CAP 4453
Robot Vision

Dr. Gonzalo Vaca-Castaño

gonzalo.vacacastano@ucf.edu

Administrative details

- Correction of the midterm exam

Credits

- Some slides comes directly from:
  - Yosesh Rawat
  - Andrew Ng

Robot Vision

17. Introduction to Deep Learning II

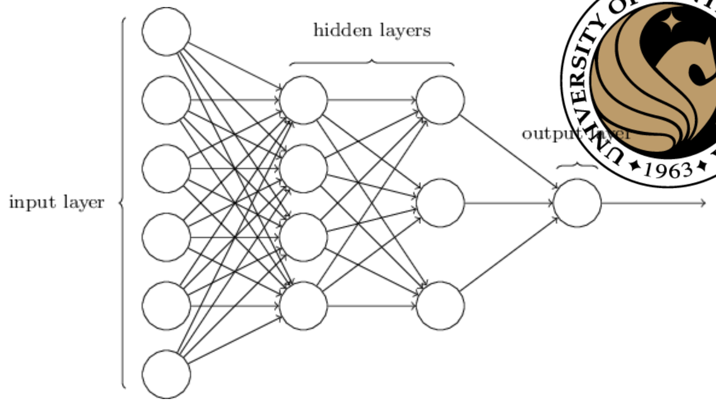
Outline

- Fully connected Neural network
  - Activation functions:
    - Forward and backward
  - Back propagation
  - Network definitions
  - Initialization
  - Training
    - Hyper parameters
    - Gradient updates: RMS prop,
    - Amount of training data
    - Batch normalization
  - Dataset
    - Train set, test set, validation set
    - Bias and variance
- Implementation network to solve digit identification

Fully connected networks: The math

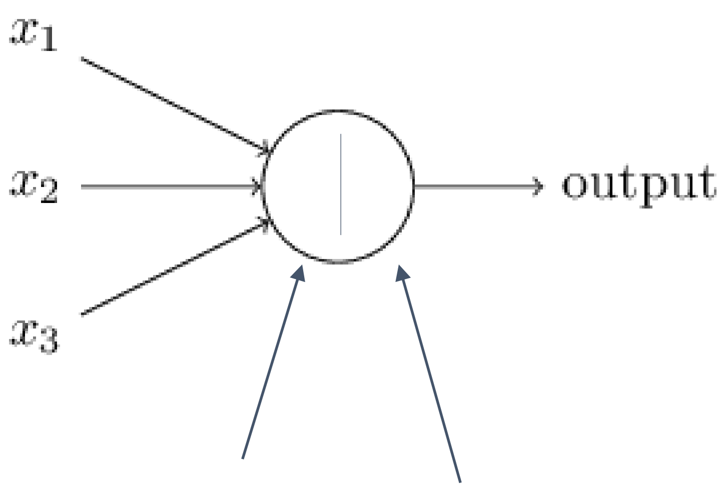
A REVIEW

Fully connected Neural network



- A deep network is a neural network with many layers
- A neuron in a linear function followed for an activation function
- Activation function must be non-linear
- A loss function measures how close is the created function (network) from a desired output
- The “training” is the process of find parameters (‘weights’) that reduces the loss functions
- Updating the weights as $w_{new} = w_{prev} - \alpha \frac{dJ}{dW}$ reduces the loss
- An algorithm named back-propagation allows to compute $\frac{dJ}{dW}$ for all the weights of the network in 2 steps: 1 forward, 1 backward

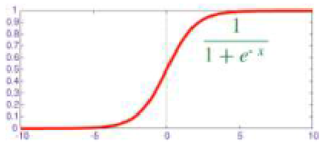
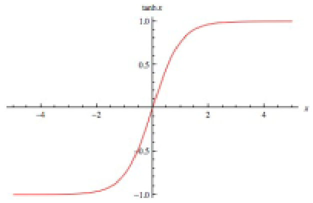
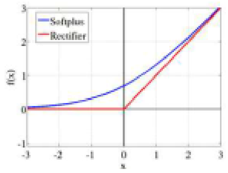
A Neuron
A REVIEW



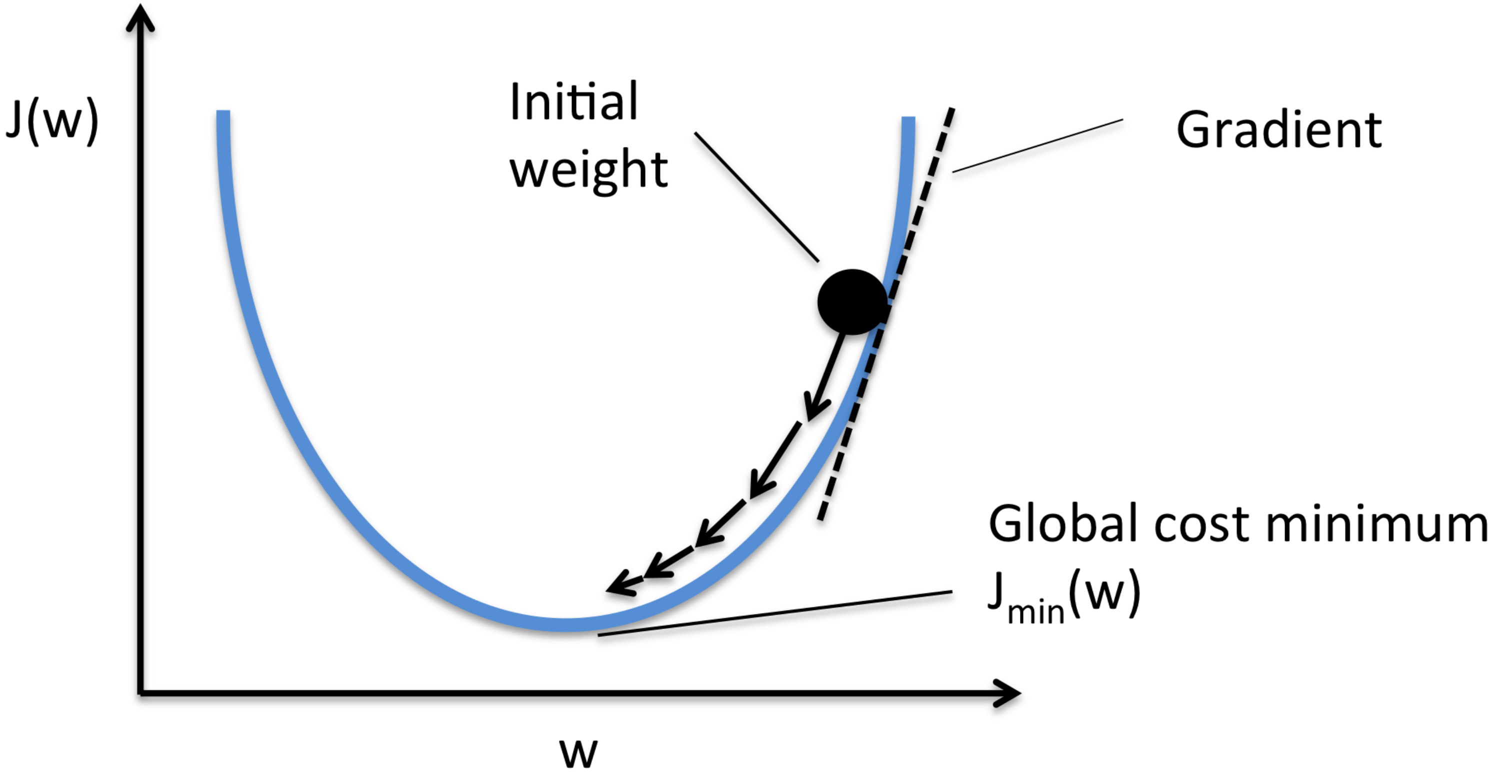
$z = w^T x$ $\quad$ $y = f(z) = f(w^T x)$

$x = [x_1, x_2, x_3, 1]$

Activations and their derivatives

	$f(z) = \dfrac{1}{1+\exp(-z)}$	$f'(z) = f(z)(1 - f(z))$
	$f(z) = \tanh(z)$	$f'(z) = (1 - f^2(z))$
	$f(z) = \begin{cases} 0, & z < 0 \\ z, & z \geq 0 \end{cases}$	This space left intentionally (kind of) blank
	$f(z) = \log(1 + \exp(z))$	$f'(z) = \dfrac{1}{1+\exp(-z)}$

How to minimize a function ?

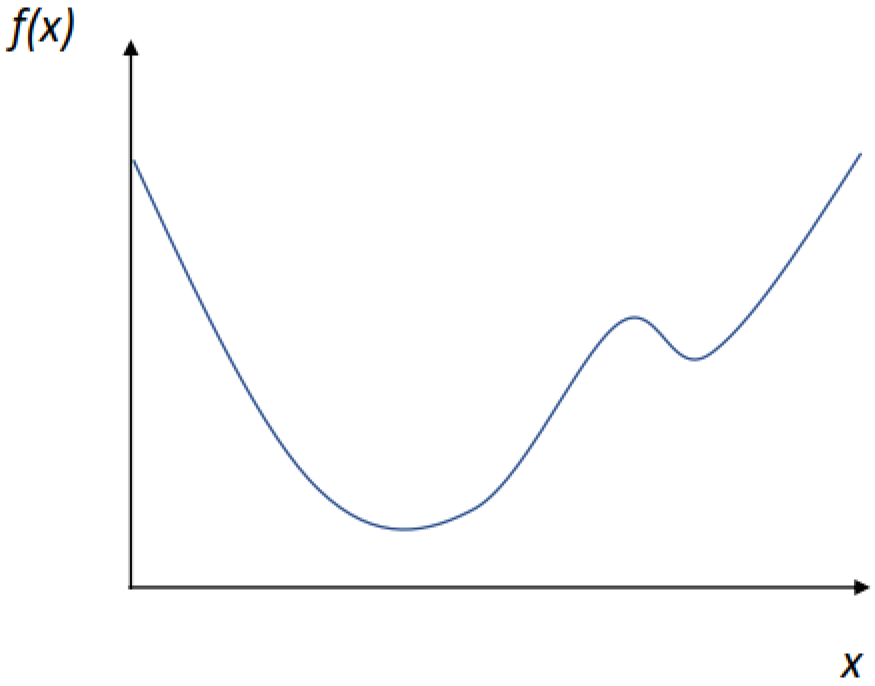


Repeat until there is almost not change

$$w_{new} = w_{prev} - \alpha \frac{dJ}{dW}$$

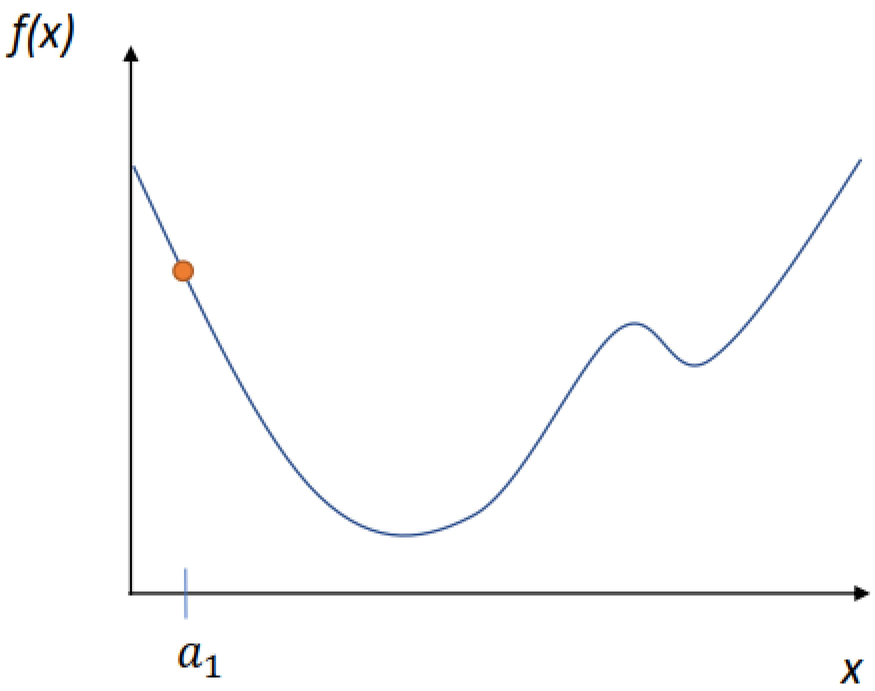
HOW TO COMPUTE THIS GRADIENT?

Gradient descent



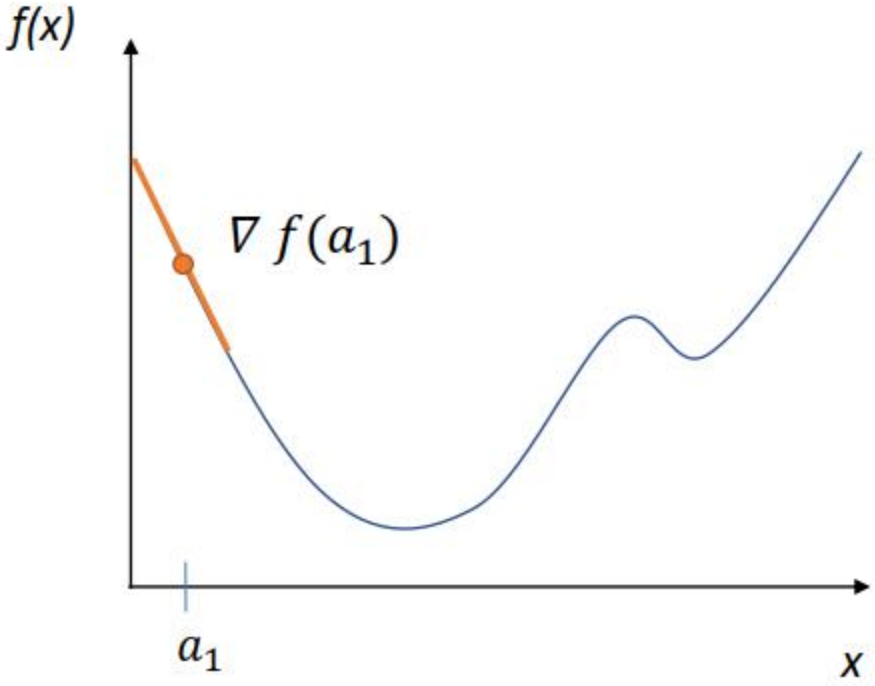
General approach

Pick random starting point.



