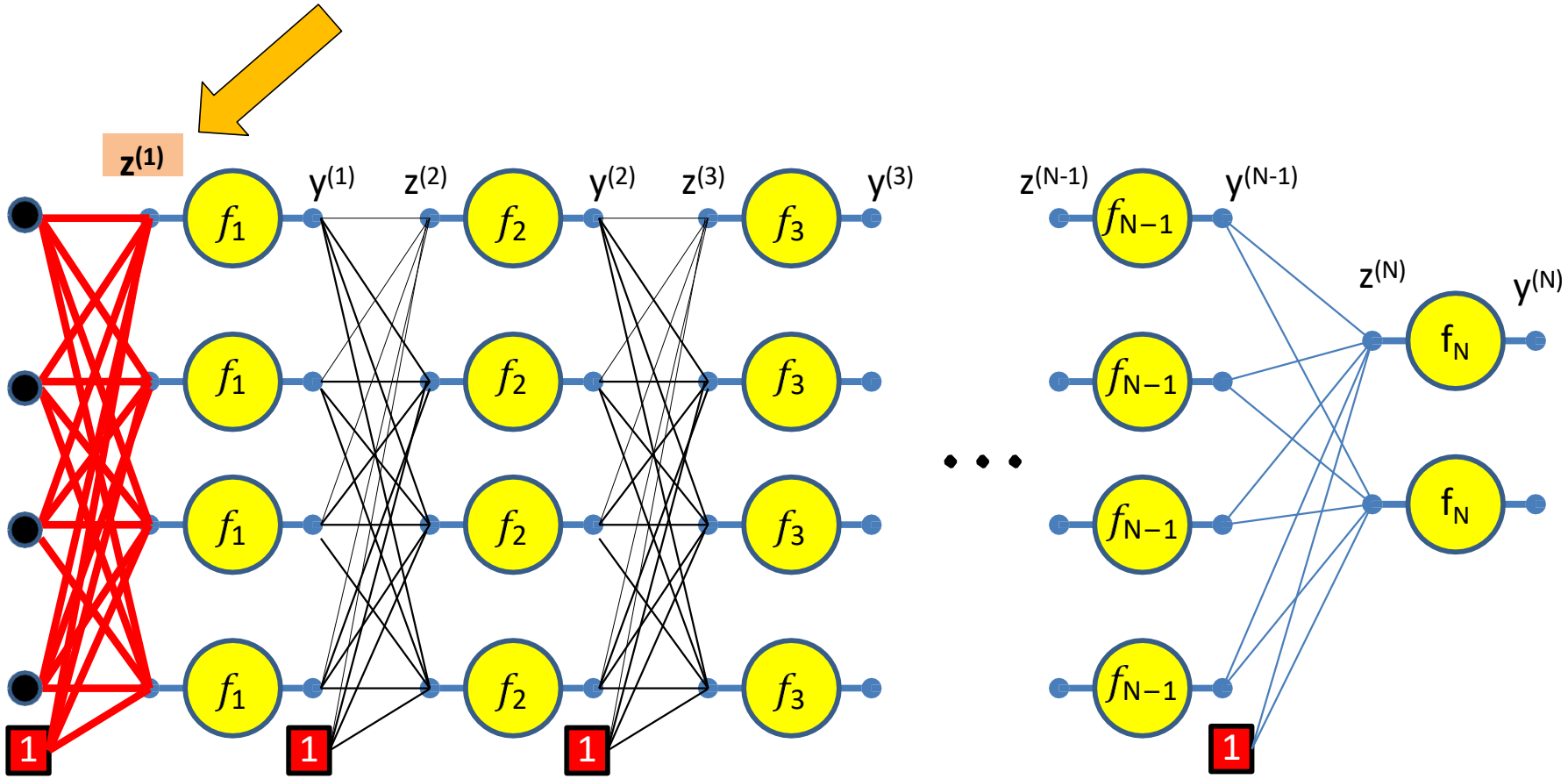
Setting $y_i^{(0)} = x_i$ for notational convenience

