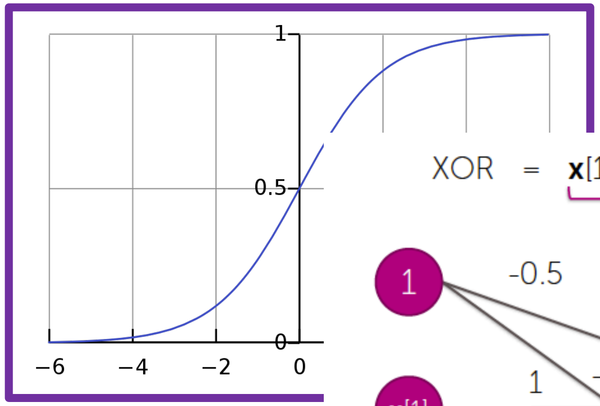
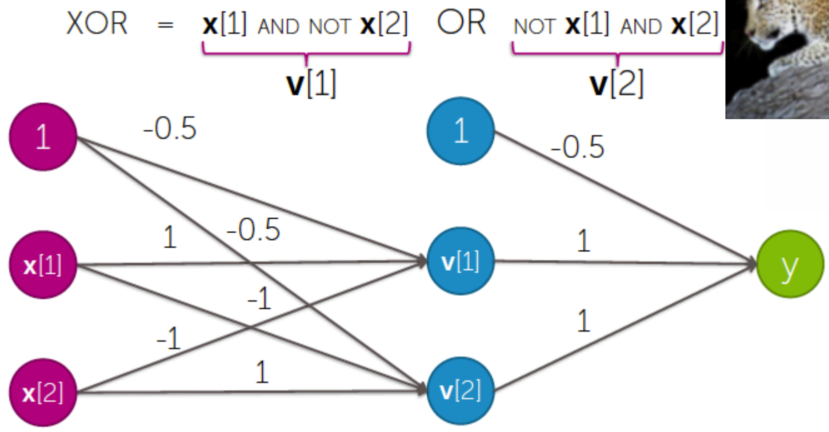
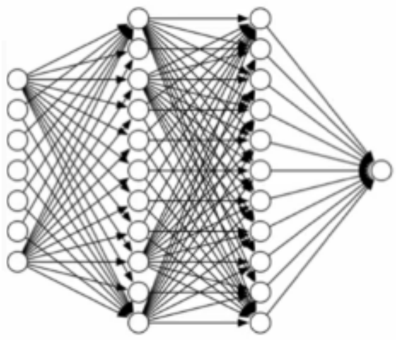
Tufts COMP 135: Introduction to Machine Learning

https://www.cs.tufts.edu/comp/135/2020f/

Backpropagation







Many slides attributable to:
Erik Sudderth (UCI), Emily Fox (UW),
Finale Doshi-Velez (Harvard)
James, Witten, Hastie, Tibshirani (ISL/ESL books)

Prof. Mike Hughes

Objectives Today (day 11)
Backpropagation

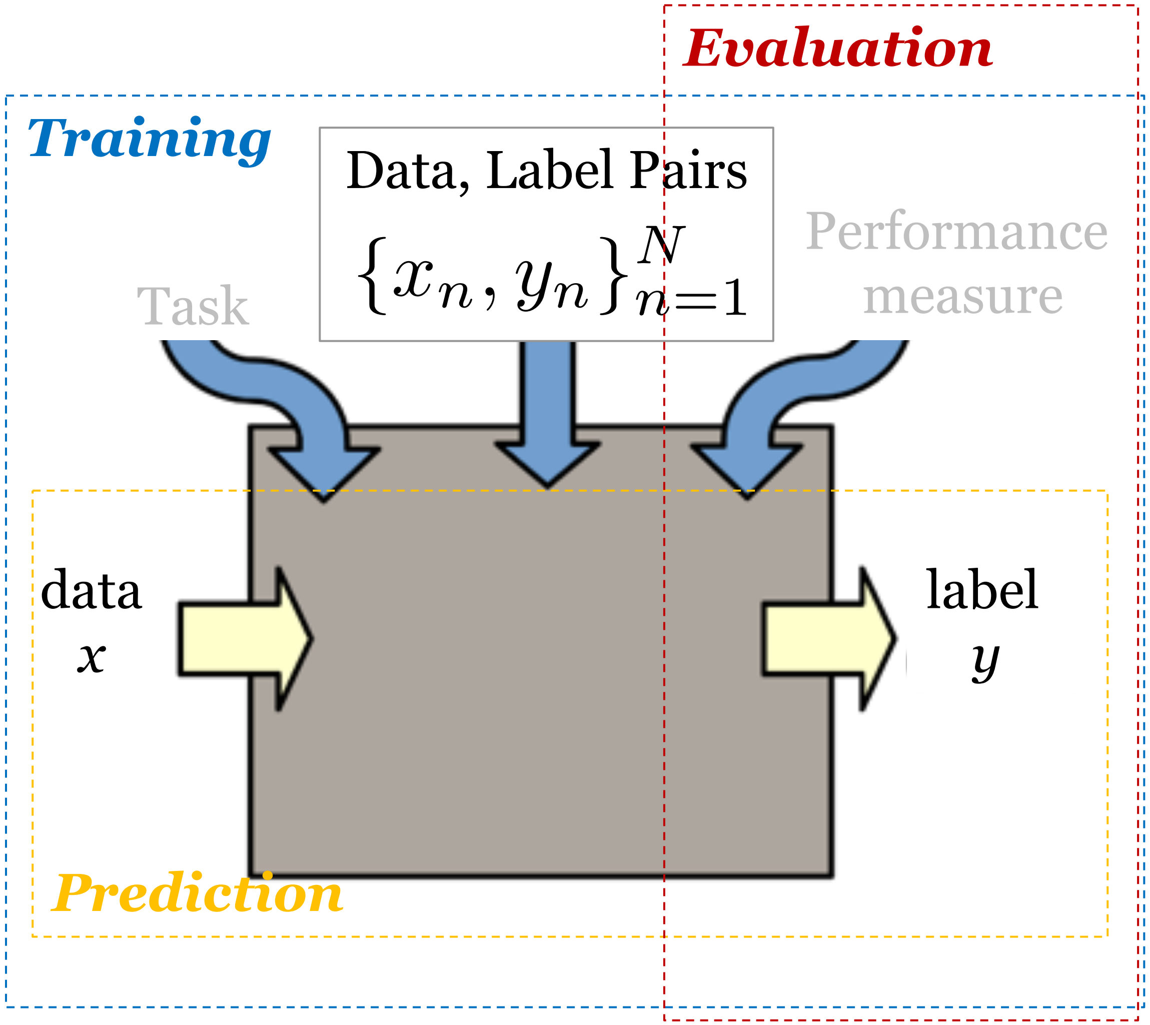
- Review: **Multi-layer perceptrons**
  - MLPs can learn feature representations
  - Activation functions
- Training via gradient descent
  - Back-propagation = gradient descent + chain rule

What will we learn?