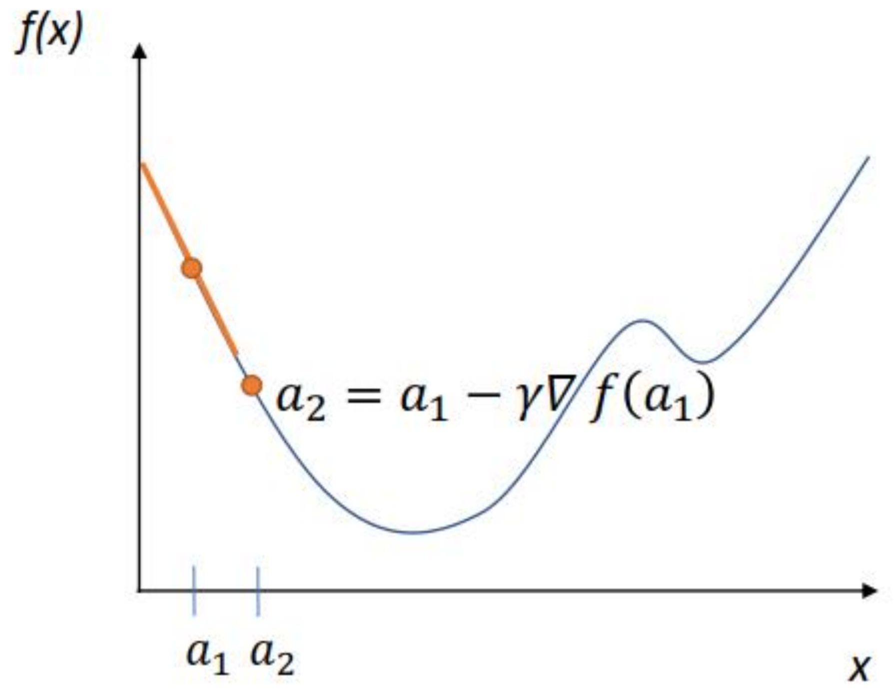
General approach

Compute gradient at point (analytically or by finite differences)



General approach

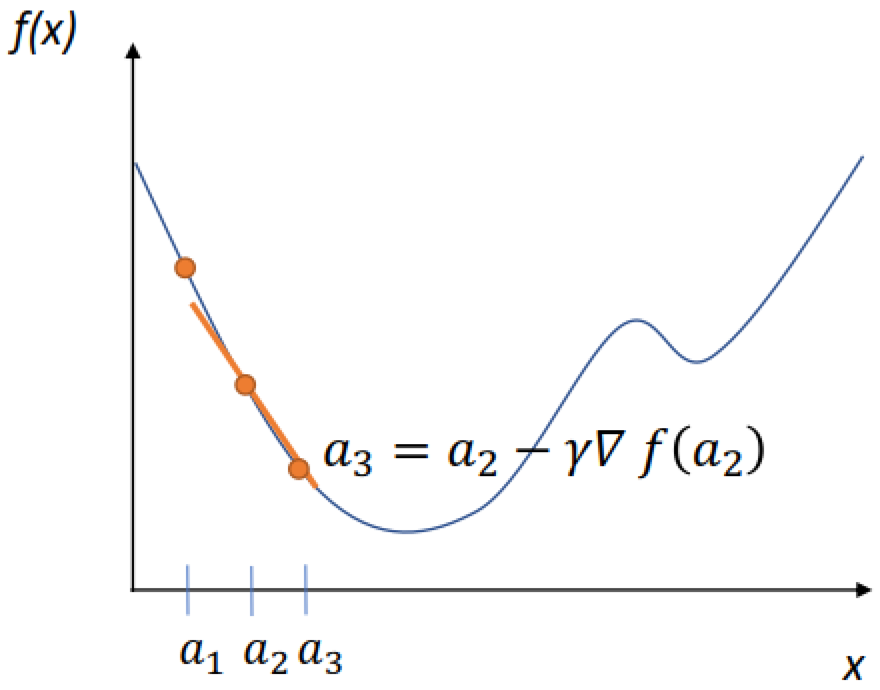
Move along parameter space in direction of negative gradient



γ = amount to move
= *learning rate*

General approach

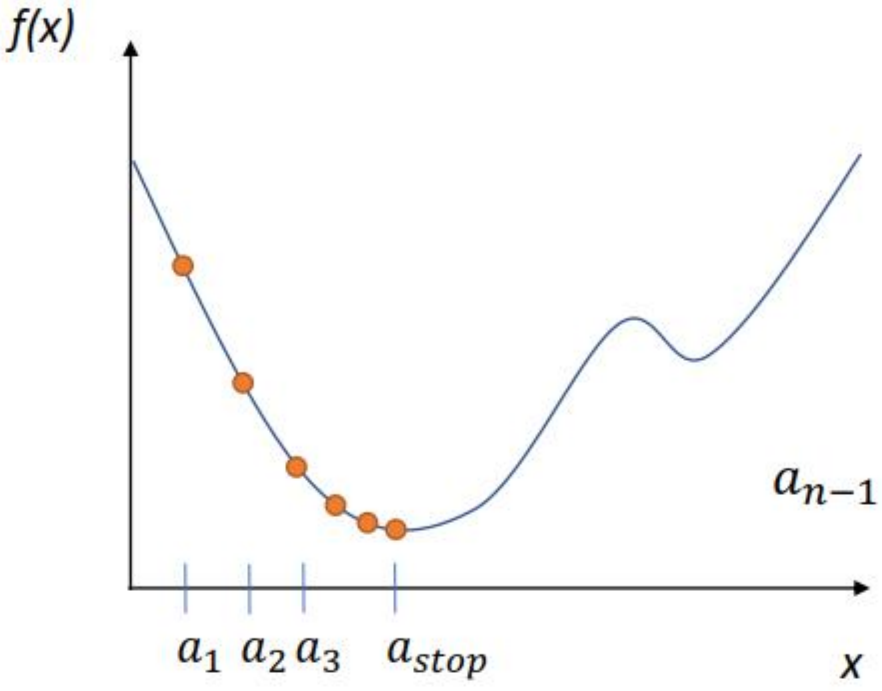
Move along parameter space in direction of negative gradient.



γ = amount to move
= *learning rate*

General approach

Stop when we don't move any more.



a_{stop}:

$$a_{n-1} - \gamma \nabla f(a_{n-1}) = 0$$

Gradient Descent

- The gradient is the direction of fastest increase in J(X)
- Updating the weights as $w_{new} = w_{prev} - \alpha \frac{dJ}{dW}$ reduces the loss

Learning rate

gradient

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 positive derivative → moving left decreases error
      – A negative derivative → moving right decreases error
  – Shift point in this direction

Overall Gradient Descent Algorithm

- Initialize:
  – x^0
  – $k = 0$
- While $|f(x^{k+1}) - f(x^k)| > \varepsilon$
  – $x^{k+1} = x^k - \eta^k \nabla f(x^k)^T$
  – $k = k + 1$

Train with Gradient Descent

- x^i, y^i = n training examples
- $f(\boldsymbol{x})$ = feed forward network
- L($\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta}$) = some *loss function*

Loss function measures how 'good' our network is at classifying the training examples wrt. the parameters of the model (the perceptron weights).

Loss Function

- Way to define how good the network is performing
  - In terms of prediction
- Network training (Optimization)
  - Find the best network parameters to minimize the loss

input

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), yi)$$

Ground truth

Loss function

network

Network parameters

Loss Functions

- Cross entropy

Ground-truth

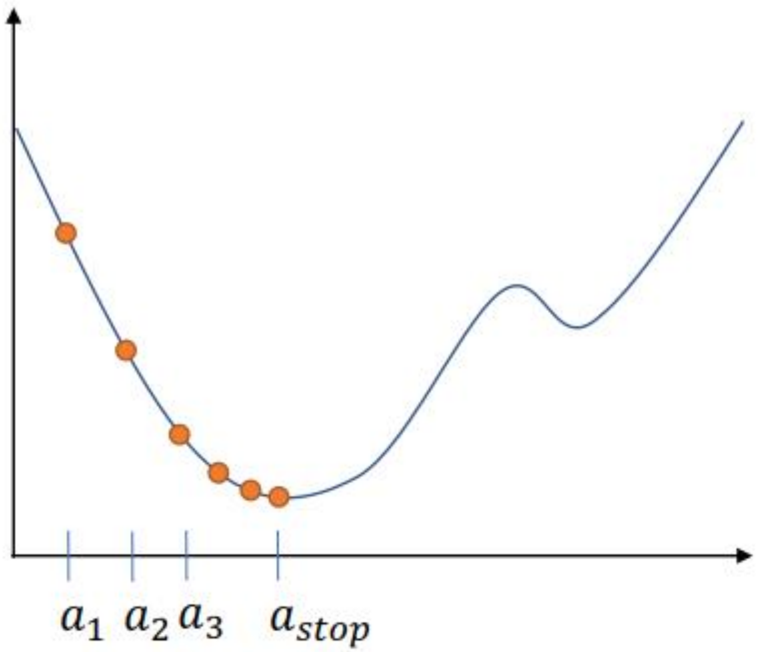
Predicted value

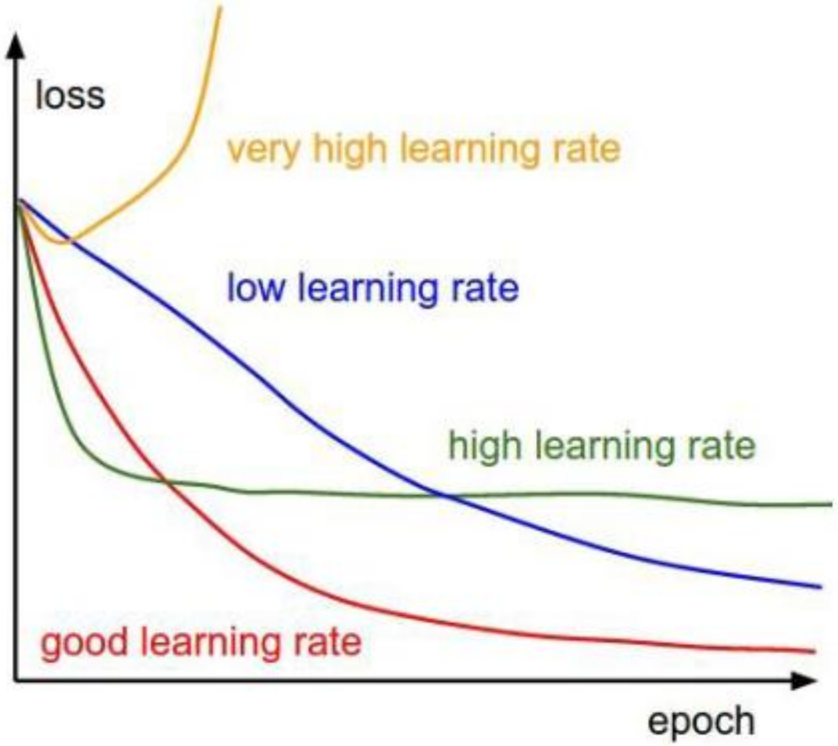
$$-\frac{1}{N}\sum_{i=1}^{N}(y_i log(\hat{y}_i) + (1 - y_i) log(1 - \hat{y}_i))$$

- Mean squared error (MSE)

$$\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

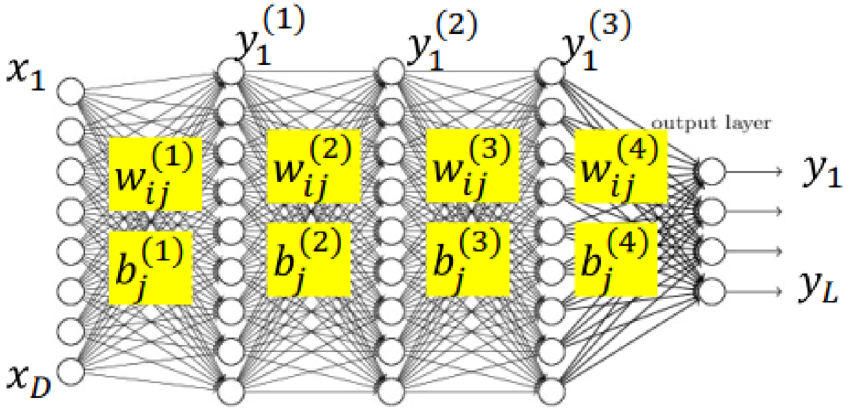
Learning rate





https://www.cs.cmu.edu/~bhiksha/courses/deeplearning/Fa
5/

Notation



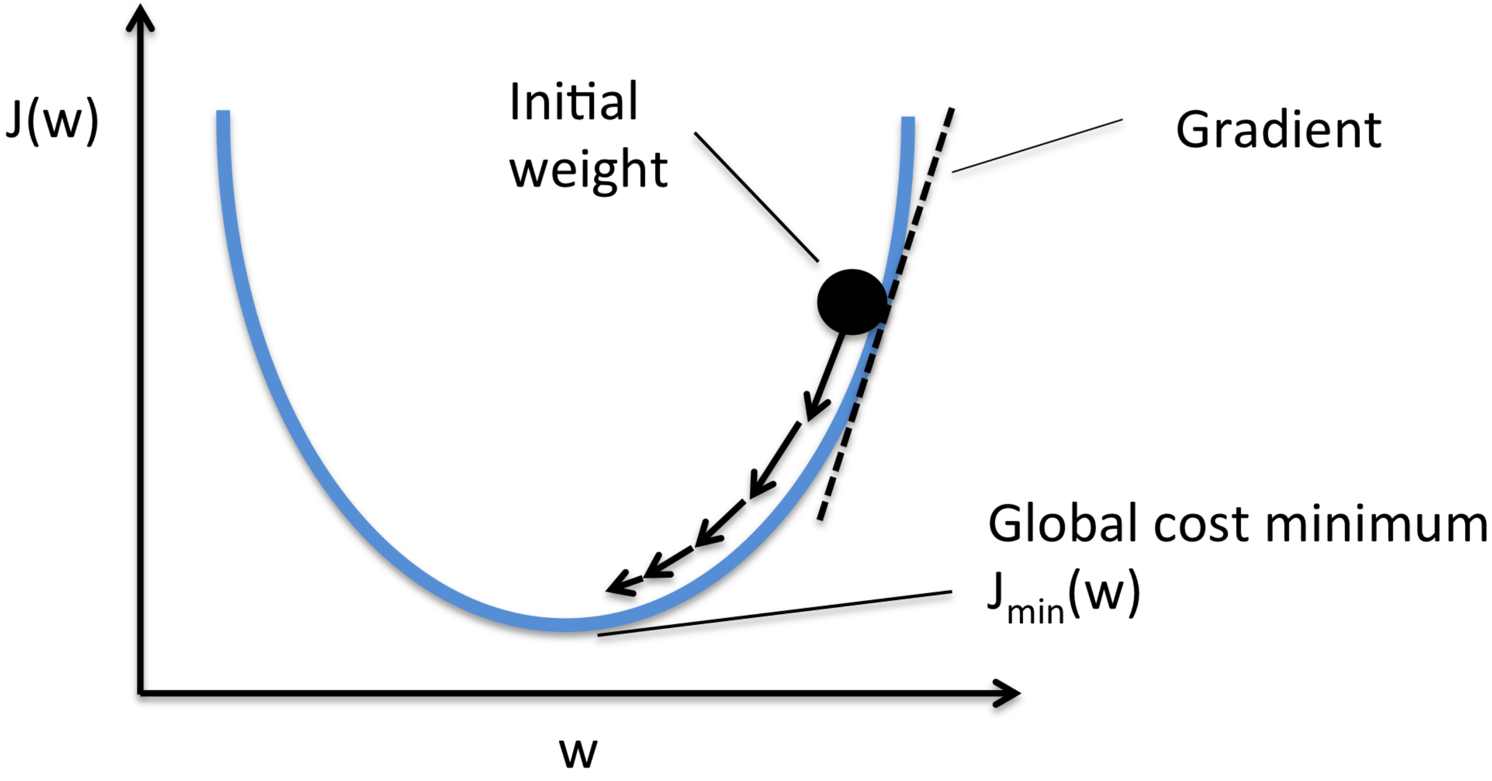
- The input layer is the 0th layer
- We will represent the output of the i-th perceptron of the kth layer as $y_i^{(k)}$
  - **Input to network:** $y_i^{(0)} = x_i$
  - **Output of network:** $y_i = y_i^{(N)}$
- We will represent the weight of the connection between the i-th unit of the k-1th layer and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

Training steps

- Define network
- Loss function
- Initialize network parameters
- Get training data
  - Prepare batches
- Feedforward one batch
  - Compute loss
  - Update network parameters
  - Repeat

IN OUR CASE THE LOSS FUNCTION

How to minimize a function ?