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

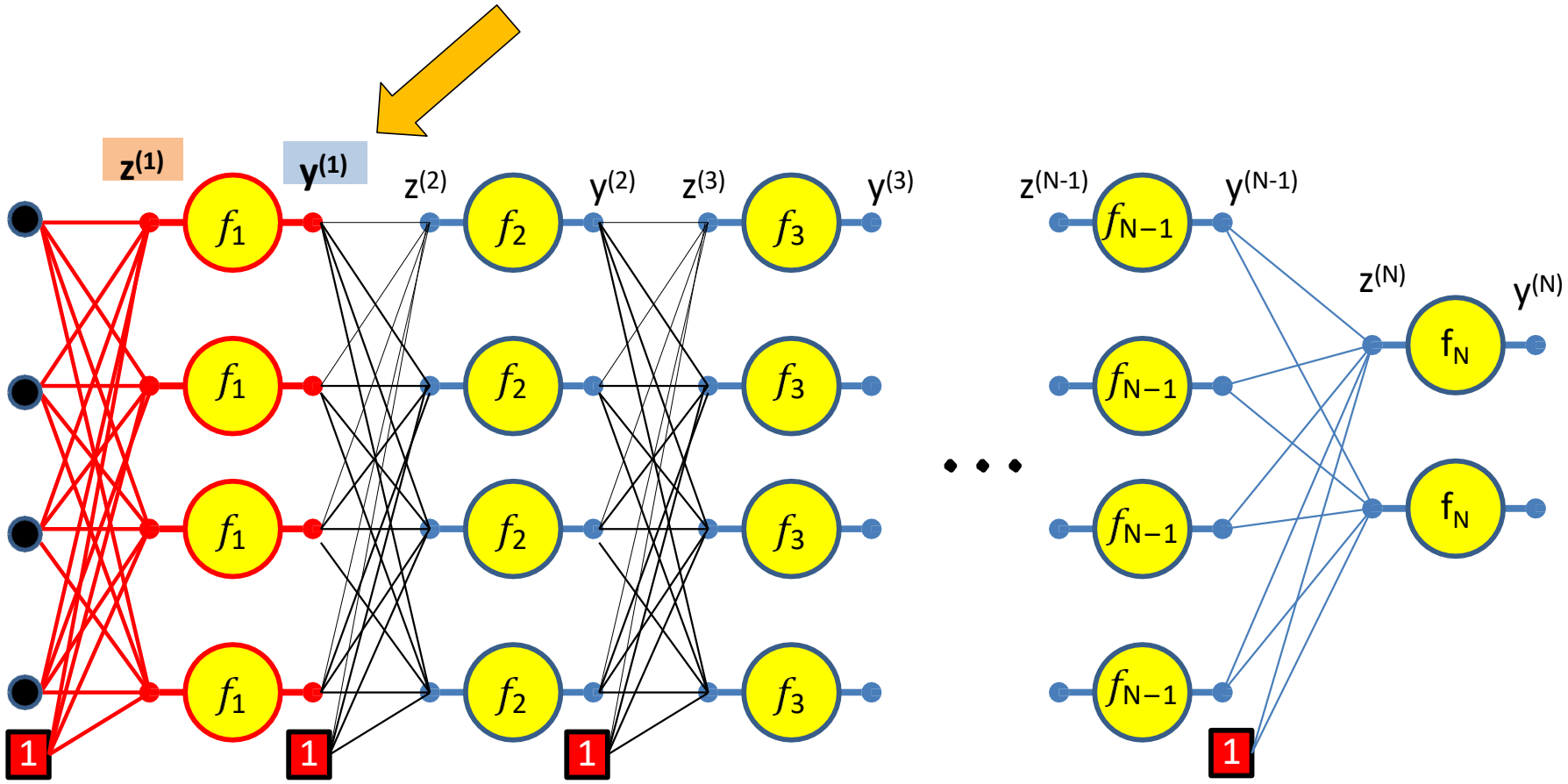
Forward pass



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