Supervised Learning

Unsupervised Learning

Reinforcement Learning



Task: Binary Classification

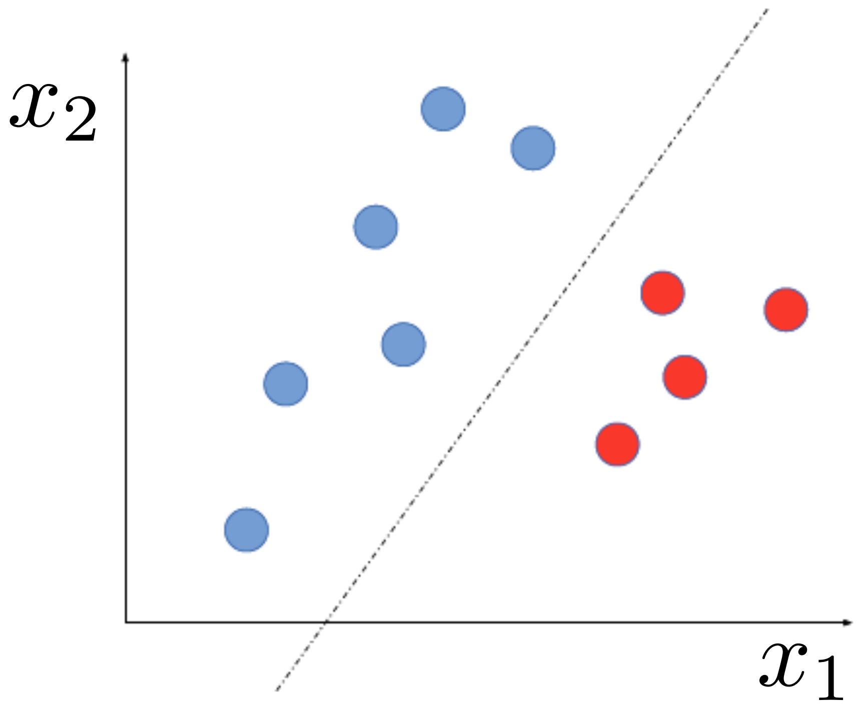
Supervised Learning

binary classification

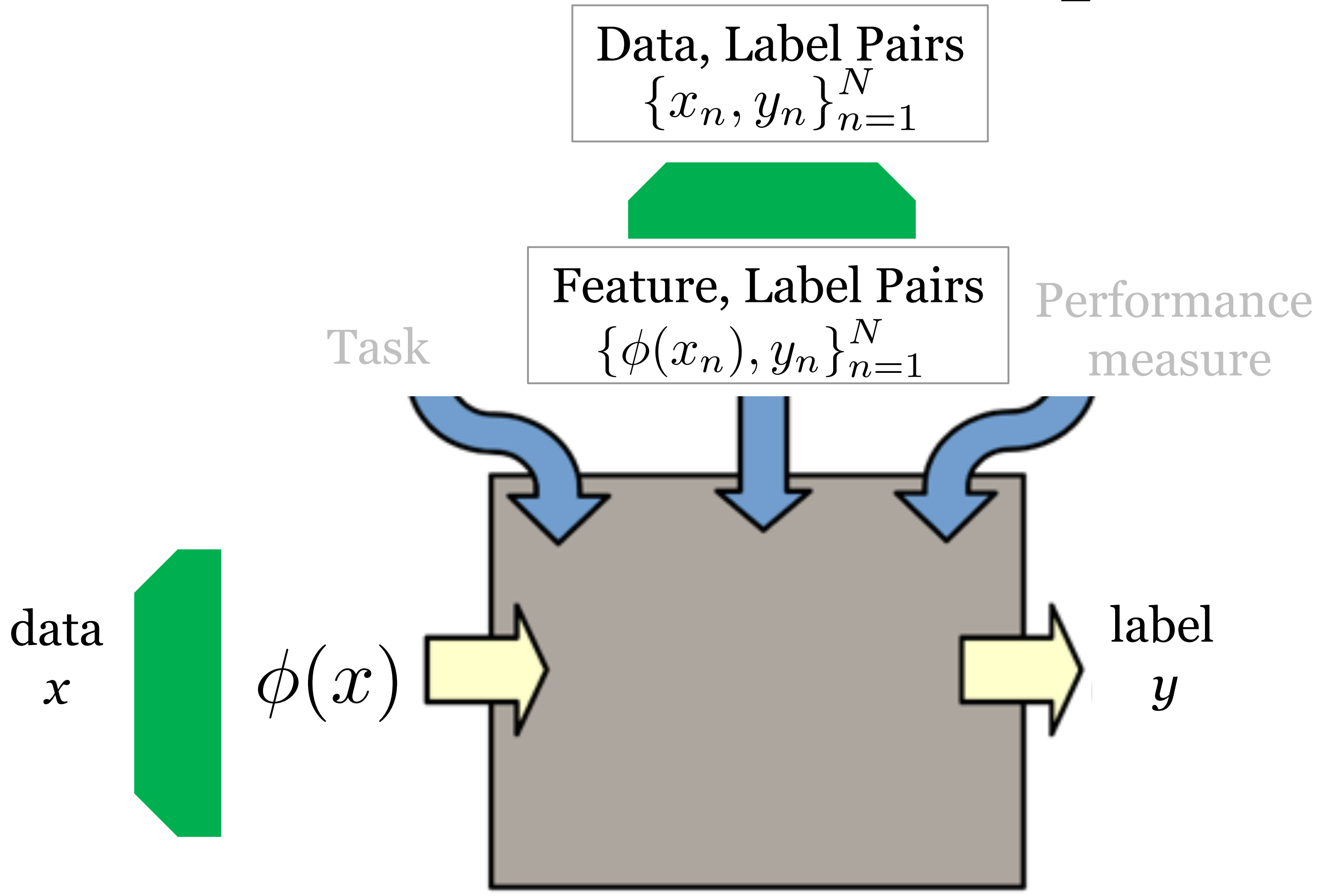
Unsupervised Learning

Reinforcement Learning

y is a binary variable (red or blue)