Repeat until there is almost not change

$$w_{new} = w_{prev} - \alpha \frac{dJ}{dW}$$

HOW TO COMPUTE THIS GRADIENT?

Training Neural Nets through Gradient Descent

Total training error:

$$Err = \frac{1}{T}\sum_t Div(\mathbf{Y}_t, \mathbf{d}_t)$$

- Gradient descent algorithm:
- Initialize all weights and biases $\left\{w_{ij}^{(k)}\right\}$
  - Using the extended notation: the bias is also a weight

Assuming the bias is also represented as 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

Example: L2

$$Div = \frac{1}{2}(y_t - d_t)^2$$

$$\frac{dDiv}{dy_i} = (y_t - d_t)$$

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:

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

The derivative

Total training error:

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

Total derivative:

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

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

Calculus Refresher: Basic rules of calculus

For any differentiable function

$$y = f(x)$$

with derivative

$$\frac{dy}{dx}$$

the following must hold for sufficiently small Δx $\Rightarrow$ $\Delta y \approx \frac{dy}{dx}\Delta x$

For any differentiable function

$$y = f(x_1, x_2, \dots, x_M)$$

with partial derivatives

$$\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}, \dots, \frac{\partial y}{\partial x_M}$$

the following must hold for sufficiently small $\Delta x_1, \Delta x_2, \dots, \Delta x_M$

$$\Delta y \approx \frac{\partial y}{\partial x_1}\Delta x_1 + \frac{\partial y}{\partial x_2}\Delta x_2 + \cdots + \frac{\partial y}{\partial x_M}\Delta x_M$$

Calculus Refresher: Chain rule

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

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

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

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

$$y = f(z) \Rightarrow \Delta y = \frac{dy}{dz}\Delta z = \frac{dy}{dz}\frac{dg(x)}{dx}\Delta x \checkmark$$

Calculus Refresher: Distributed Chain rule

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

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

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

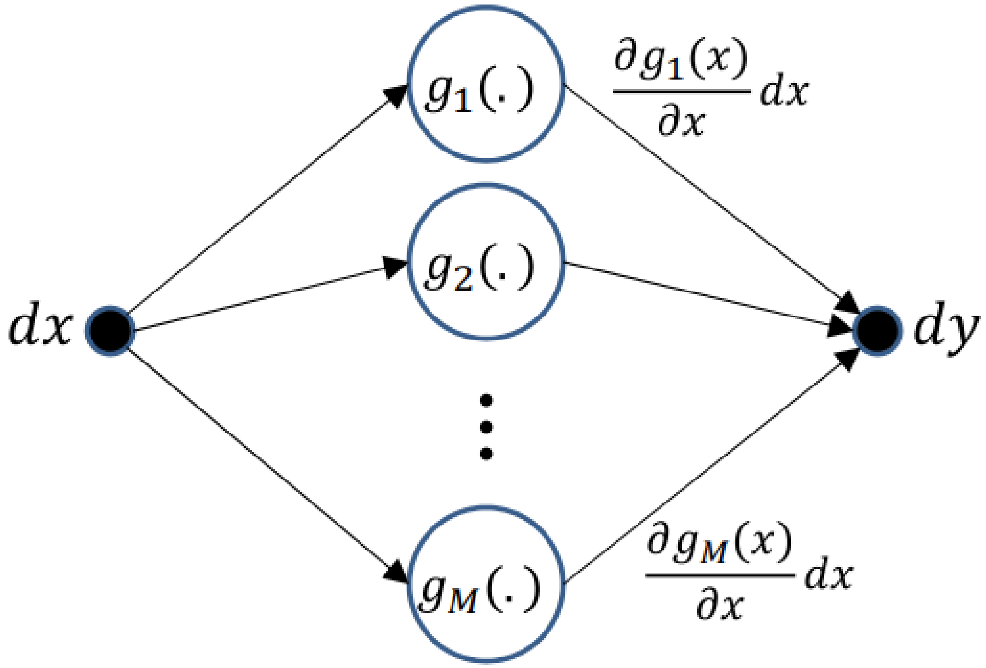
$$\Delta y = \frac{\partial y}{\partial g_1(x)}\Delta g_1(x) + \frac{\partial y}{\partial g_2(x)}\Delta g_2(x) + \cdots + \frac{\partial y}{\partial g_M(x)}\Delta g_M(x)$$

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

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

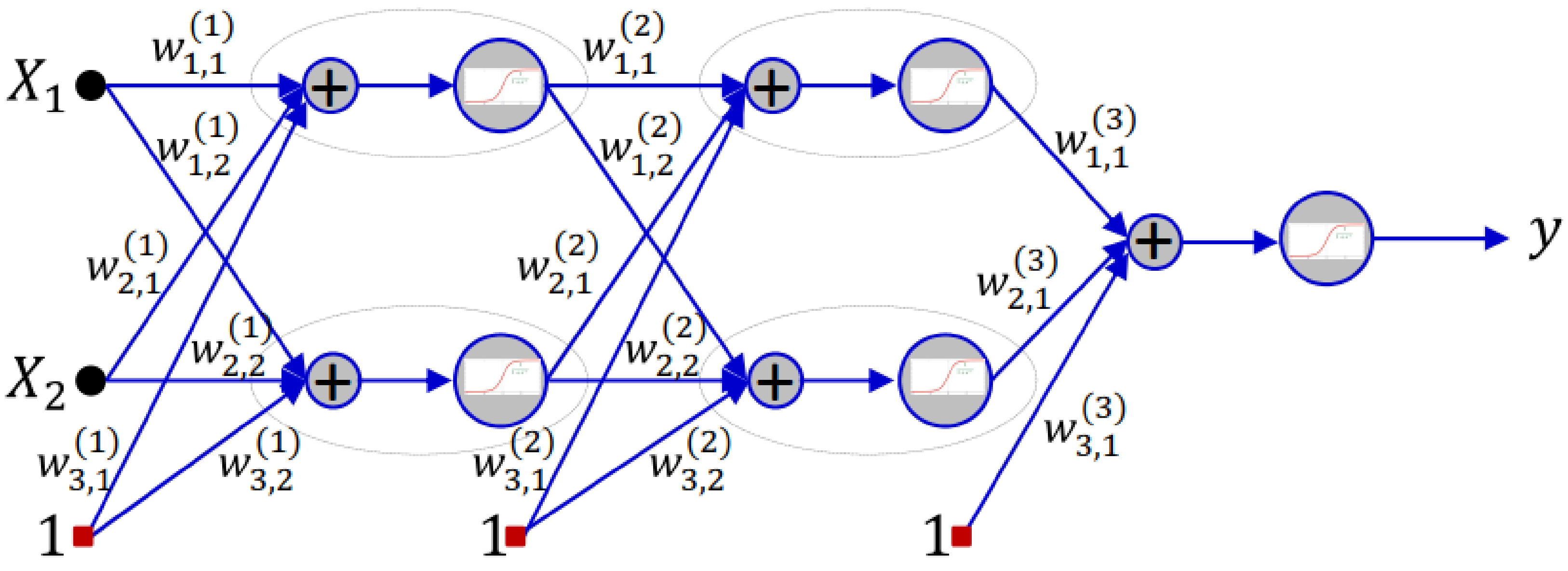
96

Distributed Chain Rule: Influence Diagram



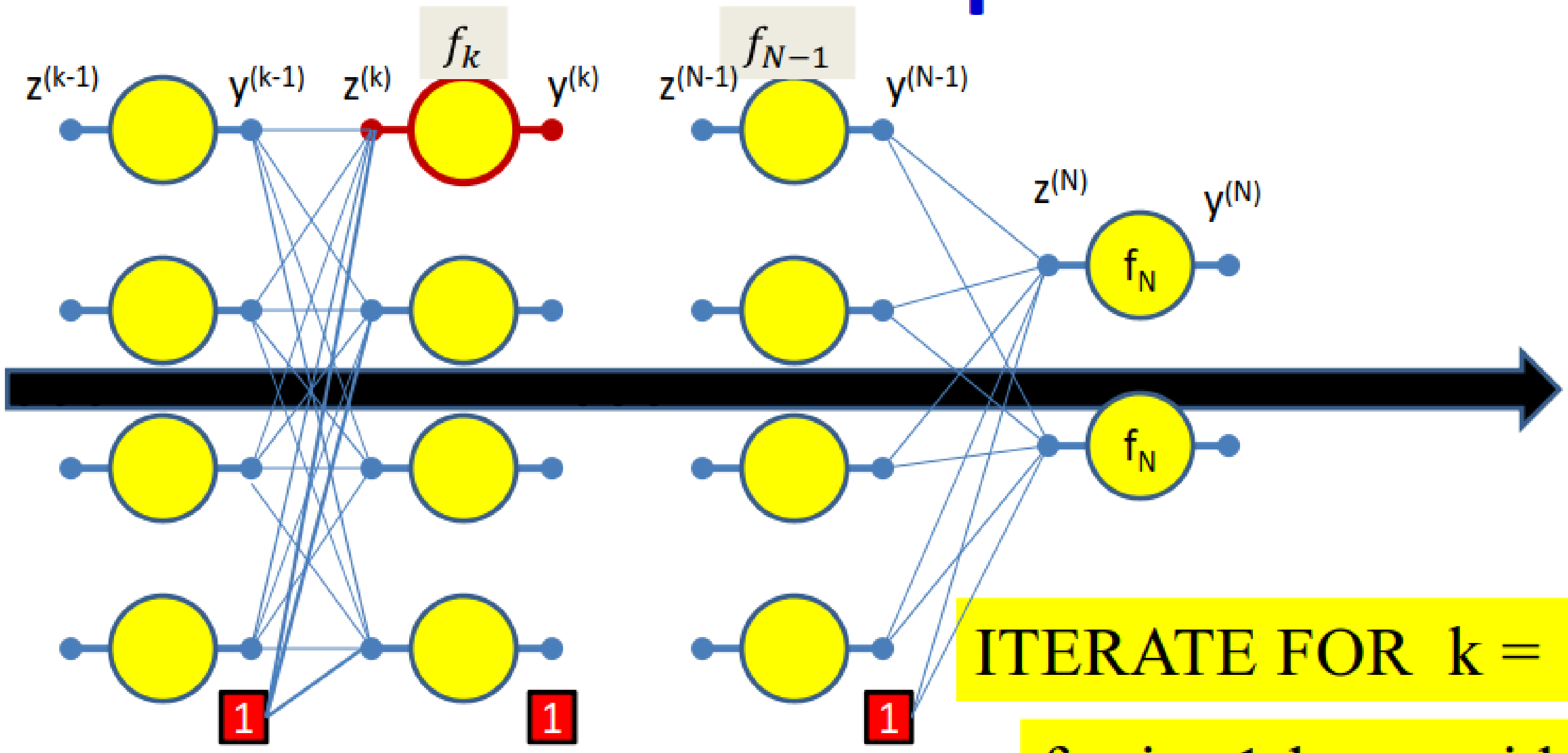
- Small perturbations in x cause small perturbations in each of $g_1 \dots g_M$, each of which individually additively perturbs y

A first closer look at the network



- Showing a tiny 2-input network for illustration
  - Actual network would have many more neurons and inputs
- Expanded **with all weights and activations shown**
- The overall function is differentiable w.r.t every weight, bias and input