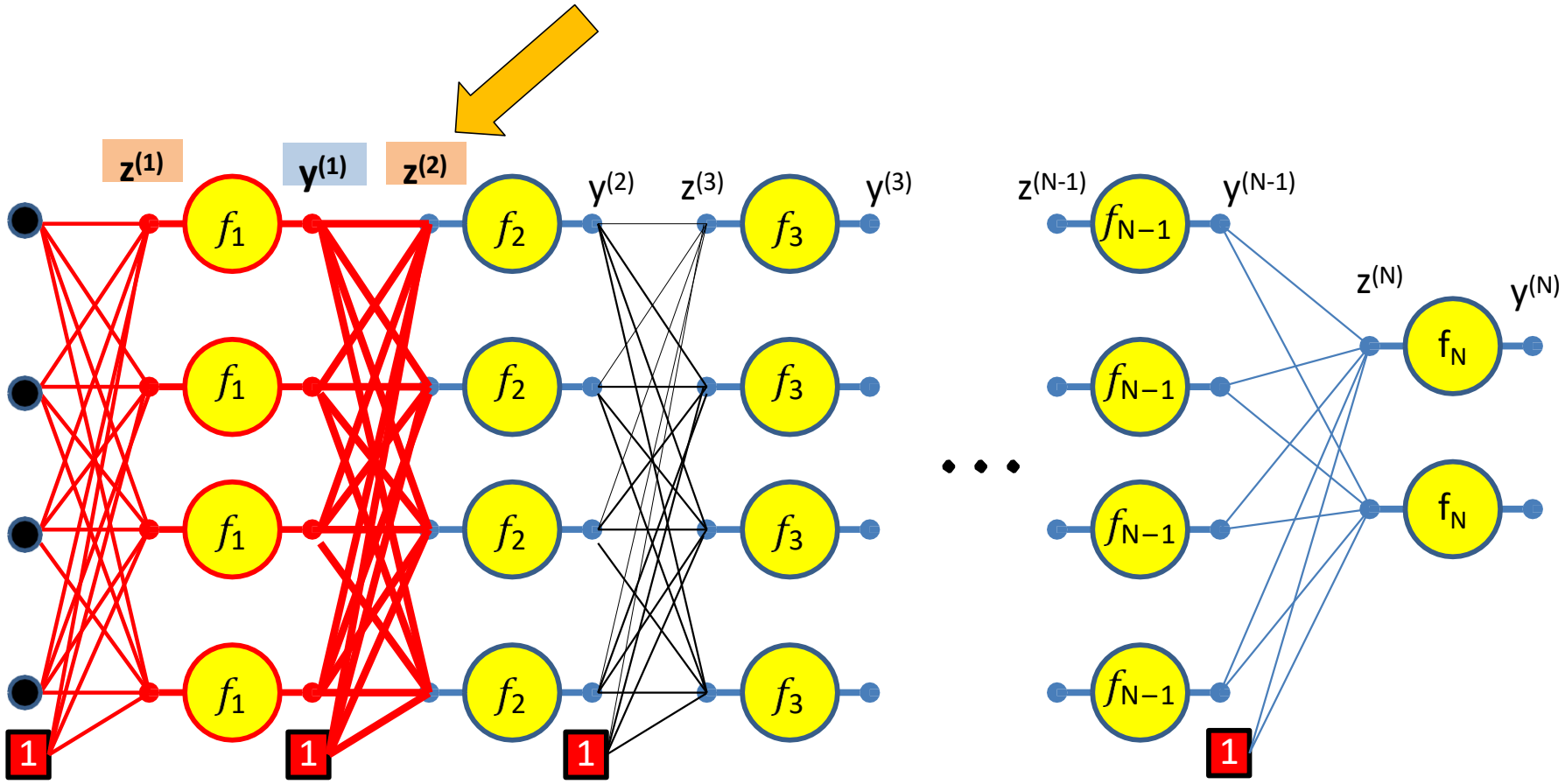


$$z_j^{(1)} = \sum_i w_{ij}^{(1)} y_i^{(0)}$$



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