Feature Transform Pipeline



MLP: Multi-Layer Perceptron

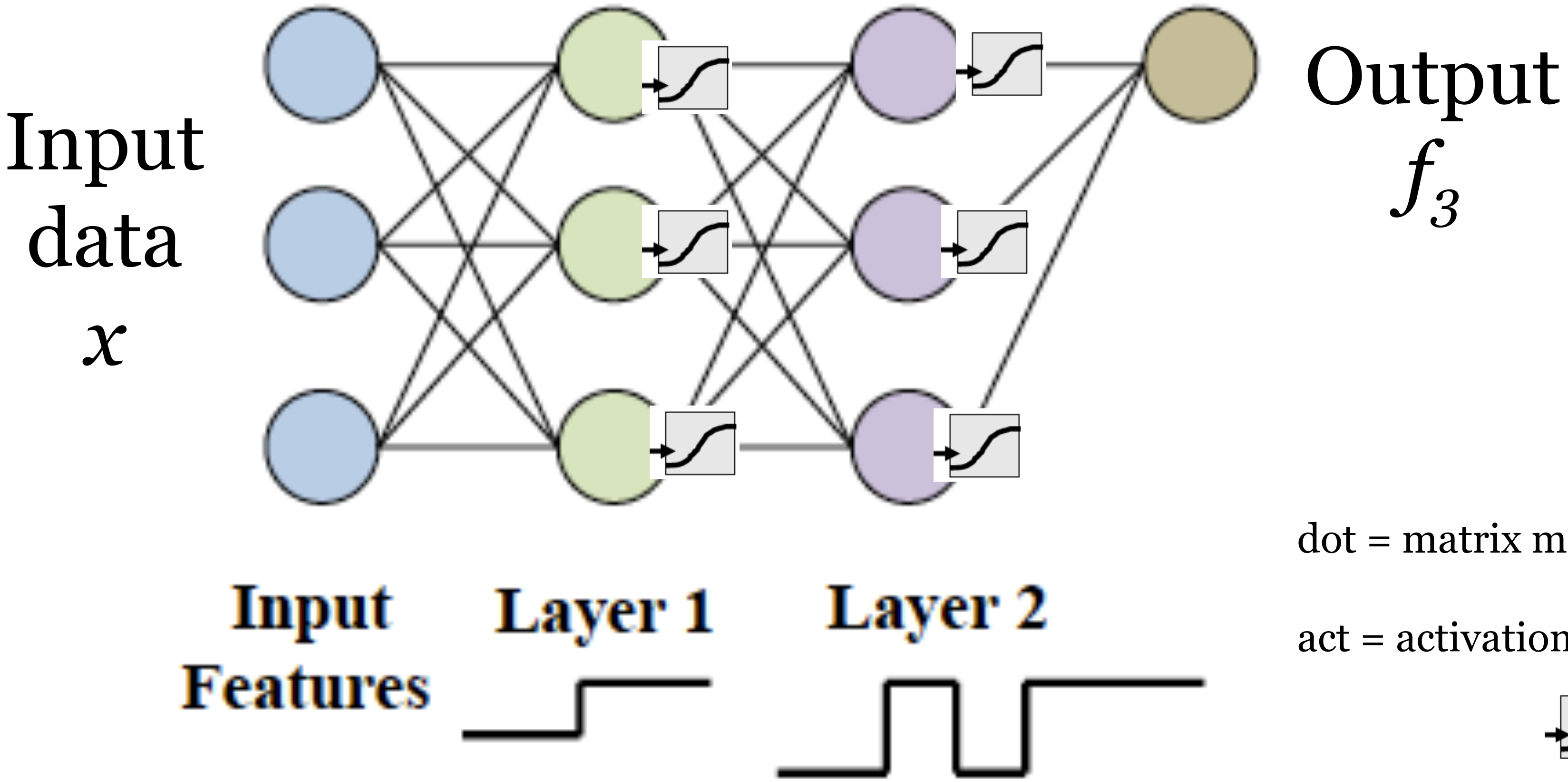
Neural network with
1 or more hidden layers
followed by 1 output layer

Neural nets with **many** layers

$$f_1 = \mathrm{dot}(w_1, x) + b_1$$

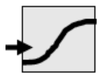
$$f_2 = \mathrm{dot}(w_2, \mathrm{act}(f_1)) + b_2$$