Forward Computation



ITERATE FOR k = 1:N

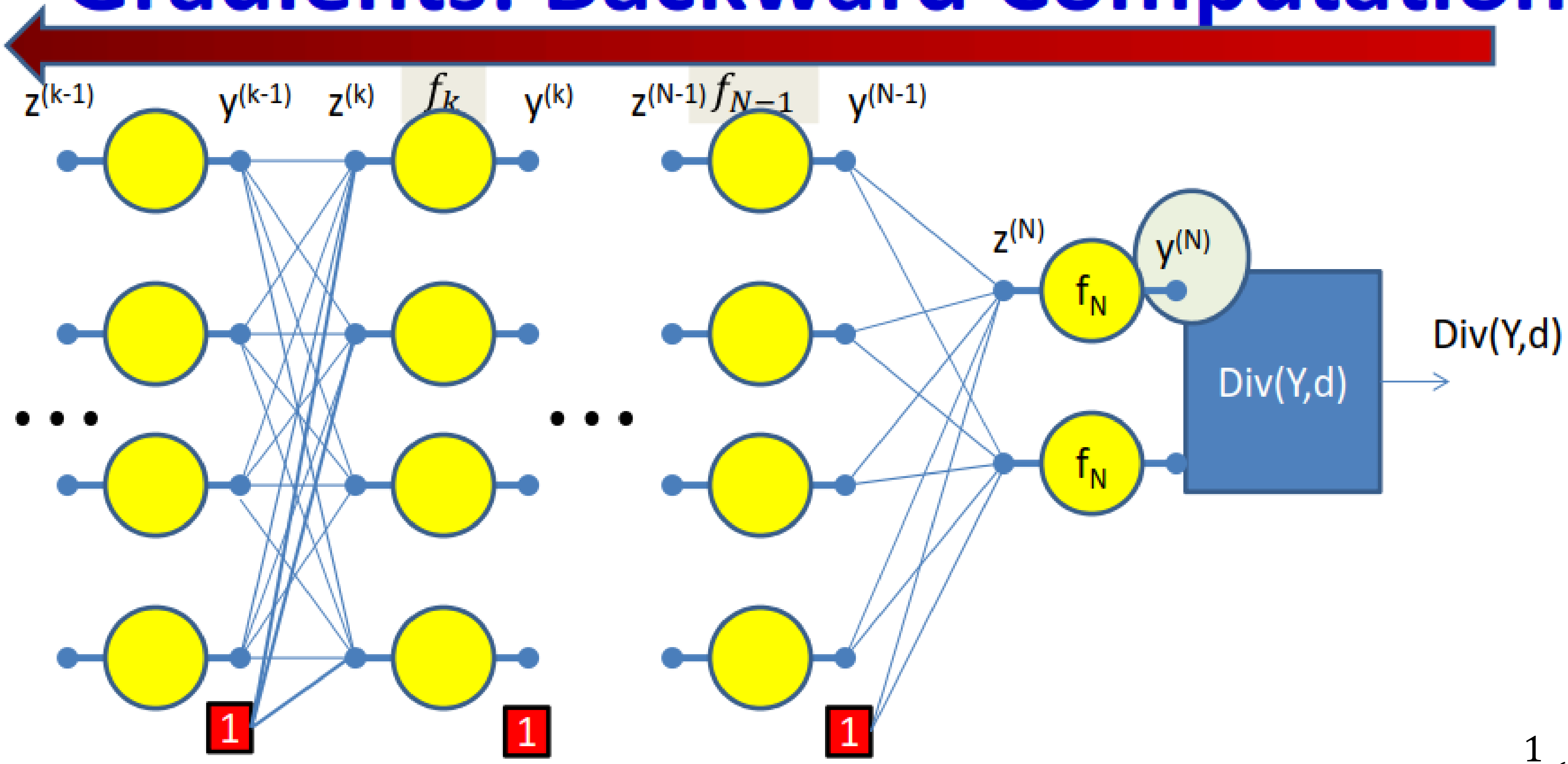
for j = 1:layer-width

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

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

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

Gradients: Backward Computation

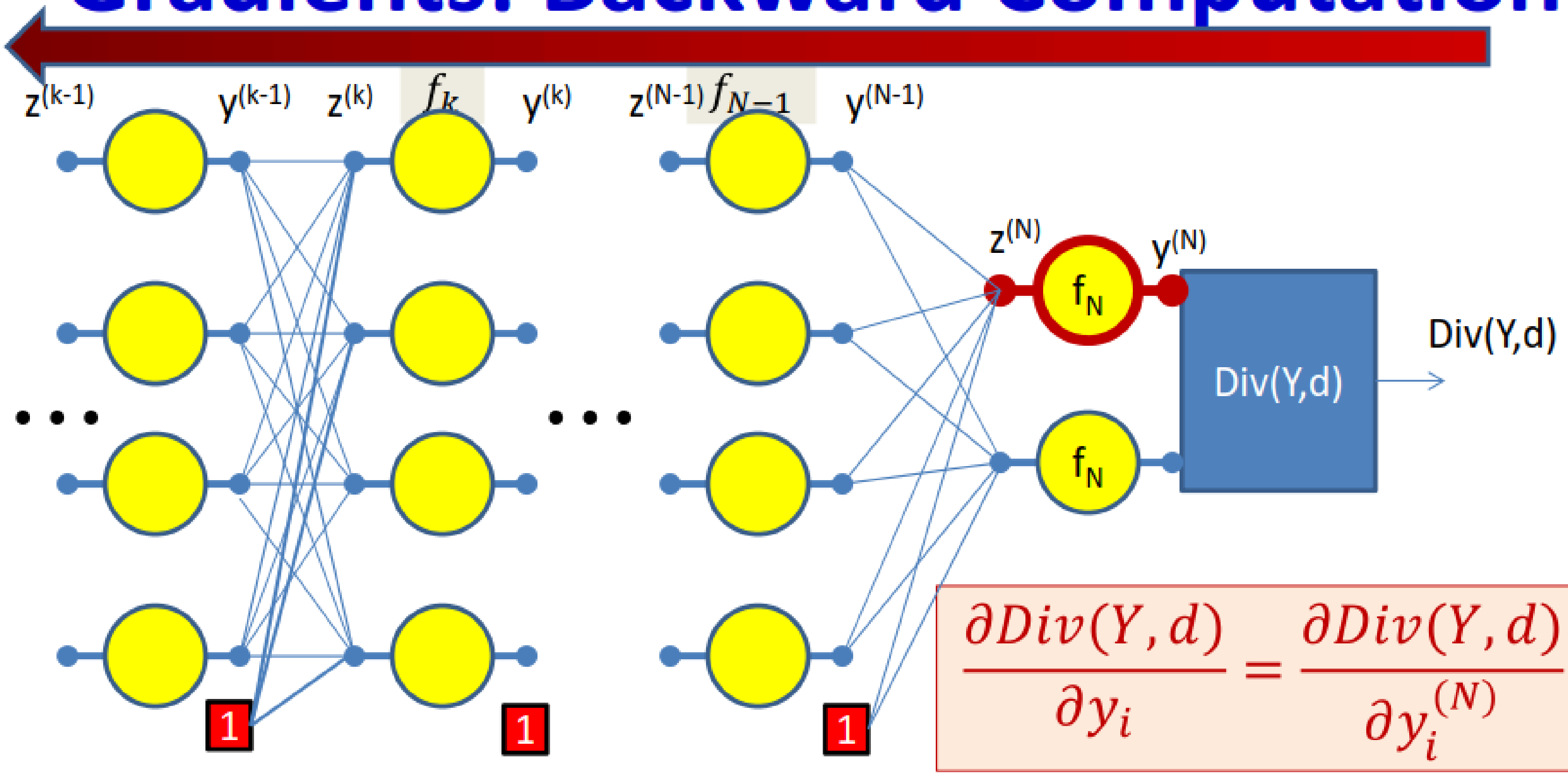


$$\frac{\partial Div(Y,d)}{\partial y_i} = \frac{\partial Div(Y,d)}{\partial y_i^{(N)}}$$

$$Div = \frac{1}{2}(y_t - d_t)^2$$

$$\frac{dDiv}{dy_i} = (y_t - d_t)$$

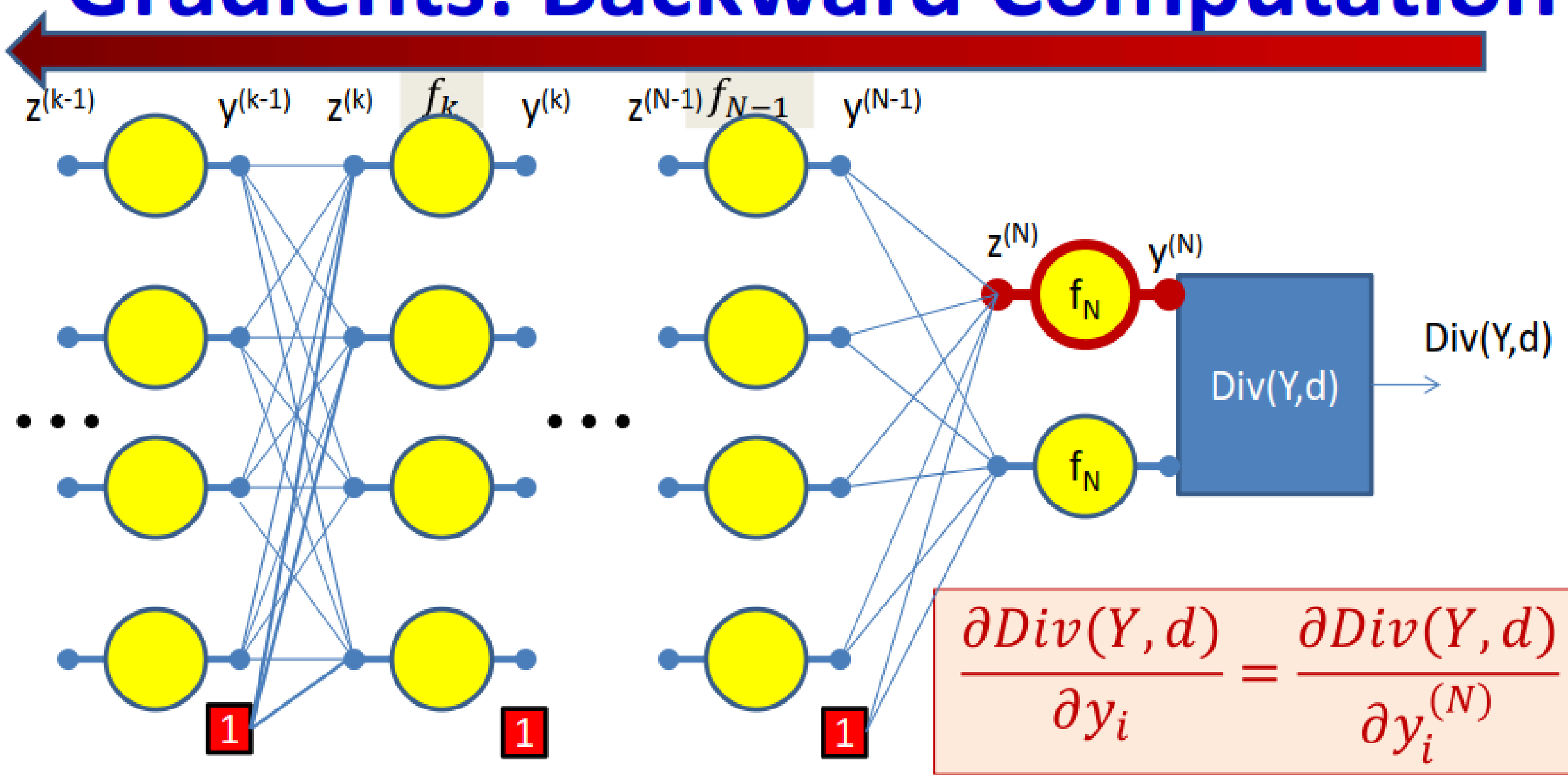
Gradients: Backward Computation



$$\frac{\partial Div(Y,d)}{\partial y_i} = \frac{\partial Div(Y,d)}{\partial y_i^{(N)}}$$

$$\frac{\partial Div}{\partial z_i^{(N)}} = \frac{\partial y_i^{(N)}}{\partial z_i^{(N)}} \frac{\partial Div}{\partial y_i} = f_N'\left(z_i^{(N)}\right) \frac{\partial Div}{\partial y_i^{(N)}}$$

Gradients: Backward Computation



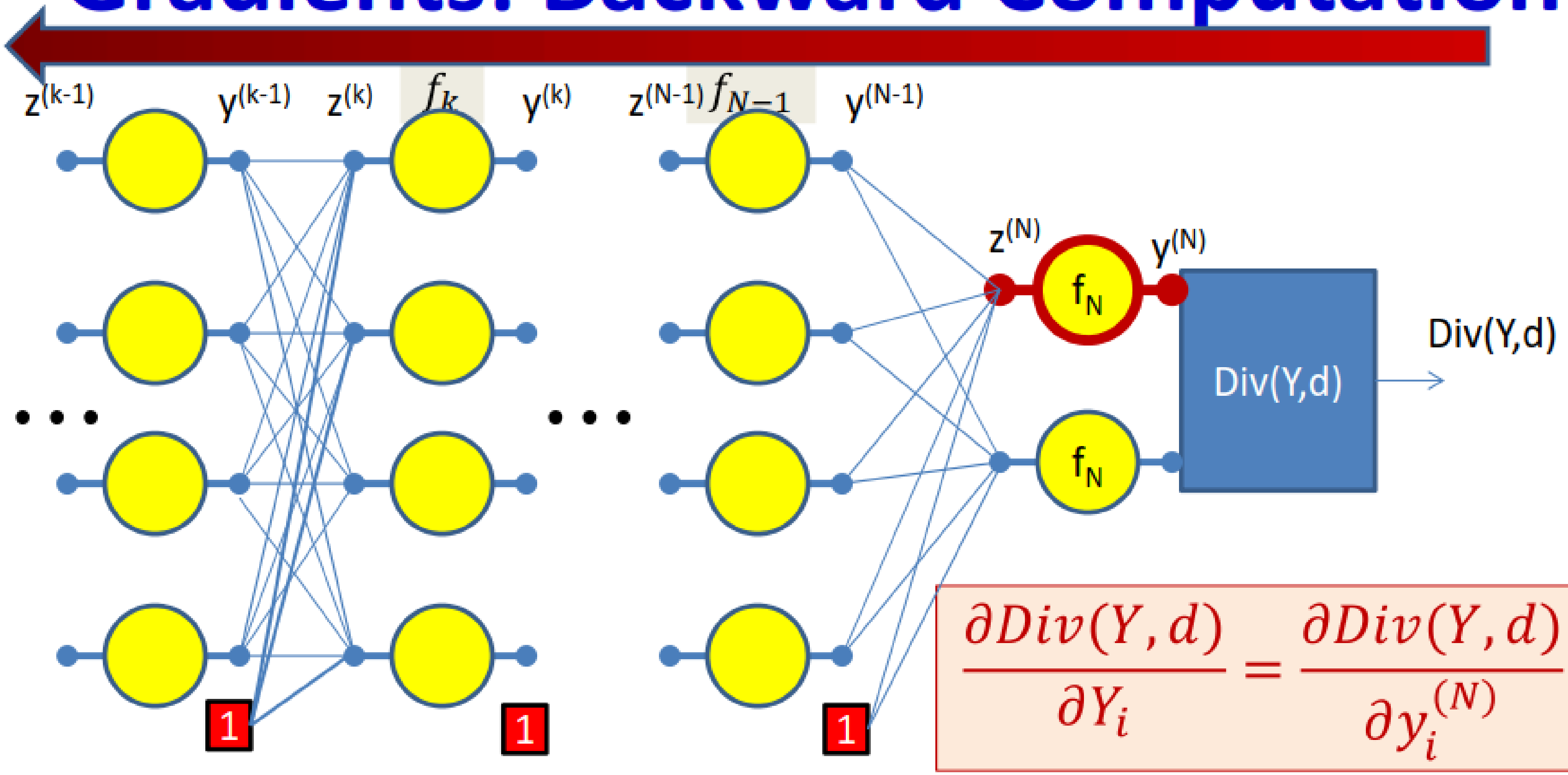
$$\frac{\partial Div(Y,d)}{\partial y_i} = \frac{\partial Div(Y,d)}{\partial y_i^{(N)}}$$

$$y_i^{[N]} = f(z_i^{[N]})$$

$$\frac{\partial y_i^{[N]}}{\partial z_i^{[N]}} = f^{[N]\prime}(z_i^{[N]})$$

$$\frac{\partial Div}{\partial z_i^{(N)}} = \frac{\partial y_i^{(N)}}{\partial z_i^{(N)}} \frac{\partial Div}{\partial y_i} = f_N'\left(z_i^{(N)}\right) \frac{\partial Div}{\partial y_i^{(N)}}$$