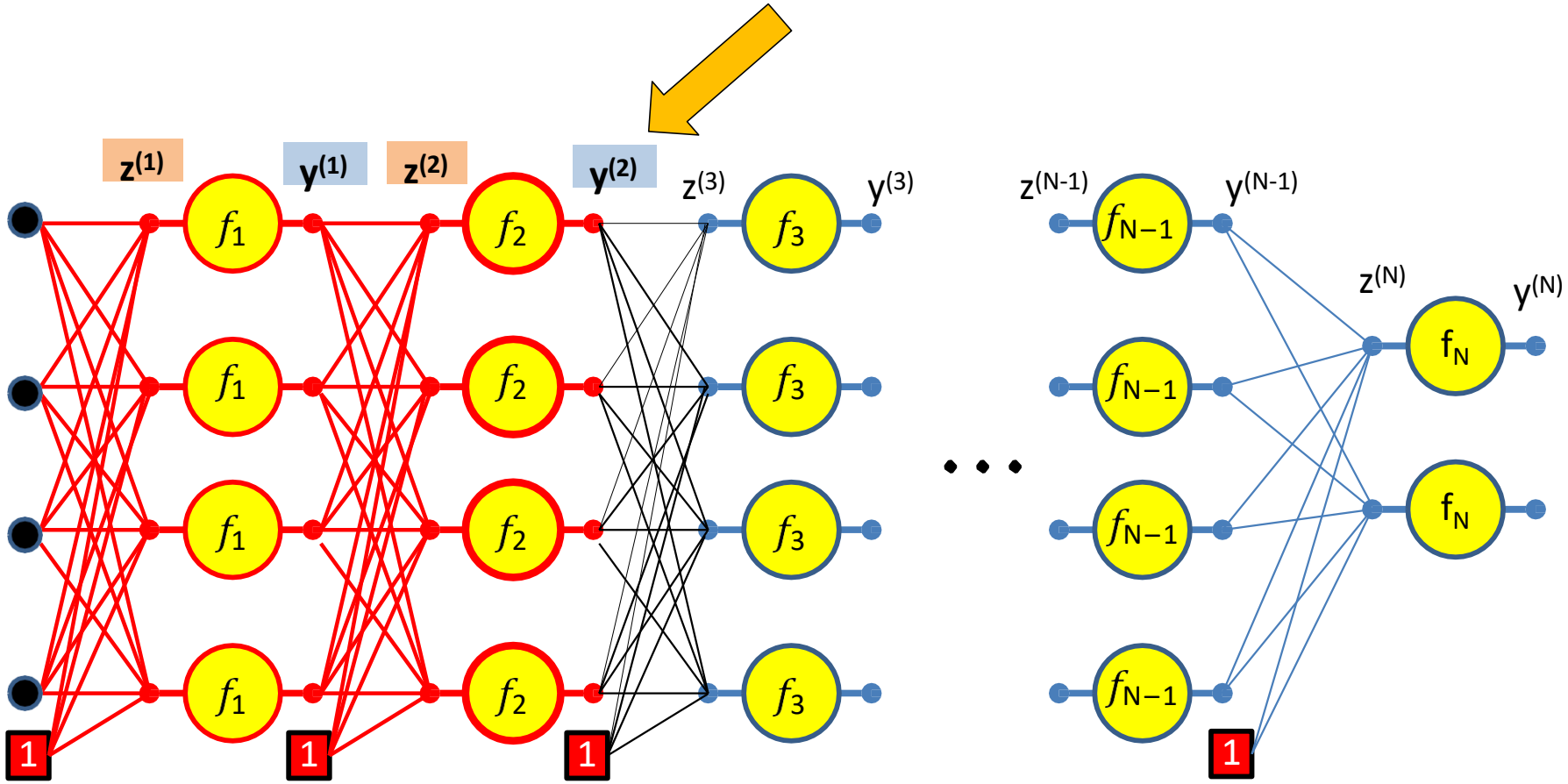
$$y_j^{(1)} = f_1\left(z_j^{(1)}\right)$$



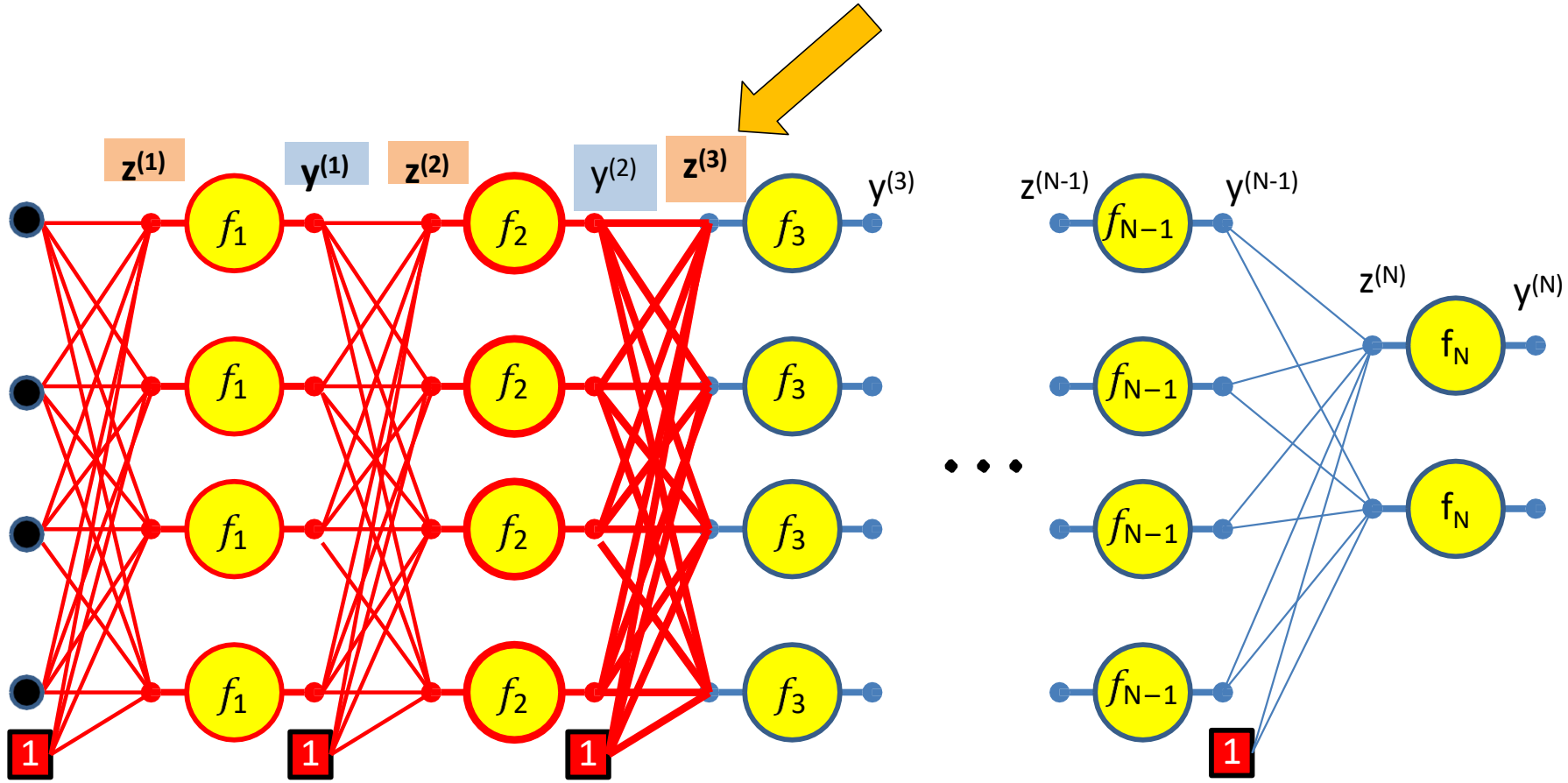
$$z_j^{(1)} = \sum_i w_{ij}^{(1)} y_i^{(0)}$$