$$f_3 = \mathrm{dot}(w_3, \mathrm{act}(f_2)) + b_3$$

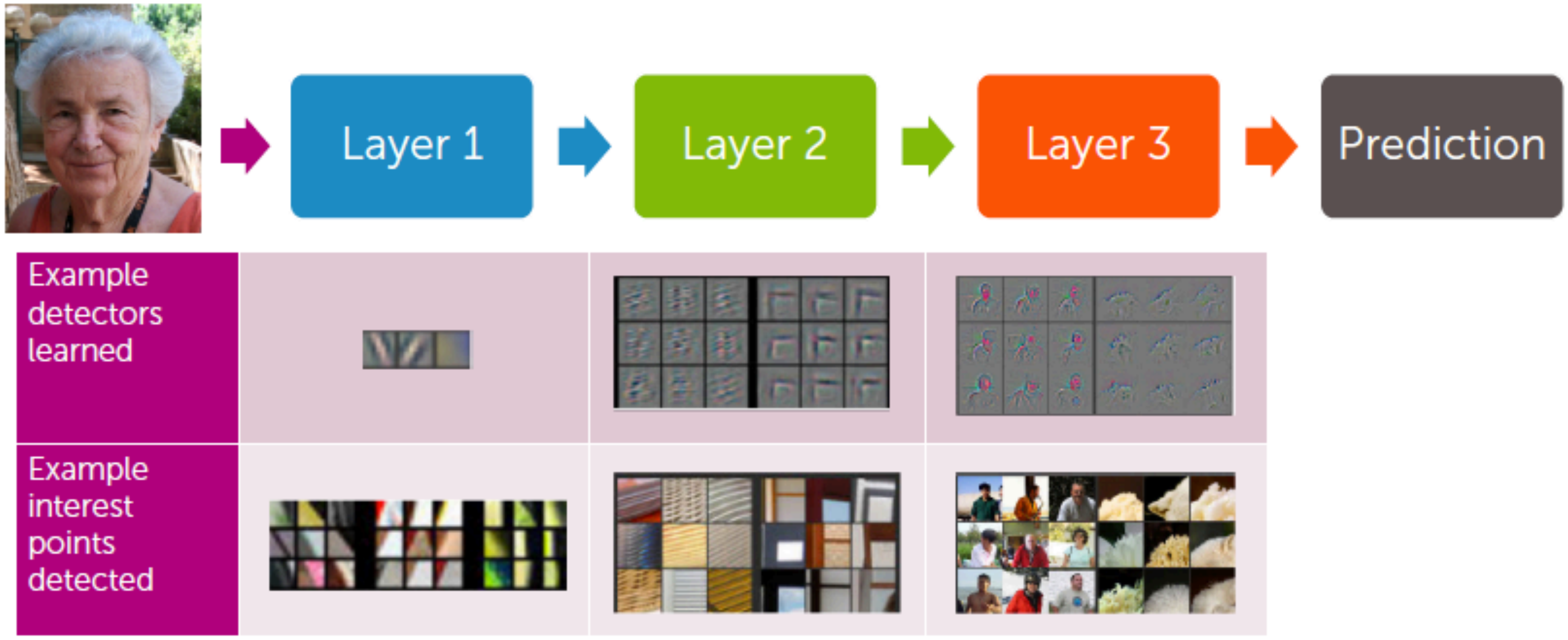


dot = matrix multiply

act = activation function



Each Layer Extracts “Higher Level” Features



Multi-layer perceptron (MLP)

You can define an MLP by specifying:

- Number of hidden layers (L-1) and size of each layer
  - `hidden_layer_sizes = [A, B, C, D, …]`
- Hidden layer activation function
  - ReLU, etc.
- Output layer activation function

$$f_1 = \mathrm{dot}(w_1, x) + b_1 \qquad f_1 \in \mathbb{R}^A$$

$$f_2 = \mathrm{dot}(w_2, \mathrm{act}(f_1)) + b_2 \qquad f_2 \in \mathbb{R}^B$$

$$f_3 = \mathrm{dot}(w_3, \mathrm{act}(f_2)) + b_3 \qquad f_3 \in \mathbb{R}^C$$

$$f_L = \mathrm{dot}(w_L, \mathrm{act}(f_{L-1})) + b_L \qquad f_L \in \mathbb{R}^1$$

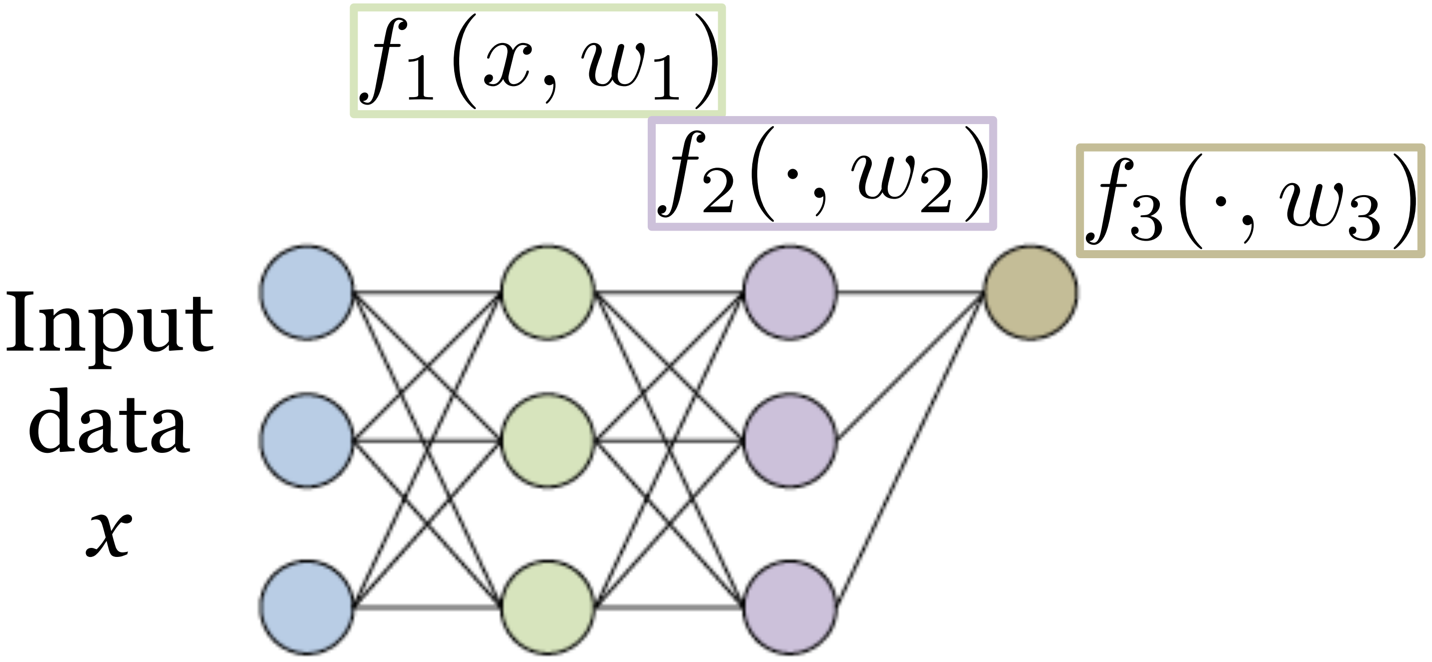
How to train Neural Nets?