Gradients: Backward Computation

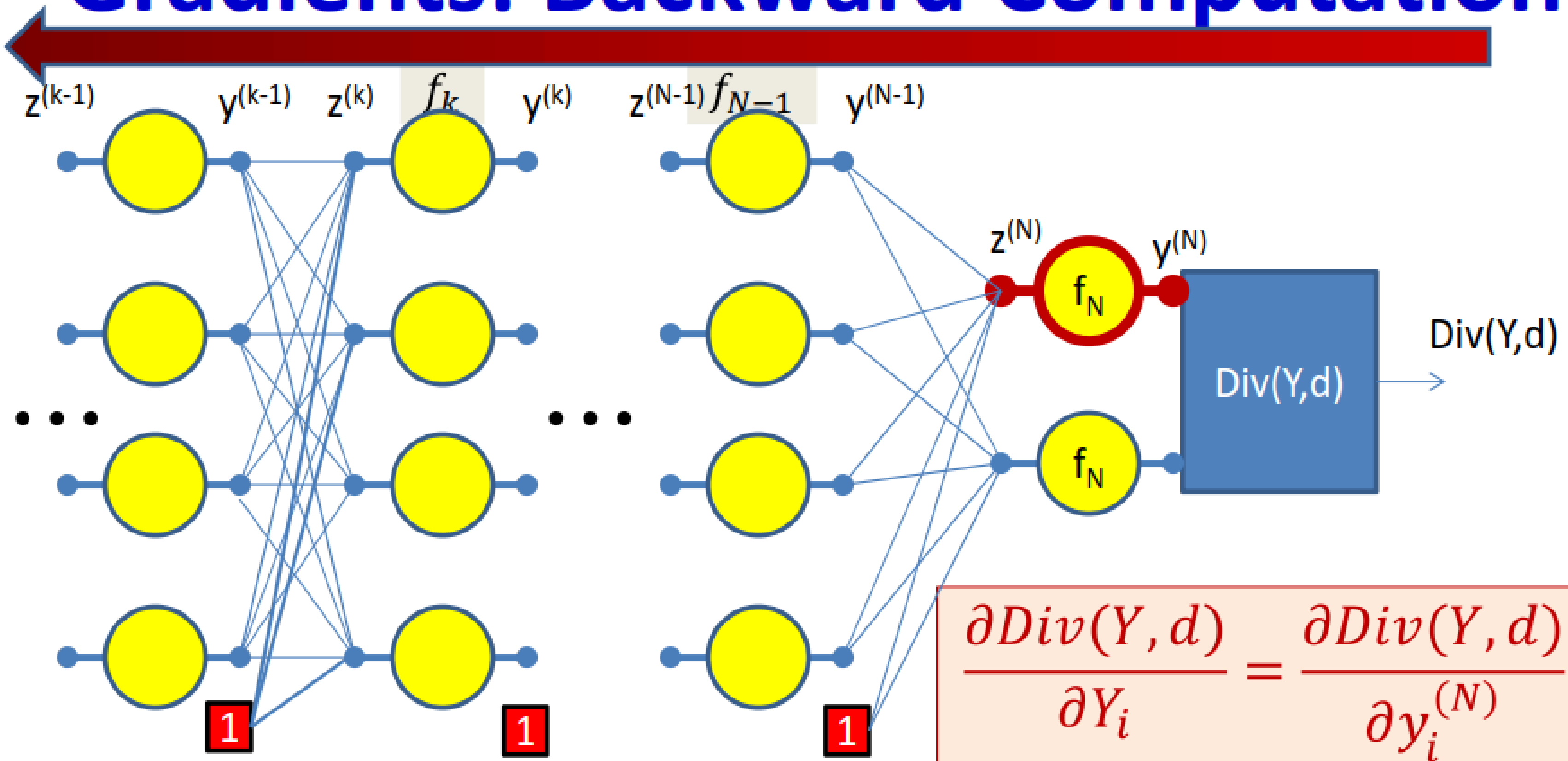


$$\frac{\partial Div(Y,d)}{\partial Y_i} = \frac{\partial Div(Y,d)}{\partial y_i^{(N)}}$$

$z_i^{(N)}$ computed during the forward pass

$$\frac{\partial Div}{\partial z_i^{(N)}} = \frac{\partial y_i^{(N)}}{\partial z_i^{(N)}} \frac{\partial Div}{\partial Y_i} = f_N'\left(z_i^{(N)}\right) \frac{\partial Div}{\partial y_i^{(N)}}$$

Gradients: Backward Computation

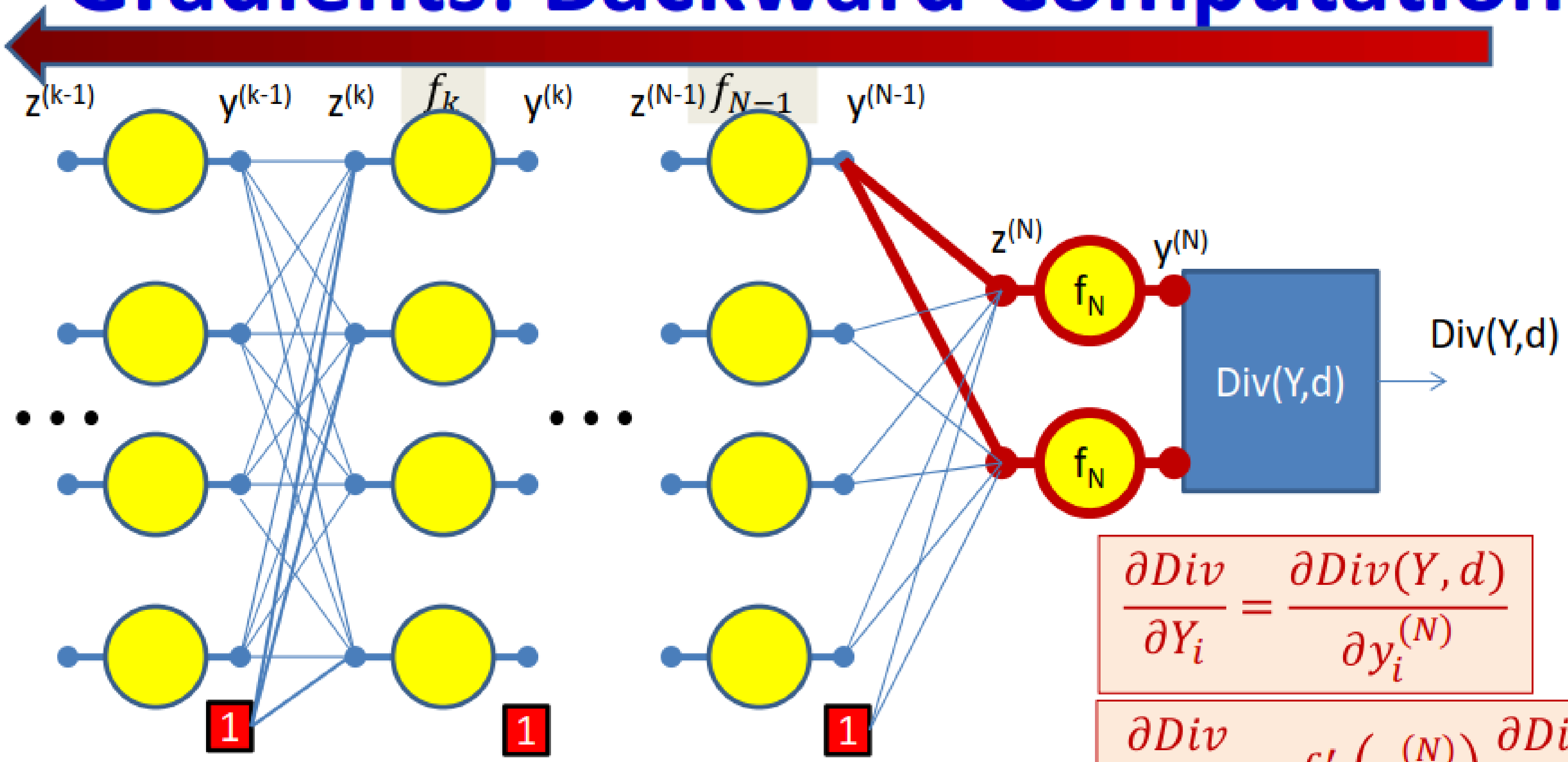


$$\frac{\partial Div(Y,d)}{\partial Y_i} = \frac{\partial Div(Y,d)}{\partial y_i^{(N)}}$$

Derivative of the activation function of Nth layer

$$\frac{\partial Div}{\partial z_i^{(N)}} = \frac{\partial y_i^{(N)}}{\partial z_i^{(N)}} \frac{\partial Div}{\partial Y_i} = f_N'\left(z_i^{(N)}\right) \frac{\partial Div}{\partial y_i^{(N)}}$$

Gradients: Backward Computation



$$\frac{\partial Div}{\partial Y_i} = \frac{\partial Div(Y,d)}{\partial y_i^{(N)}}$$

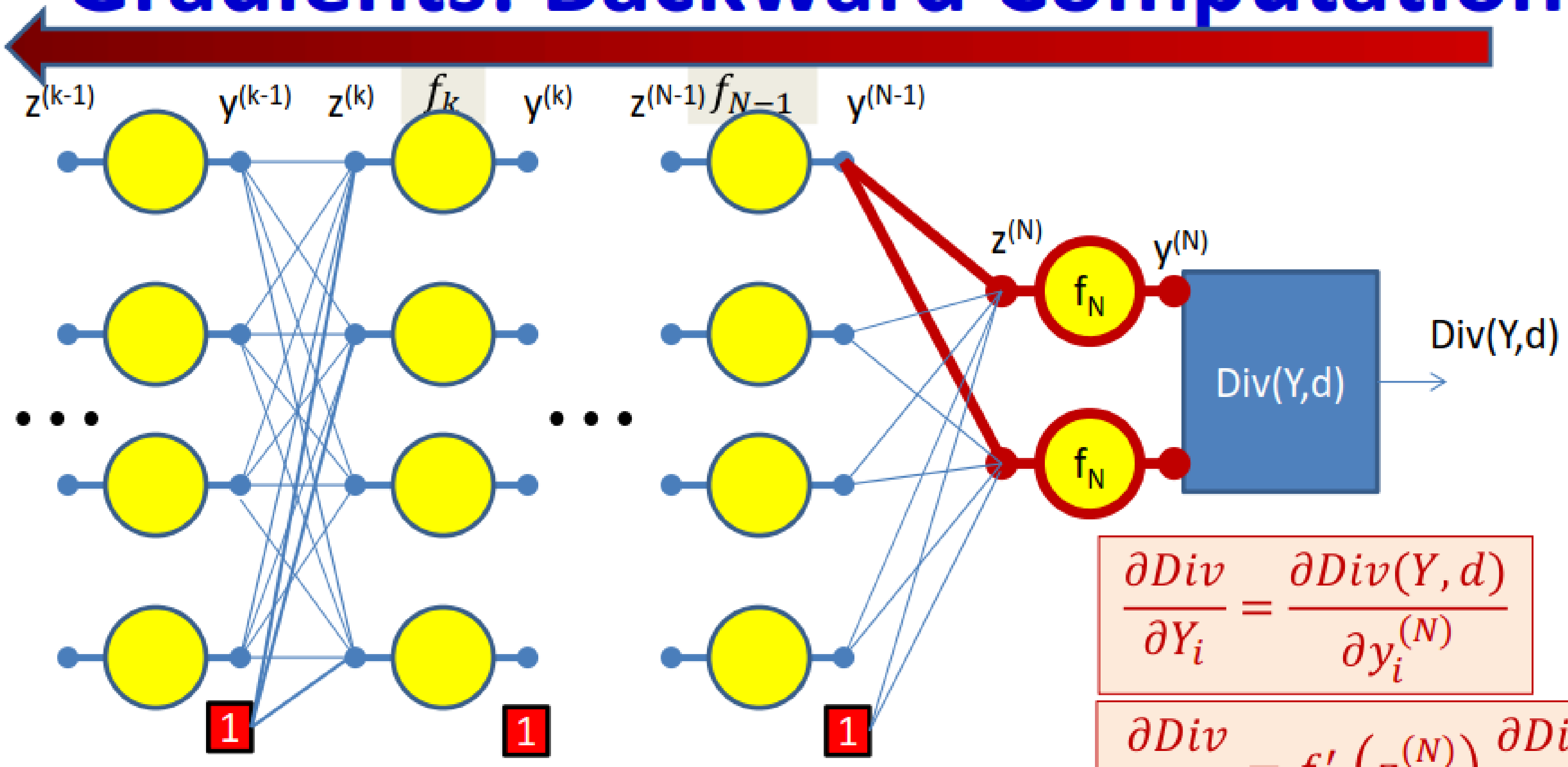
$$\frac{\partial Div}{\partial z_i^{(N)}} = f_N'\left(z_i^{(N)}\right)\frac{\partial Div}{\partial y_i^{(N)}}$$

$$\frac{\partial Div}{\partial y_i^{(N-1)}} = \sum_j \frac{\partial z_j^{(N)}}{\partial y_i^{(N-1)}} \frac{\partial Div}{\partial z_j^{(N)}} = \sum_j w_{ij}^{(N)} \frac{\partial Div}{\partial z_j^{(N)}}$$

Because :

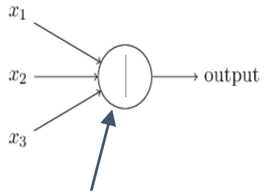
$$\frac{\partial z_j^{(N)}}{\partial y_i^{(N-1)}} = w_{ij}^{(N)}$$

Gradients: Backward Computation



$$\frac{\partial Div}{\partial Y_i} = \frac{\partial Div(Y,d)}{\partial y_i^{(N)}}$$

$$\frac{\partial Div}{\partial z_i^{(N)}} = f_N'\left(z_i^{(N)}\right)\frac{\partial Div}{\partial y_i^{(N)}}$$



$z = w^T x$

$$\frac{\partial Div}{\partial y_i^{(N-1)}} = \sum_j \frac{\partial z_j^{(N)}}{\partial y_i^{(N-1)}} \frac{\partial Div}{\partial z_j^{(N)}} = \sum_j w_{ij}^{(N)} \frac{\partial Div}{\partial z_j^{(N)}}$$

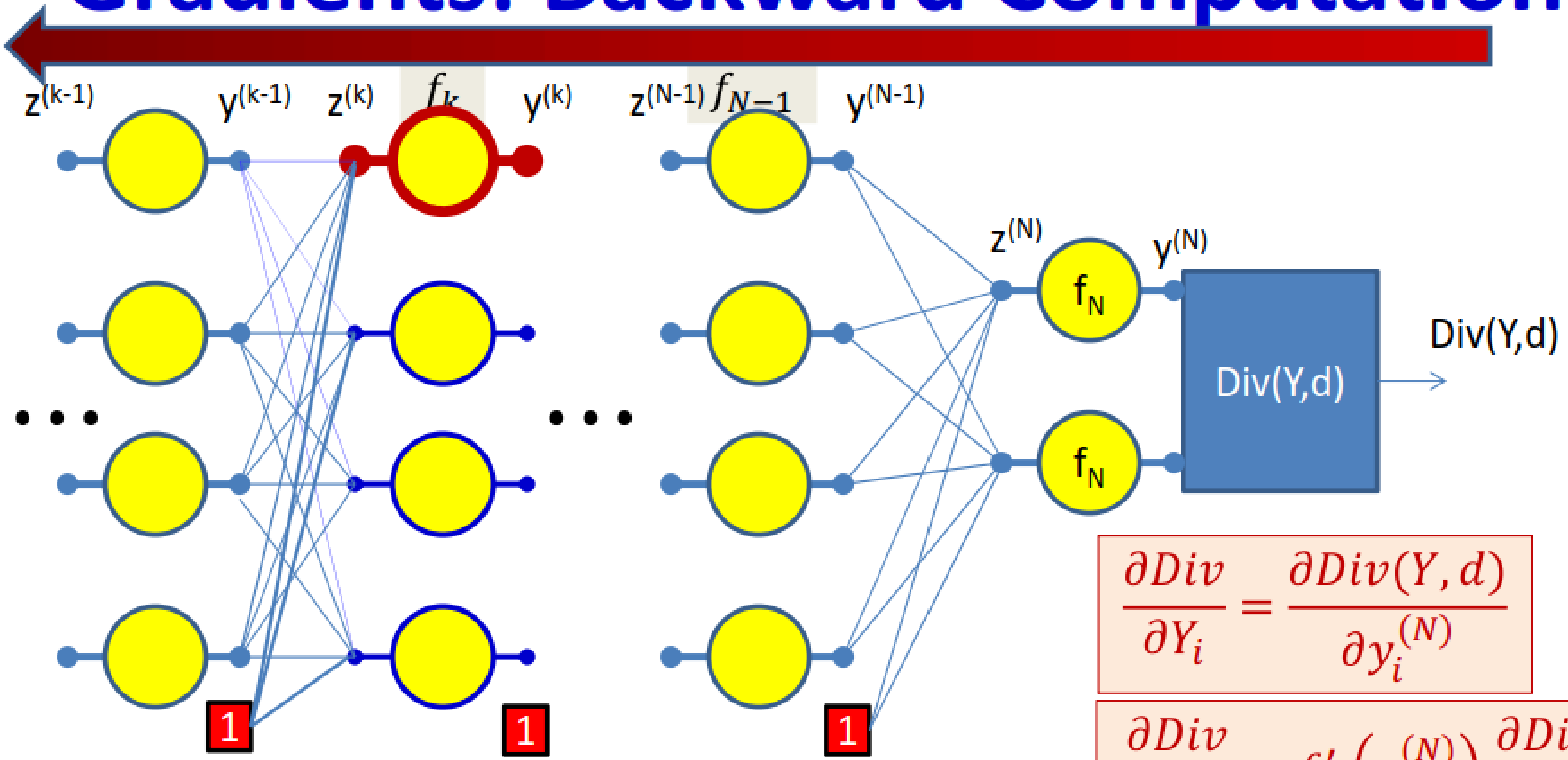
Because :