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

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

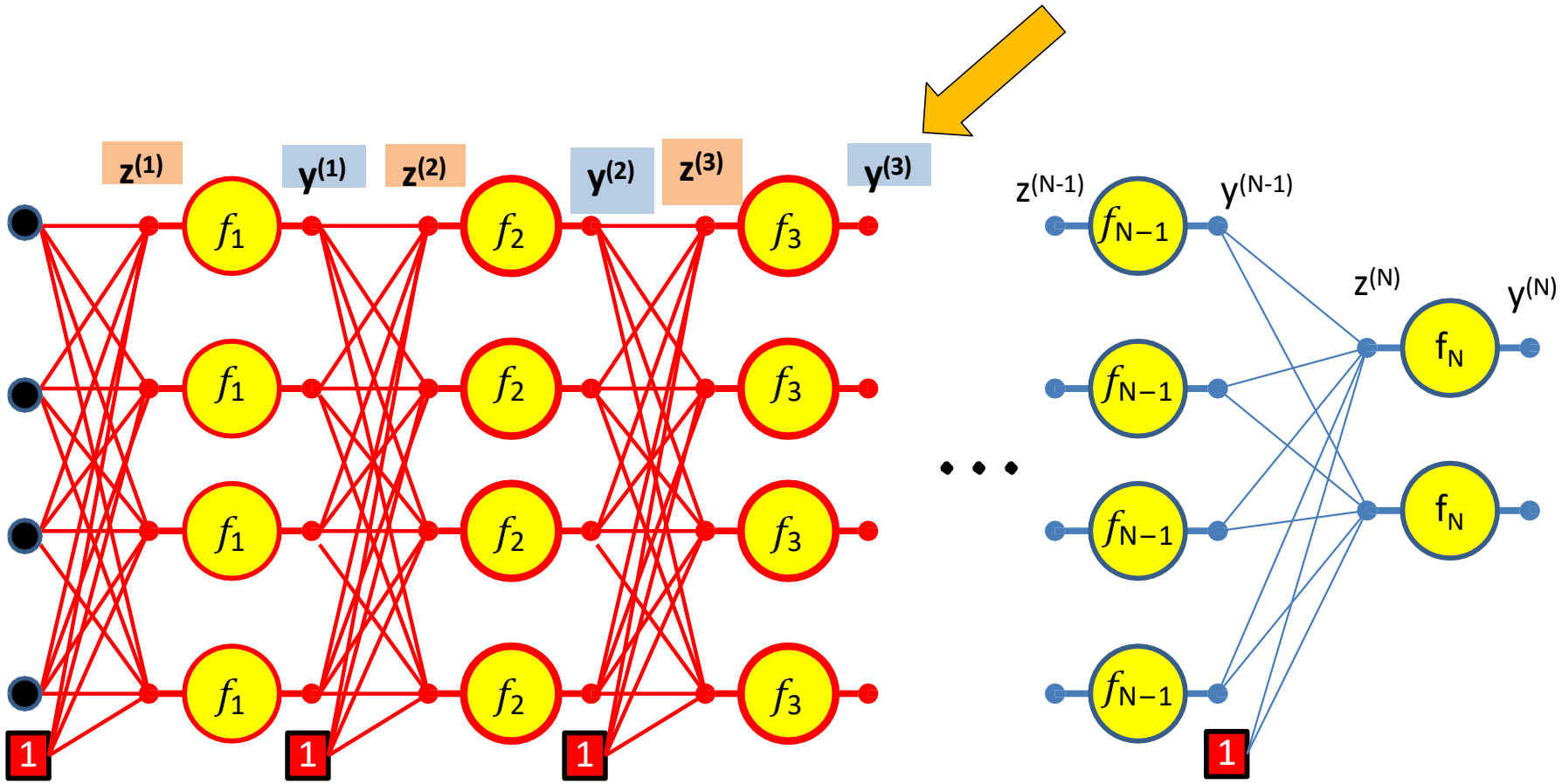


$$z_j^{(1)} = \sum_i w_{ij}^{(1)} y_i^{(0)} \qquad y_j^{(1)} = f_1\left(z_j^{(1)}\right) \qquad z_j^{(2)} = \sum_i w_{ij}^{(2)} y_i^{(1)} \qquad