Just like logistic regression

1. Set up a loss function
2. Apply gradient descent!

Output as function of weights

$$f_3(f_2(f_1(x, w_1), w_2), w_3)$$



Minimizing loss for multi-layer functions

$$\min_{w_1, w_2, w_3} \sum_{n=1}^{N} \mathrm{loss}(y_n, f_3(f_2(f_1(x_n, w_1), w_2), w_3))$$

Loss can be:

- Squared error for regression problems
- Log loss for binary classification problems
- … many others possible!

Can try to find best possible weights with gradient descent…. But **how do we compute gradients**?

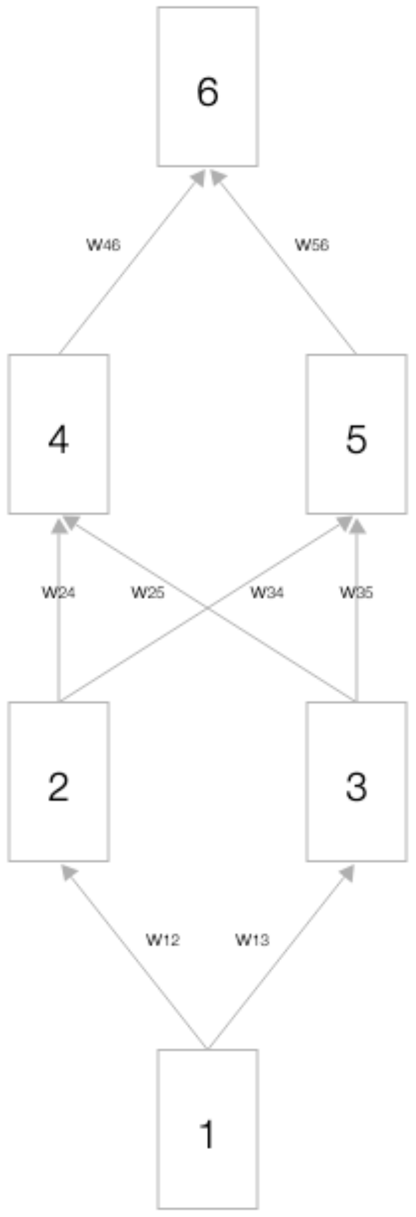
Big idea: NN as a Computation Graph

Each node represents **one scalar result** produced by our NN

- Node 1: Input x
- Node 2 and 3: Hidden layer 1
- Node 4 and 5: Hidden layer 2
- Node 6: Output

Each edge represents **one scalar weight** that is a parameter of our NN

To keep this simple, we omit bias parameters.



$$E = \frac{1}{2}(y_{output} - y_{target})^2$$

Notation

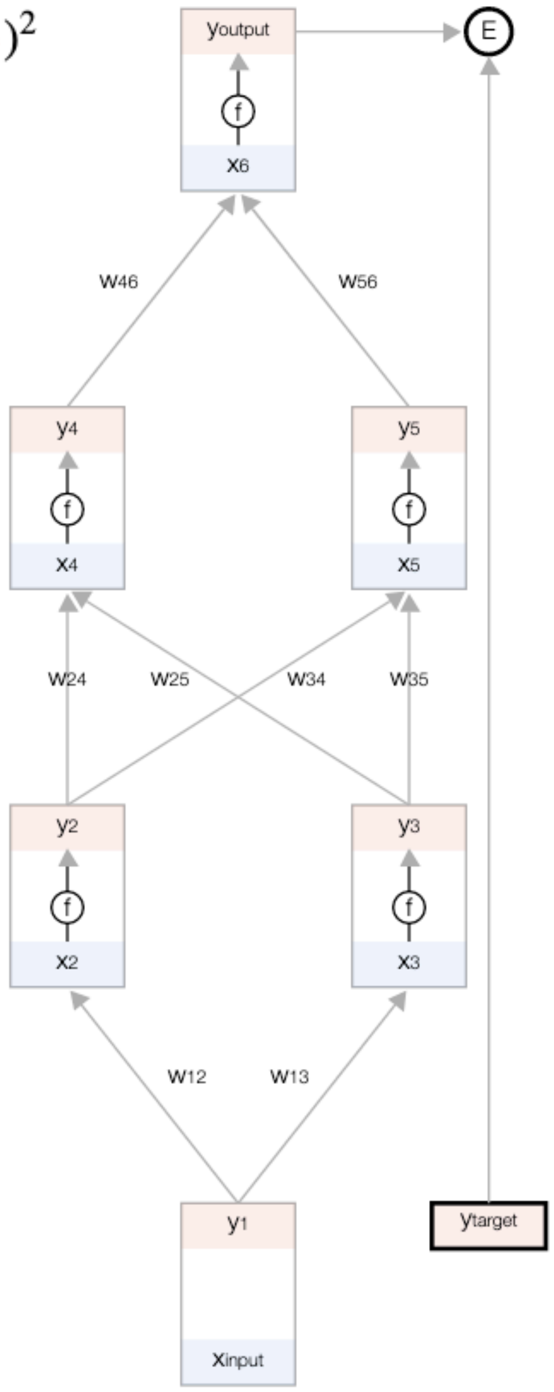
Each node *i* in our graph has:

- Input scalar x_i
- Activation function *f*
  - Input node: identity $f(x) = x$
  - Hidden node: f = ReLU
  - Output node:
    - f = identity for regression
    - f = sigmoid for classification
- Output scalar y_i