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

But In this case the input is the output from previous layer

$$z_j^{[N]} = w^T y_i^{[N-1]}$$

Gradients: Backward Computation



$$\frac{\partial Div}{\partial Y_i} = \frac{\partial Div(Y,d)}{\partial y_i^{(N)}}$$

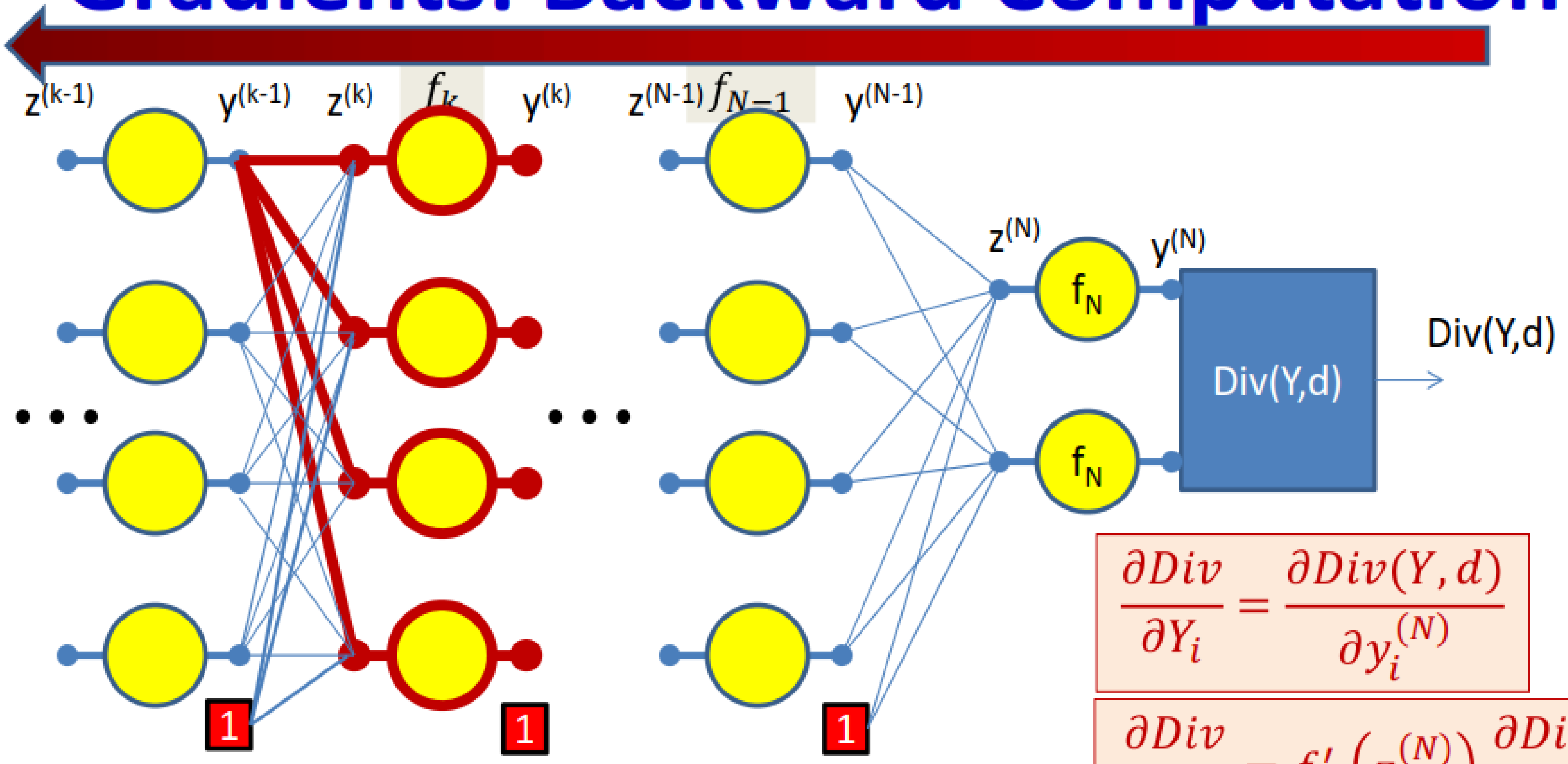
$$\frac{\partial Div}{\partial z_i^{(N)}} = f_N'\left(z_i^{(N)}\right)\frac{\partial Div}{\partial y_i^{(N)}}$$

$$\frac{\partial Div}{\partial y_i^{(N-1)}} = \sum_j w_{ij}^{(N)} \frac{\partial Div}{\partial z_j^{(N)}}$$

computed during the forward pass

$$\frac{\partial Div}{\partial z_i^{(k)}} = f_k'\left(z_i^{(k)}\right)\frac{\partial Div}{\partial y_i^{(k)}}$$

Gradients: Backward Computation

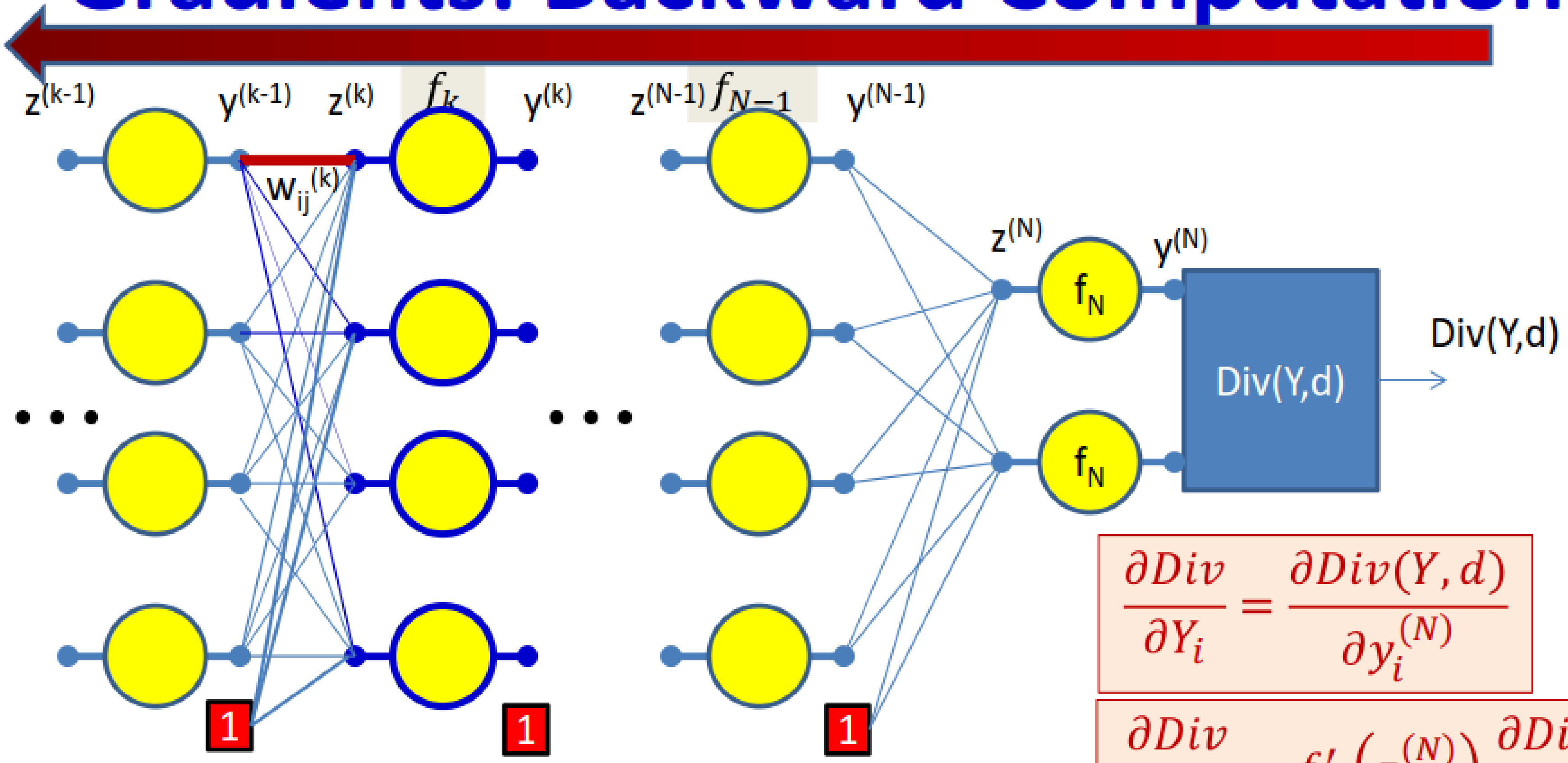


$$\frac{\partial Div}{\partial Y_i} = \frac{\partial Div(Y,d)}{\partial y_i^{(N)}}$$

$$\frac{\partial Div}{\partial z_i^{(N)}} = f_N'\left(z_i^{(N)}\right)\frac{\partial Div}{\partial y_i^{(N)}}$$

$$\frac{\partial Div}{\partial y_i^{(k-1)}} = \sum_j \frac{\partial z_j^{(k)}}{\partial y_i^{(k-1)}} \frac{\partial Div}{\partial z_j^{(k)}} = \sum_j w_{ij}^{(k)} \frac{\partial Div}{\partial z_j^{(k)}}$$

Gradients: Backward Computation



$$\frac{\partial Div}{\partial Y_i} = \frac{\partial Div(Y,d)}{\partial y_i^{(N)}}$$

$$\frac{\partial Div}{\partial z_i^{(N)}} = f_N'\left(z_i^{(N)}\right)\frac{\partial Div}{\partial y_i^{(N)}}$$

$$\frac{\partial Div}{\partial y_i^{(k-1)}} = \sum_j w_{ij}^{(k)} \frac{\partial Div}{\partial z_j^{(k)}}$$

$$\frac{\partial Div}{\partial w_{ij}^{(k)}} = \frac{\partial z_j^{(k)}}{\partial w_{ij}^{(k)}}\frac{\partial Div}{\partial z_j^{(k)}} = y_i^{(k-1)}\frac{\partial Div}{\partial z_j^{(k)}}$$

Gradients: Backward Computation

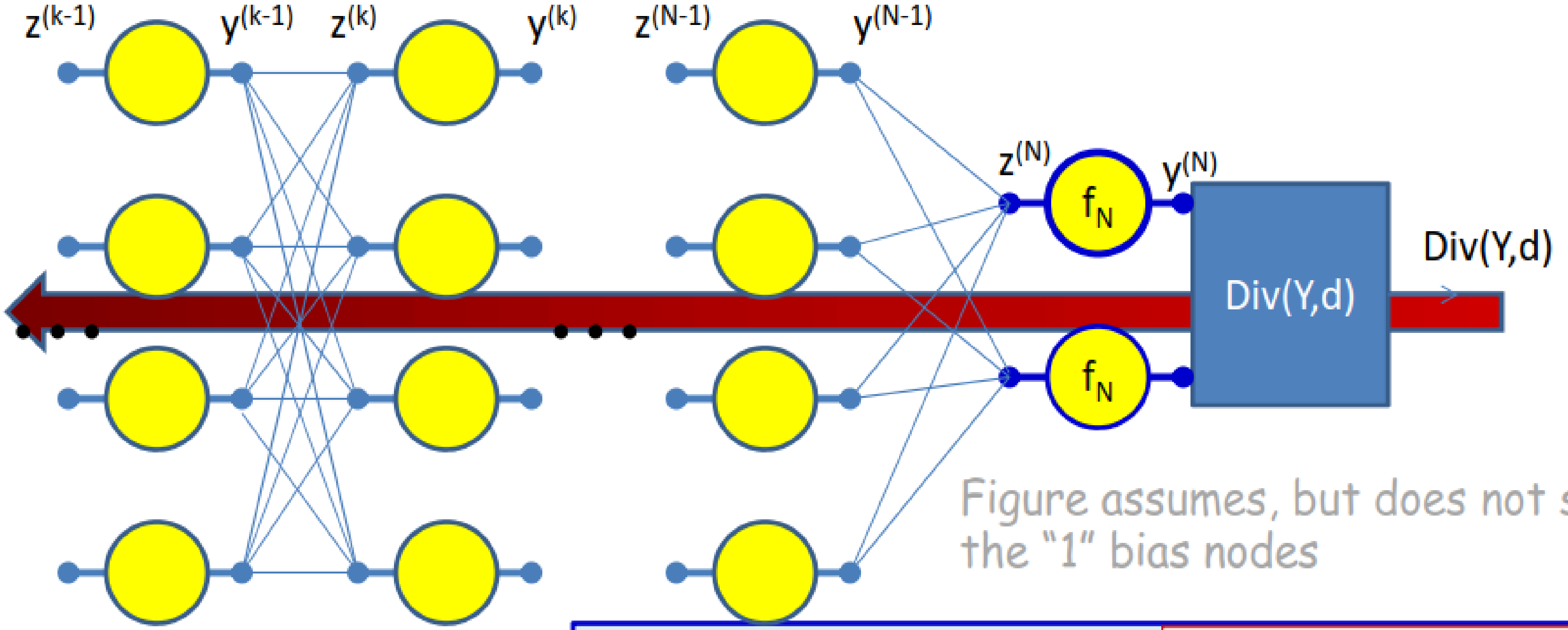


Figure assumes, but does not show the "1" bias nodes

Initialize: Gradient w.r.t network output

$$\frac{\partial Div}{\partial y_i} = \frac{\partial Div(Y,d)}{\partial y_i^{(N)}}$$