y_j^{(2)} = f_2\left(z_j^{(2)}\right)$$



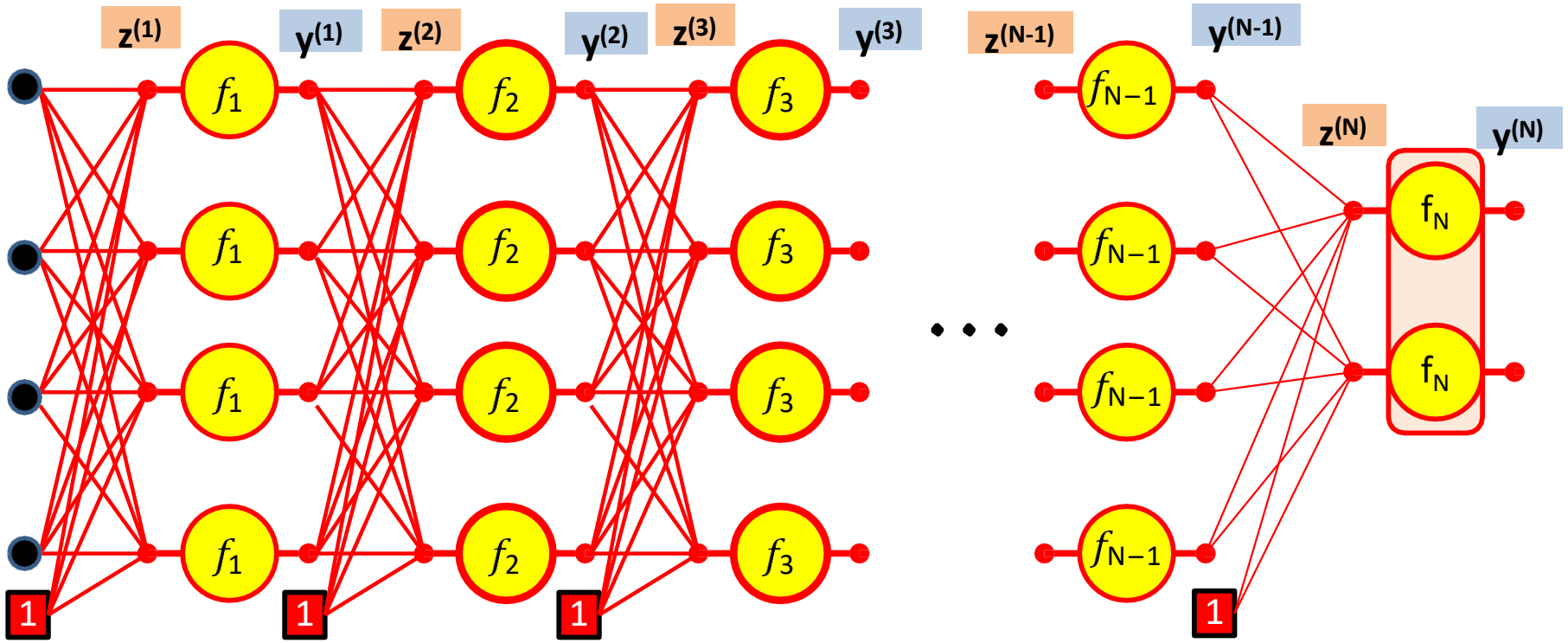
$$z_j^{(1)} = \sum_i w_{ij}^{(1)} y_i^{(0)} \qquad y_j^{(1)} = f_1\left(z_j^{(1)}\right) \qquad z_j^{(2)} = \sum_i w_{ij}^{(2)} y_i^{(1)} \qquad y_j^{(2)} = f_2\left(z_j^{(2)}\right)$$