Key idea: Nodes are ***in order***

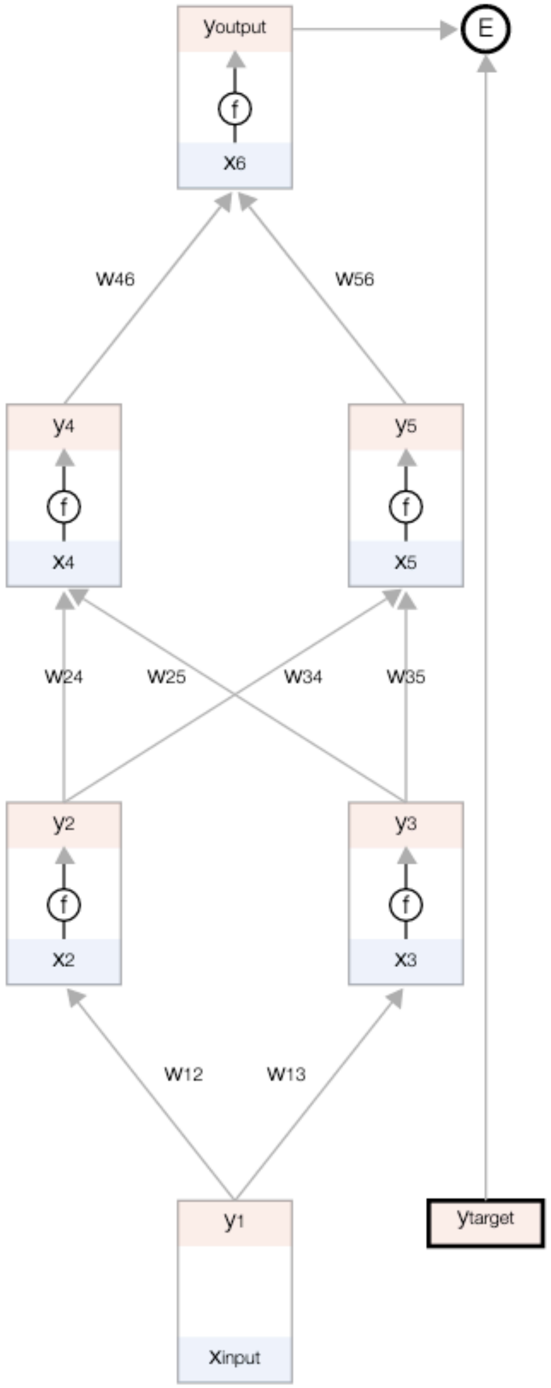
- Input of node *i* can be computed from outputs of nodes 1, 2, …. *i* -1

Extra terminal node ("E") for the result of the loss function



Two directions of propagation

Forward: compute loss
Backward: compute grad



Forward Propagation Algorithm

1. For each non-input, non-terminal node j ***in order:***

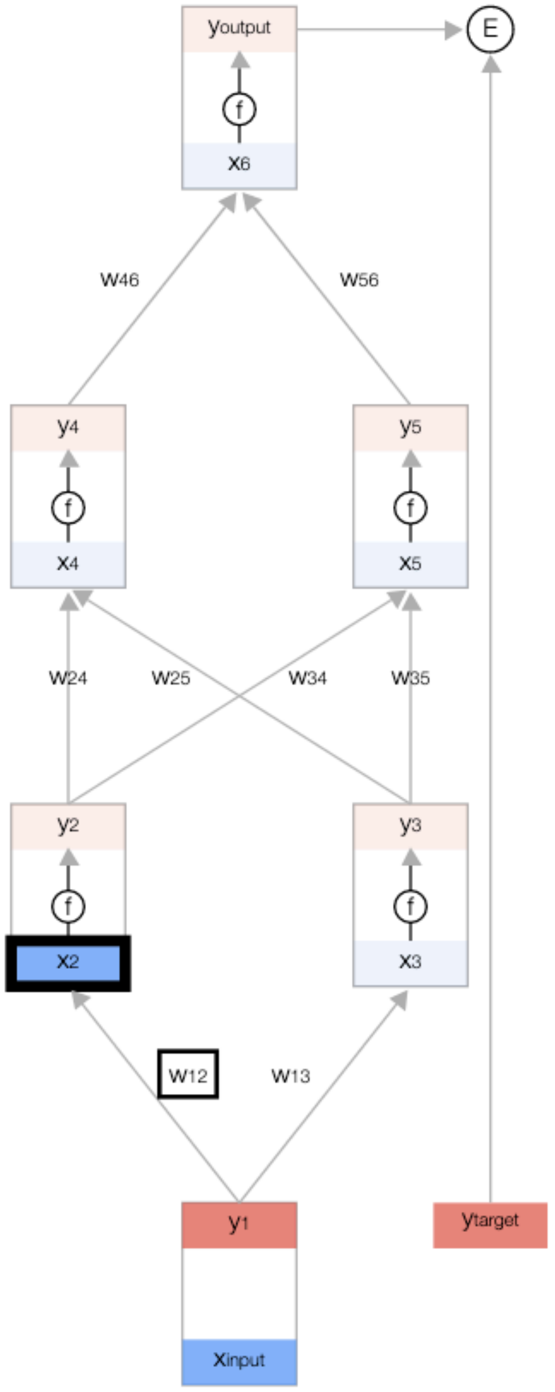
    Compute and store input value for node j

$$x_j = \sum_{i \in in(j)} w_{ij} y_i$$

    Compute and store output value for node j

$$y_j = f(x_j)$$

2. Compute terminal node value



$$E = \frac{1}{2}(y_{output} - y_{target})^2$$

Notation for Back Propagation

Remember, our ultimate goal is to compute gradient of our loss E with respect to parameters w