$For\ k = N..1$

$For\ i = 1{:}\ layer - width$

$$\frac{\partial Div}{\partial z_i^{(k)}} = f_k'\left(z_i^{(k)}\right)\frac{\partial Div}{\partial y_i^{(k)}}$$

$$\frac{\partial Div}{\partial y_i^{(k-1)}} = \sum_j w_{ij}^{(k)} \frac{\partial Div}{\partial z_j^{(k)}}$$

$$\frac{\partial Div}{\partial w_{ij}^{(k)}} = y_i^{(k-1)} \frac{\partial Div}{\partial z_j^{(k)}}$$

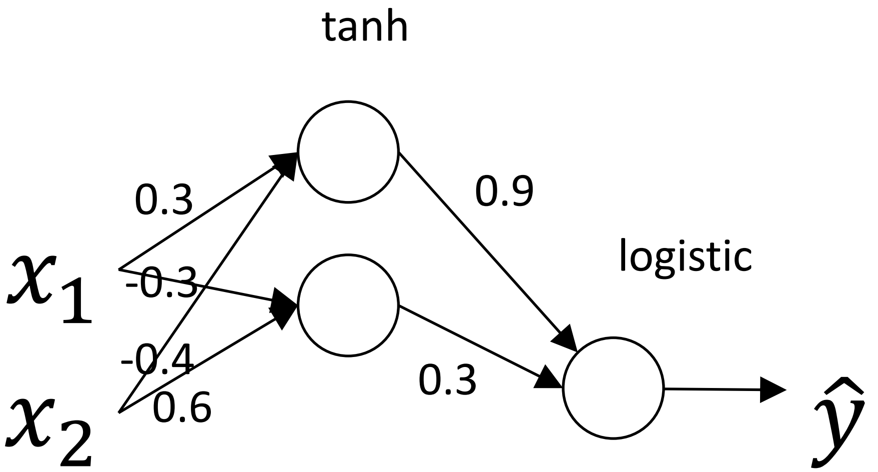
Training by BackProp

- Initialize all weights $(\boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}, \dots, \boldsymbol{W}^{(K)})$
- Do:
  - **Initialize** $Err = 0$; For all i, j, k, initialize $\frac{dErr}{dw_{i,j}^{(k)}} = 0$
  - For all $t = 1{:}T$ (Loop over training instances)
    - **Forward pass:** Compute
      - Output $\boldsymbol{Y_t}$
      - $Err \mathrel{+}= \boldsymbol{Div}(\boldsymbol{Y_t}, \boldsymbol{d_t})$
    - **Backward pass:** For all i, j, k:
      - Compute $\frac{d\boldsymbol{Div}(\boldsymbol{Y_t}, \boldsymbol{d_t})}{dw_{i,j}^{(k)}}$
      - Compute $\frac{dErr}{dw_{i,j}^{(k)}} \mathrel{+}= \frac{d\boldsymbol{Div}(\boldsymbol{Y_t}, \boldsymbol{d_t})}{dw_{i,j}^{(k)}}$
  - For all i, j, k, update:

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

- Until Err has converged

Exercise



1.1

0.4

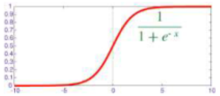
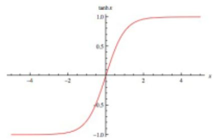
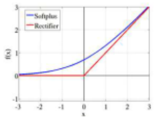
Expected output: 1

$$Div = \frac{(\hat{y}_i - y_i)^2}{2}$$

$$(\hat{y}_i - y_i)$$

?

Activations and their derivatives

	$f(z) = \frac{1}{1+\exp(-z)}$	$f'(z) = f(z)(1 - f(z))$
	$f(z) = \tanh(z)$	$f'(z) = (1 - f^2(z))$
	$f(z) = \begin{cases} 0, & z < 0 \\ z, & z \geq 0 \end{cases}$	This space left intentionally (kind of) blank
	$f(z) = \log(1 + \exp(z))$	$f'(z) = \frac{1}{1+\exp(-z)}$

Gradients: Backward Computation

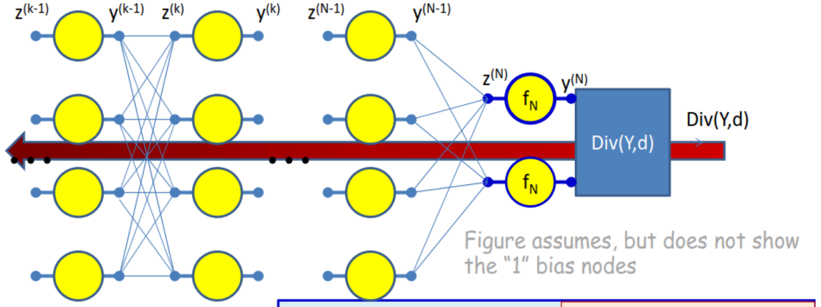


Figure assumes, but does not show the "1" bias nodes

Initialize: Gradient w.r.t network output

$$\frac{\partial Div}{\partial y_i} = \frac{\partial Div(Y,d)}{\partial y_i^{(N)}}$$

$For\ k = N..1$

$For\ i = 1: layer - width$

$$\frac{\partial Div}{\partial z_i^{(k)}} = f_k'\left(z_i^{(k)}\right)\frac{\partial Div}{\partial y_i^{(k)}}$$

$$\frac{\partial Div}{\partial y_i^{(k-1)}} = \sum_j w_{ij}^{(k)} \frac{\partial Div}{\partial z_j^{(k)}}$$

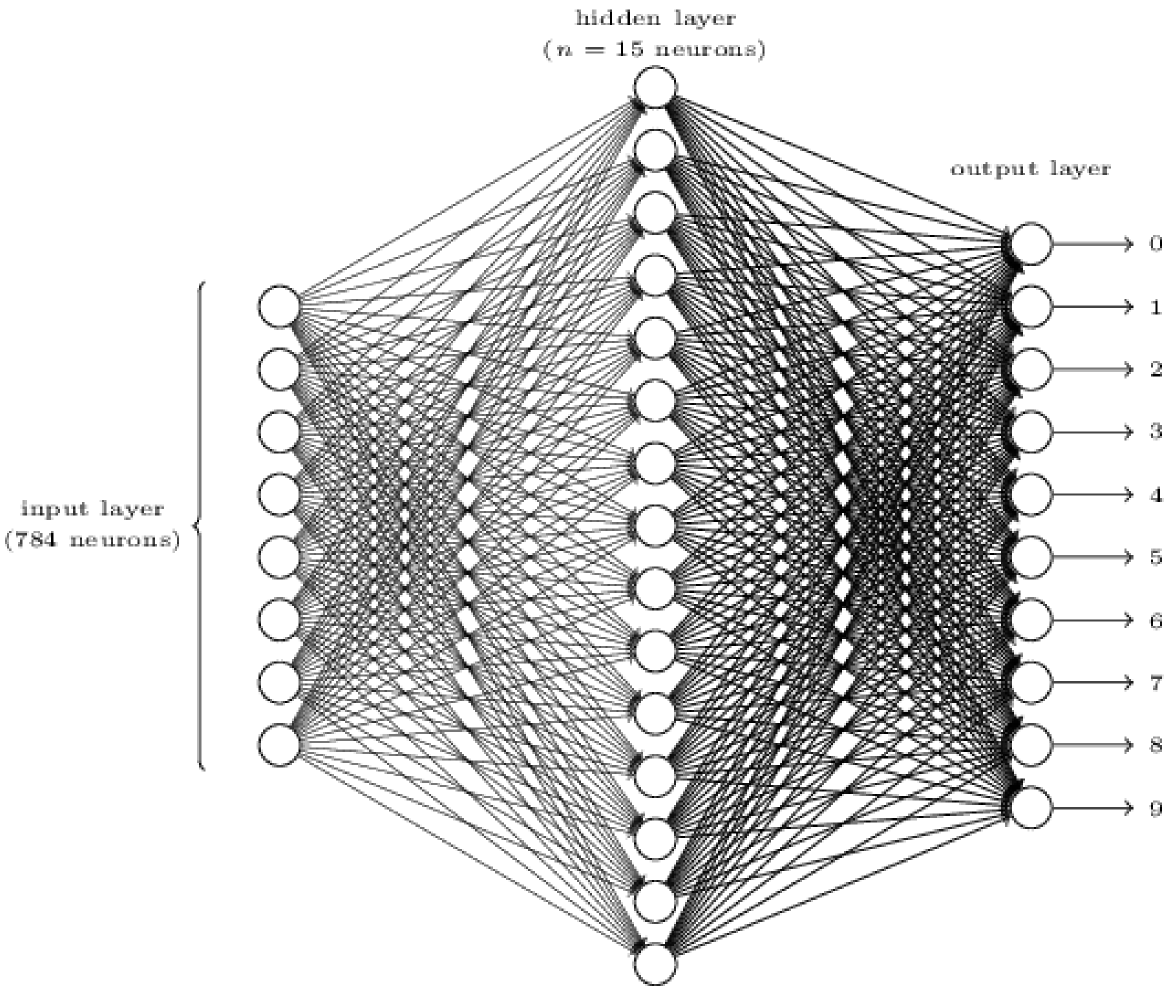
$$\boxed{\frac{\partial Div}{\partial w_{ij}^{(k)}} = y_i^{(k-1)} \frac{\partial Div}{\partial z_j^{(k)}}}$$

A real example

Digit classification

28

28 



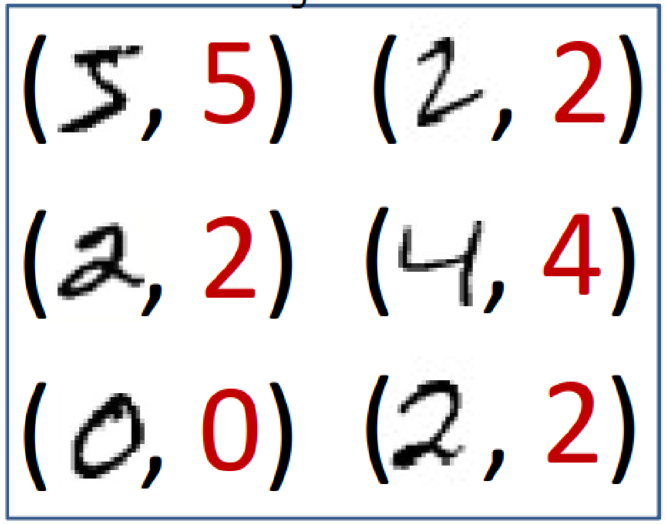
- MNIST dataset:
  - 70000 grayscale images of digits scanned.
  - 60000 for training
  - 10000 for testing
- Loss function

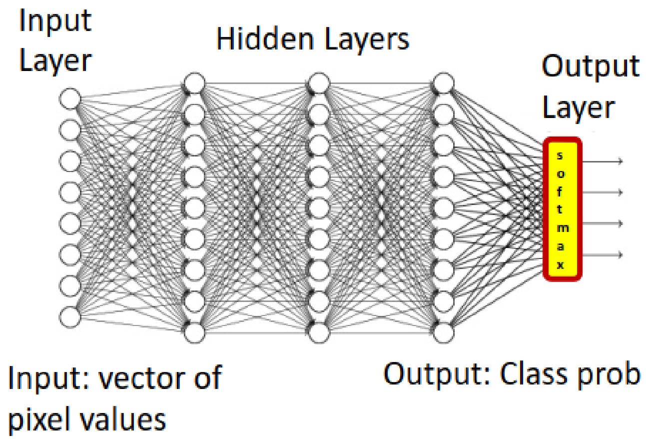
$$J_2(w) = \frac{1}{m} \sum_{train} (\,\hat{y}_i - y_i)^2$$

Digit classification

Typical Problem statement: multiclass classification

Training data

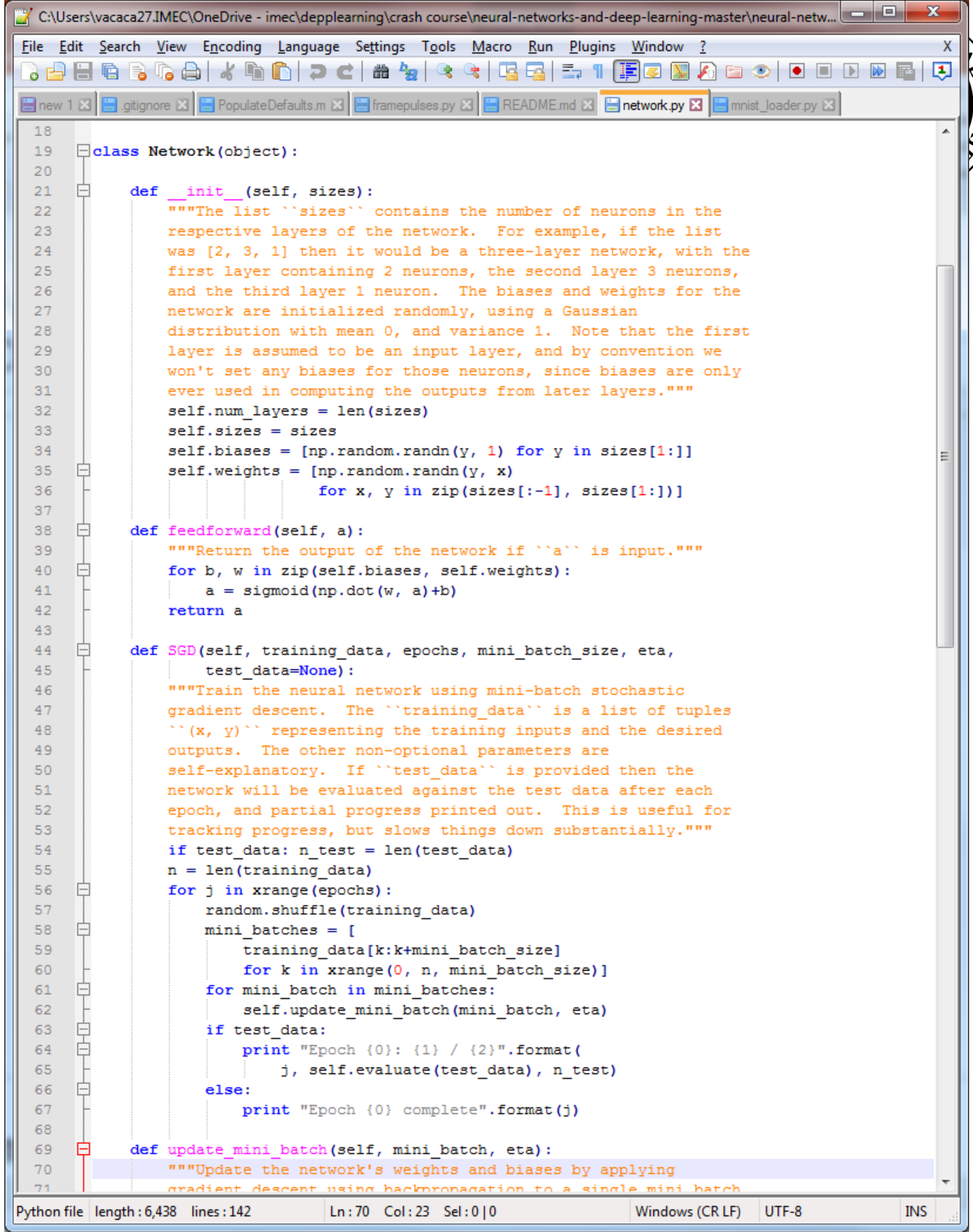




- Given, many positive and negative examples (training data),
  - learn all weights such that the network does the desired job