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



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

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



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

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

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

Forward "Pass"

- Input: D dimensional vector $\mathbf{x} = [x_j,\ j = 1 \ldots D]$
- Set:
  - $D_0 = D,$ is the width of the 0th (input) layer
  - $y_j^{(0)} = x_j,\ j = 1 \ldots D;$ $\quad y_0^{(k=1\ldots N)} = x_0 = 1$
- For layer $k = 1 \ldots N$
  - For $j = 1 \ldots D_k$ $\quad$ 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:
  - $Y = y_j^{(N)}, j = 1..D_N$

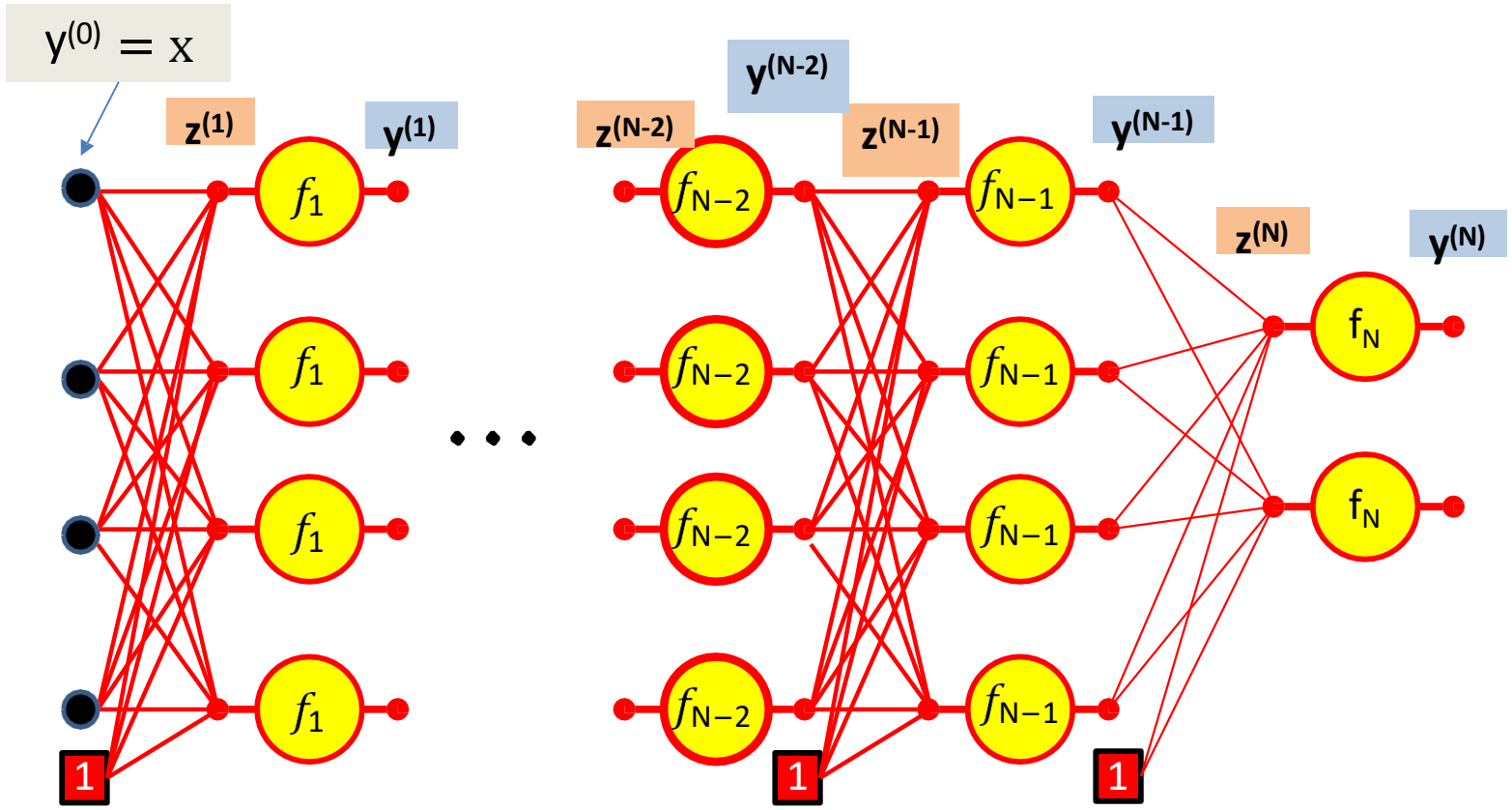
Training Neural Nets through Gradient Descent