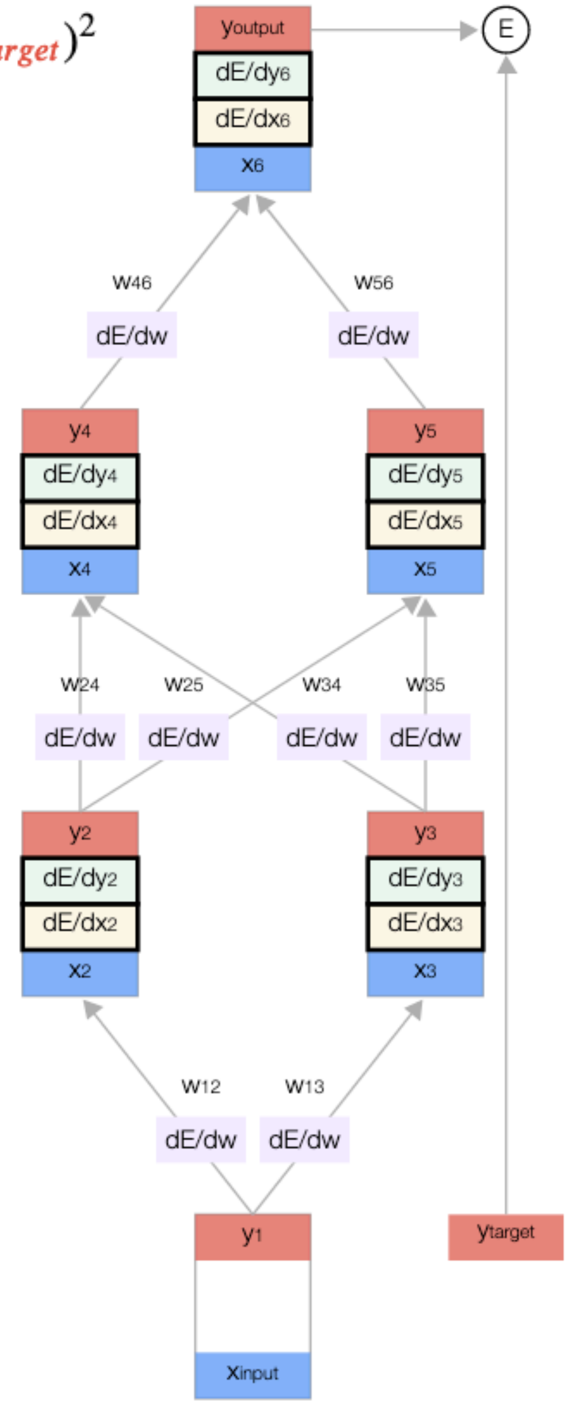
Before we begin "backward pass"…

We need to store at each node i:

- Derivative wrt its output $\frac{dE}{dy}$
- Derivative wrt its input $\frac{dE}{dx}$

We need to store at each edge i,j:

- Derivative wrt its weight $\frac{dE}{dw_{ij}}$



$$E = \frac{1}{2}(y_{output} - y_{target})^2$$

Back Propagation Algorithm Step 1

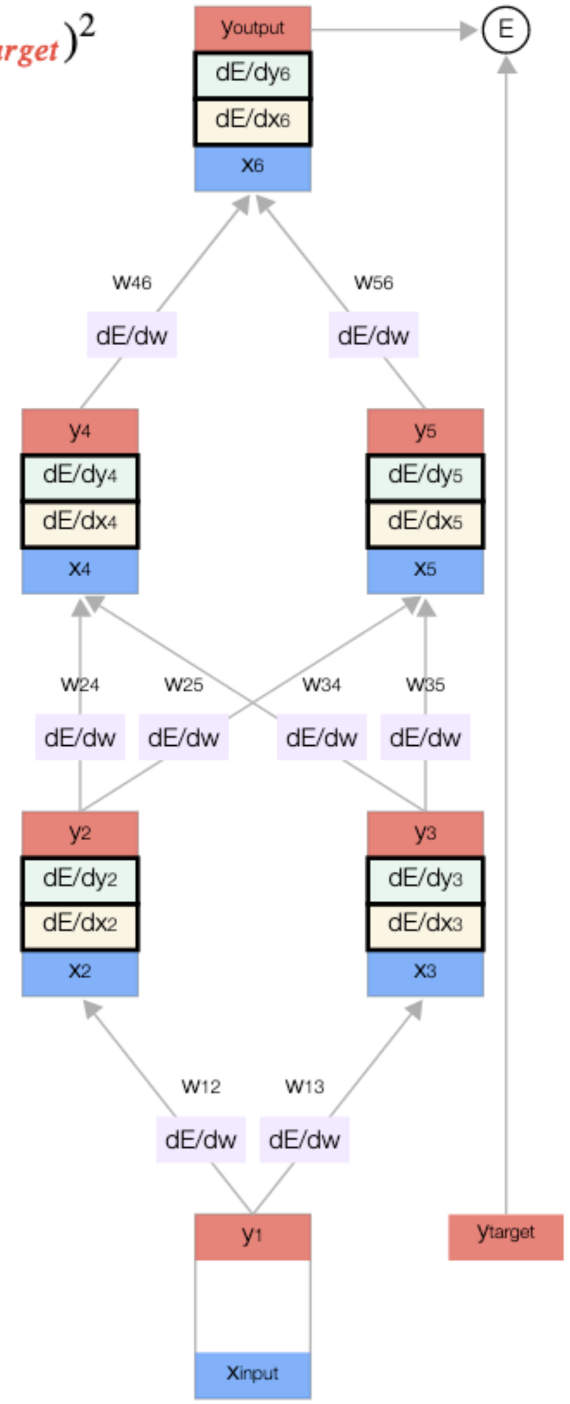
1. Update the last non-terminal node:

Compute and store grad wrt output of node

$$\frac{\partial E}{\partial y_{output}} = y_{output} - y_{target}$$

Compute and store grad wrt input of node

$$\frac{\partial E}{\partial x} = \frac{dy}{dx}\frac{\partial E}{\partial y} = \frac{d}{dx}f(x)\frac{\partial E}{\partial y}$$



$$E = \frac{1}{2}(y_{output} - y_{target})^2$$

Back Propagation Algorithm Step 2

1. Update the last non-terminal node

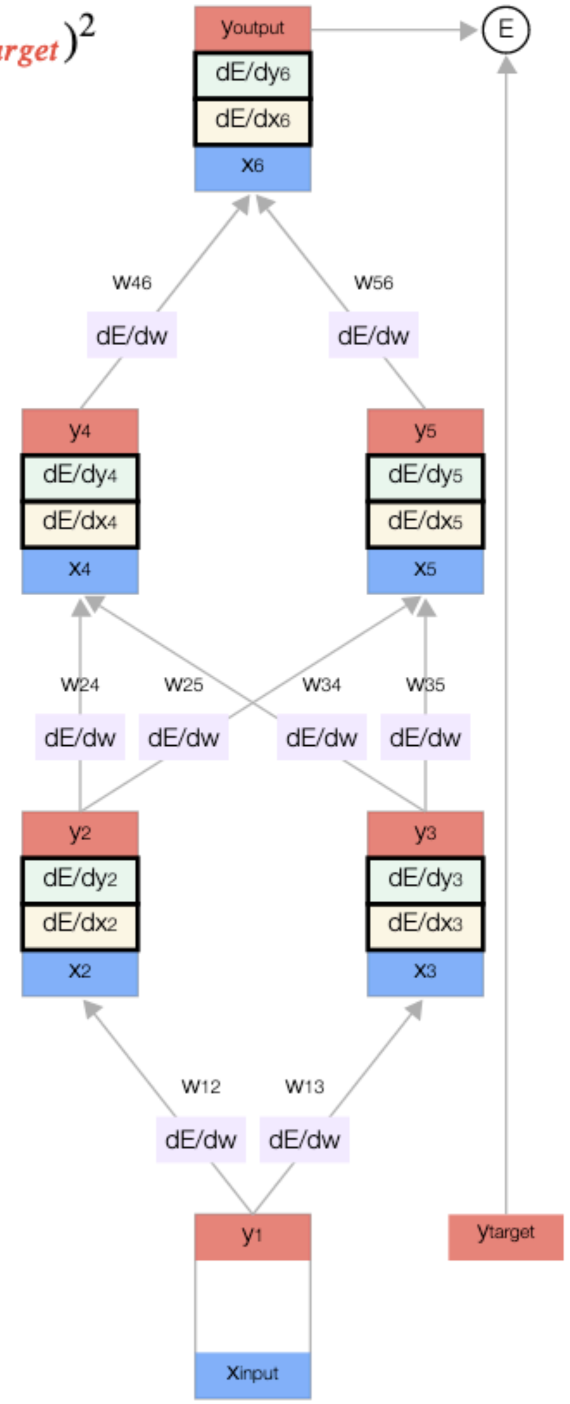
2. For each non-terminal node *i* **in reverse order**