A look in the code

- To run this code do:
  - import network
  - net = network.Network([784, 30, 10])
  - net.SGD(training_data, 30, 10, 3.0, test_data=test_data)




```python

class Network(object):

    def __init__(self, sizes):
        """The list ``sizes`` contains the number of neurons in the
        respective layers of the network.  For example, if the list
        was [2, 3, 1] then it would be a three-layer network, with the
        first layer containing 2 neurons, the second layer 3 neurons,
        and the third layer 1 neuron.  The biases and weights for the
        network are initialized randomly, using a Gaussian
        distribution with mean 0, and variance 1.  Note that the first
        layer is assumed to be an input layer, and by convention we
        won't set any biases for those neurons, since biases are only
        ever used in computing the outputs from later layers."""
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]

    def feedforward(self, a):
        """Return the output of the network if ``a`` is input."""
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a

    def SGD(self, training_data, epochs, mini_batch_size, eta,
            test_data=None):
        """Train the neural network using mini-batch stochastic
        gradient descent.  The ``training_data`` is a list of tuples
        ``(x, y)`` representing the training inputs and the desired
        outputs.  The other non-optional parameters are
        self-explanatory.  If ``test_data`` is provided then the
        network will be evaluated against the test data after each
        epoch, and partial progress printed out.  This is useful for
        tracking progress, but slows things down substantially."""
        if test_data: n_test = len(test_data)
        n = len(training_data)
        for j in xrange(epochs):
            random.shuffle(training_data)
            mini_batches = [
                training_data[k:k+mini_batch_size]
                for k in xrange(0, n, mini_batch_size)]
            for mini_batch in mini_batches:
                self.update_mini_batch(mini_batch, eta)
            if test_data:
                print "Epoch {0}: {1} / {2}".format(
                    j, self.evaluate(test_data), n_test)
            else:
                print "Epoch {0} complete".format(j)

    def update_mini_batch(self, mini_batch, eta):
        """Update the network's weights and biases by applying
        gradient descent using backpropagation to a single mini batch.
```

A look in code

Initialize: Gradient w.r.t network output

$$\frac{\partial Div}{\partial y_i} = \frac{\partial Div(Y,d)}{\partial y_i^{(N)}}$$

For $k = N..1$

For $i = 1: layer - width$

$$\frac{\partial Div}{\partial z_i^{(k)}} = f_k'\left(z_i^{(k)}\right)\frac{\partial Div}{\partial y_i^{(k)}}$$

$$\frac{\partial Div}{\partial y_i^{(k-1)}} = \sum_j w_{ij}^{(k)} \frac{\partial Div}{\partial z_j^{(k)}}$$

$$\boxed{\frac{\partial Div}{\partial w_{ij}^{(k)}} = y_i^{(k-1)} \frac{\partial Div}{\partial z_j^{(k)}}}$$

*C:\Users\vacaca27.IMEC\OneDrive - imec\depplearning\crash course\neural-networks-and-deep-learning-master\neural-net...

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

new 1 | .gitignore | PopulateDefaults.m | framepulses.py | README.md | network.py | mnist_loader.py

```python

    def backprop(self, x, y):
        """Return a tuple ``(nabla_b, nabla_w)`` representing the
        gradient for the cost function C_x.  ``nabla_b`` and
        ``nabla_w`` are layer-by-layer lists of numpy arrays, similar
        to ``self.biases`` and ``self.weights``."""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        # feedforward
        activation = x
        activations = [x] # list to store all the activations, layer by layer
        zs = [] # list to store all the z vectors, layer by layer
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b
            zs.append(z)
            activation = sigmoid(z)
            activations.append(activation)
        # backward pass
        delta = self.cost_derivative(activations[-1], y) * \
            sigmoid_prime(zs[-1])
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())
        # Note that the variable l in the loop below is used a little
        # differently to the notation in Chapter 2 of the book.  Here,
        # l = 1 means the last layer of neurons, l = 2 is the
        # second-last layer, and so on.  It's a renumbering of the
        # scheme in the book, used here to take advantage of the fact
        # that Python can use negative indices in lists.
        for l in xrange(2, self.num_layers):
            z = zs[-l]
            sp = sigmoid_prime(z)
            delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
        return (nabla_b, nabla_w)

    def cost_derivative(self, output_activations, y):
        """Return the vector of partial derivatives \partial C_x /
        \partial a for the output activations."""
        return (output_activations-y)

#### Miscellaneous functions
def sigmoid(z):
    """The sigmoid function."""
    return 1.0/(1.0+np.exp(-z))

def sigmoid_prime(z):
    """Derivative of the sigmoid function."""
    return sigmoid(z)*(1-sigmoid(z))

```

Python file | length : 6,439 | lines : 142 | Ln : 140 | Col : 5 | Sel : 0 | 0 | Windows (CR LF) | UTF-8 | INS

A look in code

Initialize: Gradient w.r.t network output

$$\frac{\partial Div}{\partial y_i} = \frac{\partial Div(Y,d)}{\partial y_i^{(N)}}$$

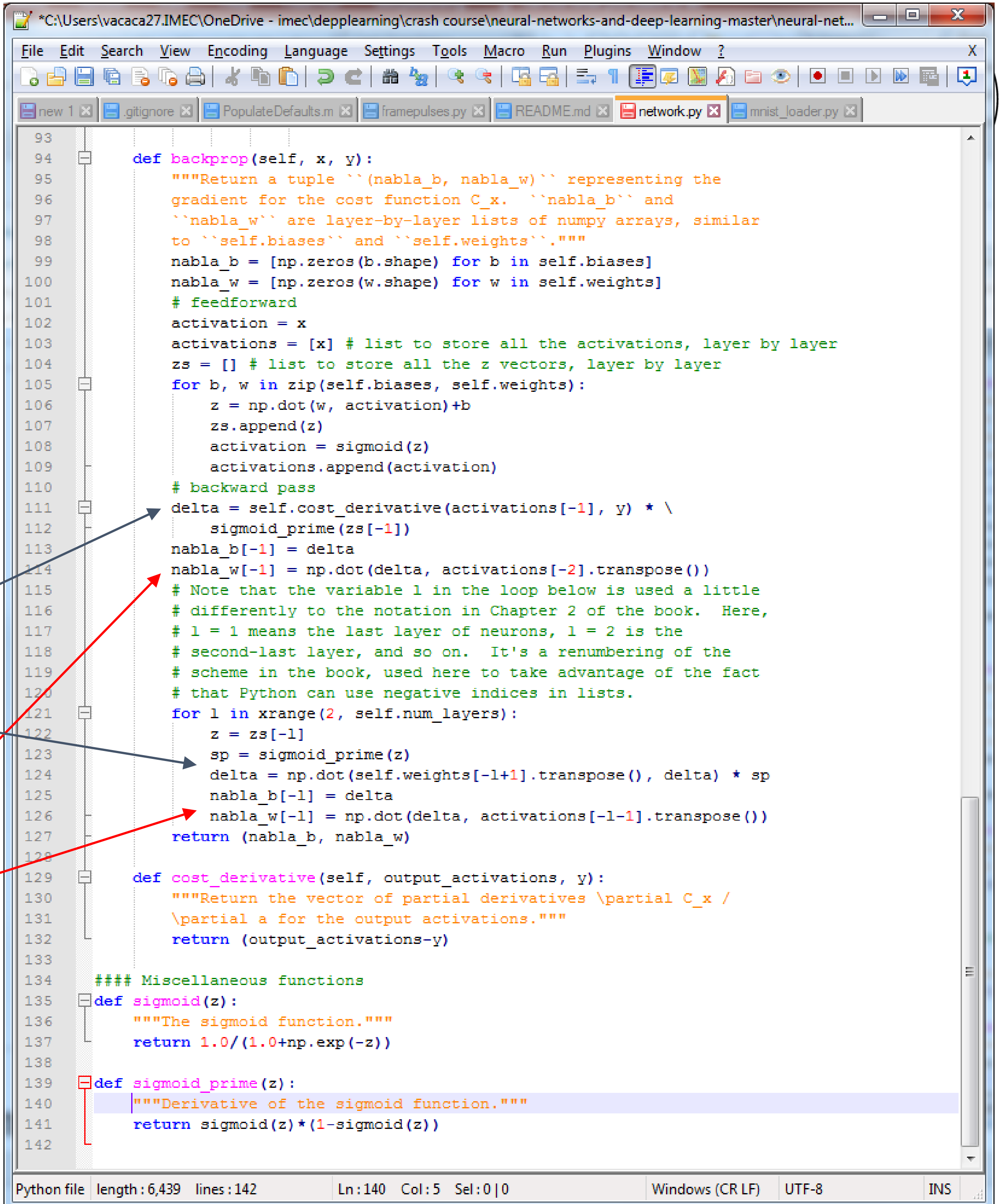
For $k = N..1$

For $i = 1 : layer - width$

$$\frac{\partial Div}{\partial z_i^{(k)}} = f_k'\left(z_i^{(k)}\right)\frac{\partial Div}{\partial y_i^{(k)}}$$

$$\frac{\partial Div}{\partial y_i^{(k-1)}} = \sum_j w_{ij}^{(k)} \frac{\partial Div}{\partial z_j^{(k)}}$$

$$\boxed{\frac{\partial Div}{\partial w_{ij}^{(k)}} = y_i^{(k-1)} \frac{\partial Div}{\partial z_j^{(k)}}}$$



A look in the code

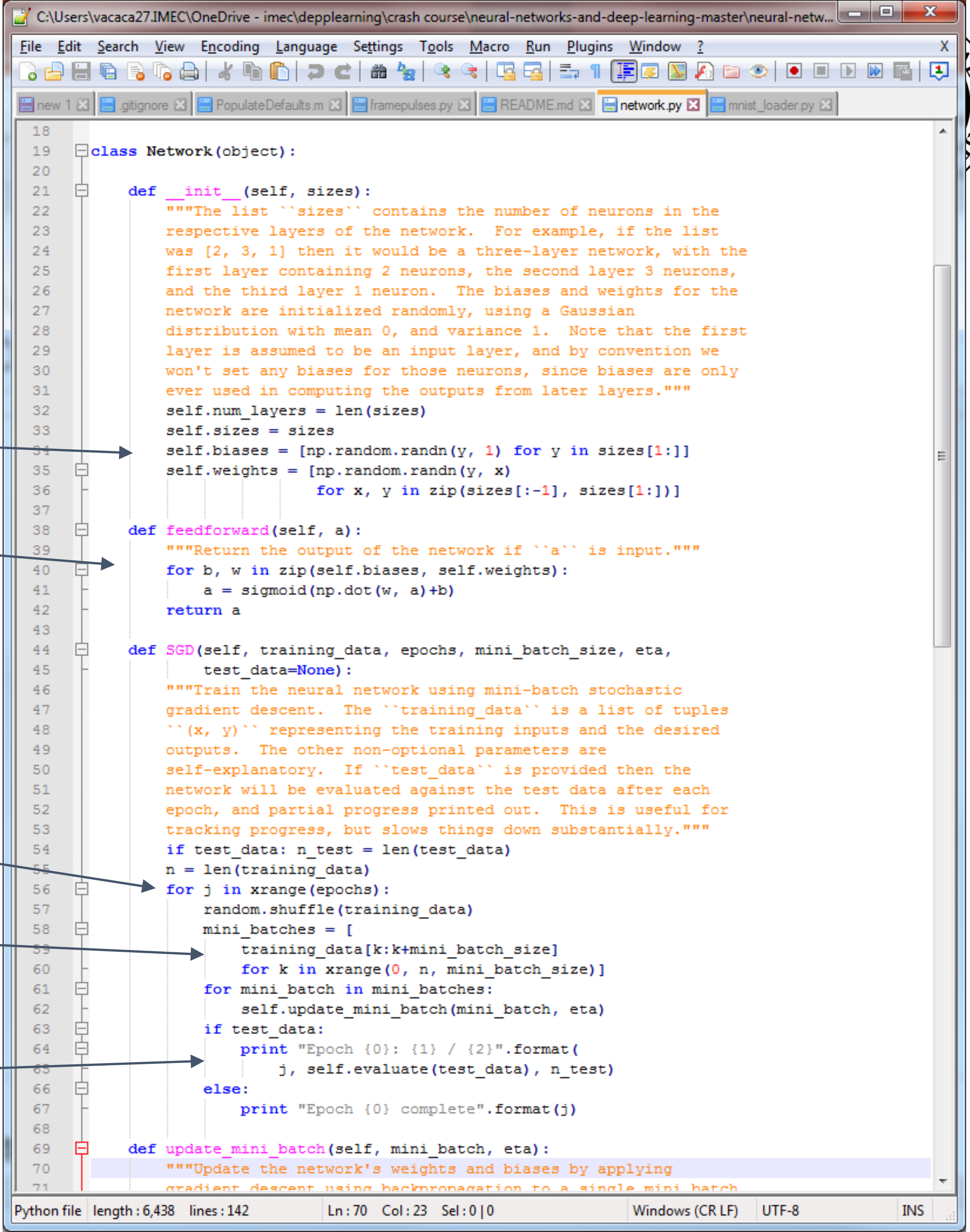
random Initialization

Feed forward ‘a’ thru all the layers

A Epoch is when all the training data has been used to update weights

A minibatch is a subset of all the data used to obtain a ‘quick’ weight updates

If there is test data perform evaluation



A look in the code

Add errors from all the training data from the mini-batch

Update the weights

```python
    def update_mini_batch(self, mini_batch, eta):
        """Update the network's weights and biases by applying
        gradient descent using backpropagation to a single mini batch.
        The ``mini_batch`` is a list of tuples ``(x, y)``, and ``eta``
        is the learning rate."""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        for x, y in mini_batch:
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)
            nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
        self.weights = [w-(eta/len(mini_batch))*nw
                        for w, nw in zip(self.weights, nabla_w)]
        self.biases = [b-(eta/len(mini_batch))*nb
                       for b, nb in zip(self.biases, nabla_b)]

    def evaluate(self, test_data):
        """Return the number of test inputs for which the neural
        network outputs the correct result. Note that the neural
        network's output is assumed to be the index of whichever
        neuron in the final layer has the highest activation."""
        test_results = [(np.argmax(self.feedforward(x)), y)
                        for (x, y) in test_data]
        return sum(int(x == y) for (x, y) in test_results)
```

references

- http://neuralnetworksanddeeplearning.com/chap1.html
- https://www.cs.cmu.edu/~bhiksha/courses/deeplearning/Fall.2015/

Questions?