Total training Loss:

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

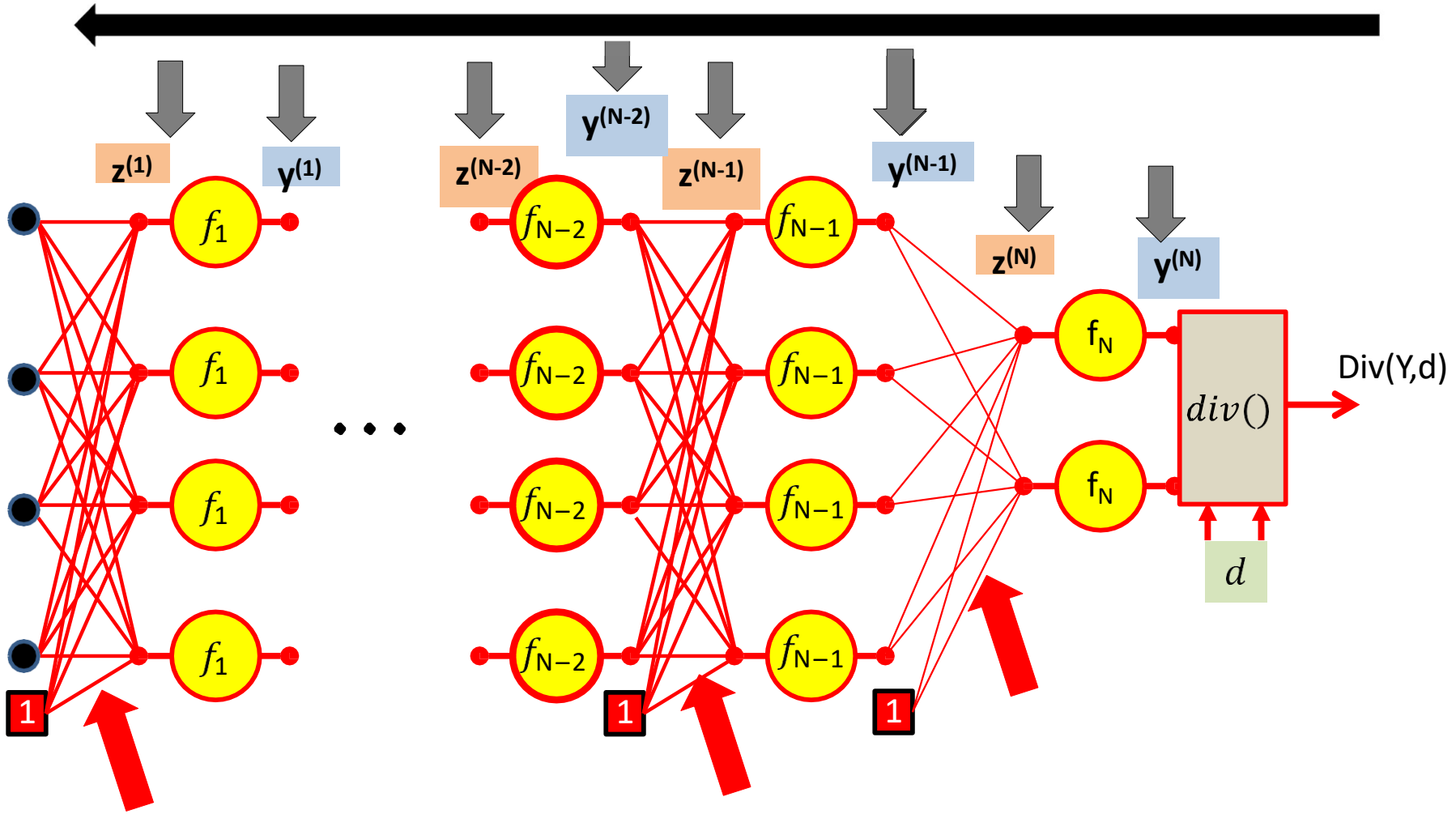
- Gradient descent algorithm:
- Initialize all weights and biases $\left\{w_{ij}^{(k)}\right\}$
  - Using the extended notation: the bias is also a weight
- Do:
  - For every layer k for all i, j, update:
    - $w_{i,j}^{(k)} = w_{i,j}^{(k)} - \eta \frac{dLoss}{dw_{i,j}^{(k)}}$
- Until $Loss$ has converged

Computing derivatives



We have computed all these intermediate values in the forward computation

We must remember them – we will need them to compute the derivatives



What is: $\frac{dDiv(Y,d)}{dw_{i,j}^{(k)}}$