Compute and store grad wrt output of node

$$\frac{\partial E}{\partial y_i} = \sum_{j \in out(i)} \frac{\partial x_j}{\partial y_i} \frac{\partial E}{\partial x_j} = \sum_{j \in out(i)} w_{ij} \frac{\partial E}{\partial x_j}$$

Compute and store grad wrt input of node

$$\frac{\partial E}{\partial x} = \frac{dy}{dx} \frac{\partial E}{\partial y} = \frac{d}{dx} f(x) \frac{\partial E}{\partial y}$$



$$E = \frac{1}{2}(y_{output} - y_{target})^2$$

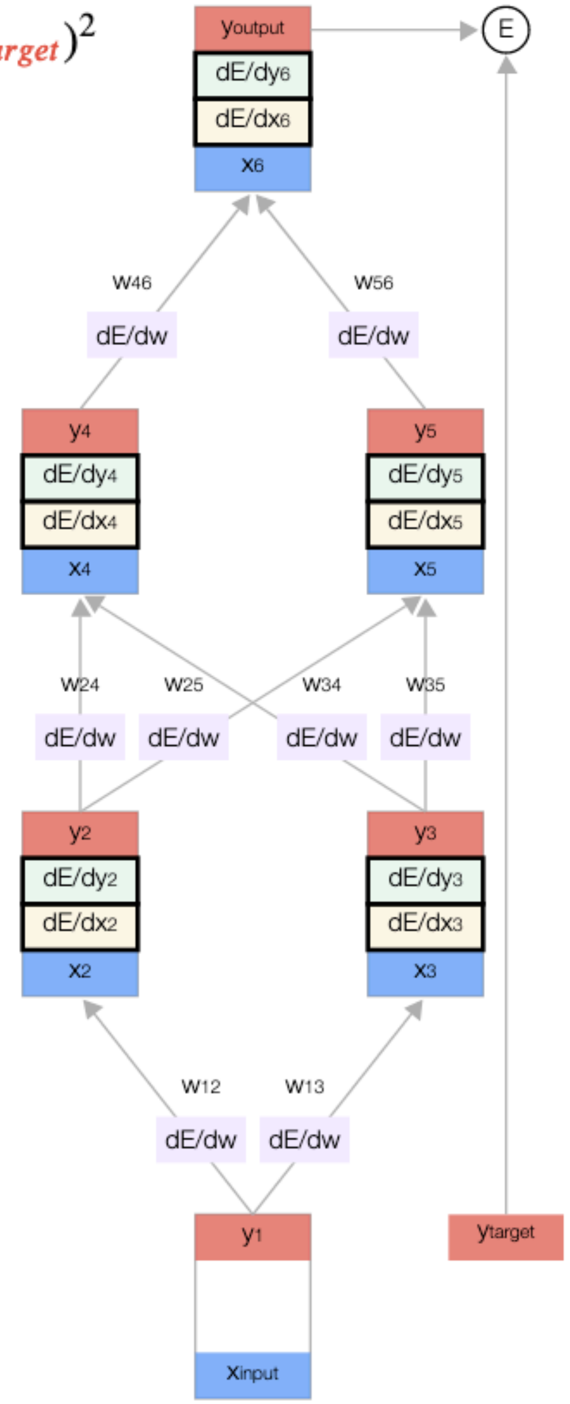
Back Propagation Algorithm Step 3

1. Update the last non-terminal node

2. Update each non-terminal node **in reverse order**

3. Update each edge's gradient wrt weights

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial x_j}{\partial w_{ij}} \frac{\partial E}{\partial x_j} = y_i \frac{\partial E}{\partial x_j}$$



How to train Neural Nets

Training Objective:

$$\min_w \sum_{n=1}^{N} E(y_n, \hat{y}(x_n, w))$$

Gradient Descent Algorithm:

```
w = initialize_weights_at_random_guess(random_state=0)
while not converged:
    total_grad_wrt_w = zeros_like(w)
    for n in 1, 2, … N:
        loss[n], grad_wrt_w[n] = forward_and_backward_prop(x[n], y[n], w)
        total_grad_wrt_w += grad_wrt_w[n]

    w = w - alpha * total_grad_wrt_w
```

$$w_{ij} = w_{ij} - \alpha \frac{dE}{dw_{ij}}$$

Takeaways for Backprop

We can compute gradient wrt weights using a standard dynamic programming algorithm

- Do not need ability to do symbolic derivatives in general
- Only need chain rule, plus a few elementary derivatives

$$\frac{\partial E}{\partial y_{output}} = y_{output} - y_{target}$$

$$\frac{d}{dx} f(x)$$

$$\frac{\partial x_j}{\partial y_i}$$

Takeaways for Backprop

We can compute gradient wrt weights using a standard dynamic programming algorithm

- Do not need ability to do symbolic derivatives in general
- Only need chain rule, plus a few elementary derivatives

Runtime cost of backward propagation algorithm has same ”big O” cost as the the forward pass

Storage cost of backward propagation algorithm is linear in number of nodes and parameters