CSC 411: Lecture 10: Neural Networks I

Raquel Urtasun & Rich Zemel

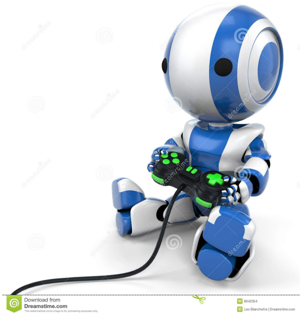
University of Toronto

Oct 14, 2015

Today

- Forward propagation
- Backward propagation
- Deep learning

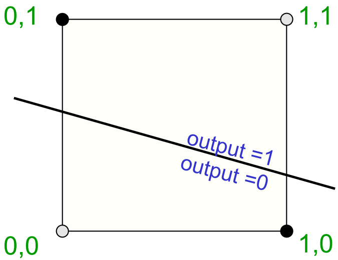
Motivation Examples



Are you excited about deep learning?

Limitations of linear classifiers

- Linear classifiers (e.g., logistic regression) classify inputs based on linear combinations of features x_i
- Many decisions involve non-linear functions of the input
- Canonical example: do 2 input elements have the same value?



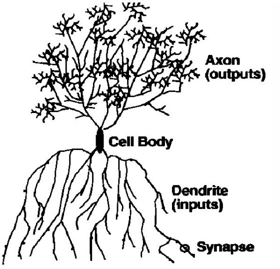
- The positive and negative cases cannot be separated by a plane
- What can we do?

How to construct nonlinear classifiers?

- Would like to construct non-linear discriminative classifiers that utilize functions of input variables
- Add large number of extra functions
  - If these functions are fixed (Gaussian, sigmoid, polynomial basis functions), then optimization still involves linear combinations of (fixed functions of) the inputs
  - Or we can make these functions depend on additional parameters $\rightarrow$ need an efficient method of training extra parameters

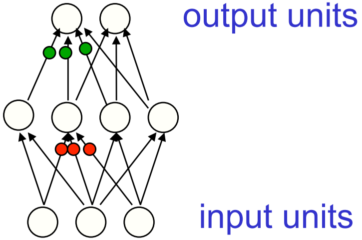
Neural Networks

- Many machine learning methods inspired by biology, brains
- Our brains contain $\sim 10^{11}$ neurons, each of which communicates to $\sim 10^4$ other neurons
- Multi-layer perceptron, or neural network, is popular supervised approach
- Defines extra functions of the inputs (hidden features), computed by neurons
- Artificial neurons called units
- Network output is linear combination of hidden units



Neural network architecture

- Network with one layer of four hidden units:

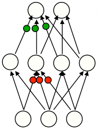


- Each unit computes its value based on linear combination of values of units that point into it
- Can add more layers of hidden units: deeper hidden unit response depends on earlier hiddens

Neural Networks

- We only need to know two algorithms
  - Forward pass: performs inference
  - Backward pass: performs learning

What does the network compute?



- Output of network can be written as (with k indexing the two output units):

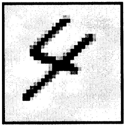
$$
\begin{aligned}
h_j(\mathbf{x}) &= f(w_{j0} + \sum_{i=1}^{D} x_i v_{ji}) \\
o_k(\mathbf{x}) &= g(w_{k0} + \sum_{j=1}^{J} h_j(\mathbf{x}) w_{kj})
\end{aligned}
$$

- Network with non-linear activation function $f()$ is a universal aproximator (esp. with increasing J)
- Standard f: sigmoid/logistic, or tanh, or rectified linear (relu)

$$
tanh(z) = \frac{\exp(z) - exp(-z)}{exp(z) + exp(-z)} \qquad relu(z) = \max(0, z)
$$

Example application

- Consider trying to classify image of handwritten digit: 32x32 pixels



- Single output units – it is a 4 (one vs. all)?
- Use the sigmoid output function:

$$o_k(\mathbf{x}) = \frac{1}{1 + \exp(-z_k)}$$

$$z_k = w_{k0} + \sum_{j=1}^{J} h_j(\mathbf{x}) w_{kj}$$

- What do I recover if $h_j(\mathbf{x}) = x_j$?
- How can we train the network, that is, adjust all the parameters $\mathbf{w}$?
- If we have trained the network, how can we do inference?

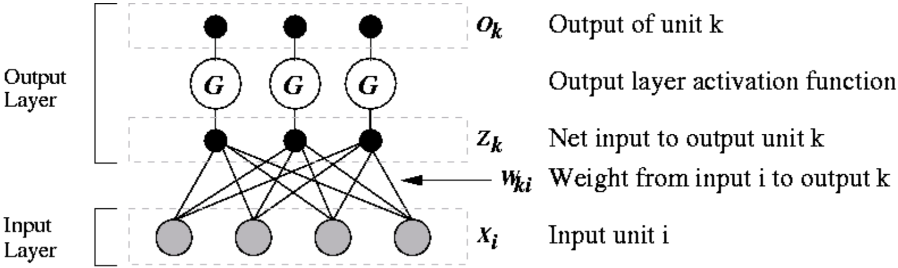
Training multi-layer networks: back-propagation

- Use gradient descent to learn the weights
- Back-propagation: an efficient method for computing gradients needed to perform gradient-based optimization of the weights in a multi-layer network
- Loop until convergence:
  - for each example n
    1. Given input $\mathbf{x}^{(n)}$, propagate activity forward ($\mathbf{x}^{(n)} \rightarrow \mathbf{h}^{(n)} \rightarrow o^{(n)}$)
    2. Propagate gradients backward
    3. Update each weight (via gradient descent)
- Given any error function E, activation functions $g()$ and $f()$, just need to derive gradients

Key idea behind backpropagation

- We don't have targets for a hidden unit, but we can compute how fast the error changes as we change its activity
  - Instead of using desired activities to train the hidden units, use error derivatives w.r.t. hidden activities
  - Each hidden activity can affect many output units and can therefore have many separate effects on the error. These effects must be combined
  - We can compute error derivatives for all the hidden units efficiently
  - Once we have the error derivatives for the hidden activities, its easy to get the error derivatives for the weights going into a hidden unit
- This is just the chain rule!

Computing gradient: single layer network



- Error gradients for single layer network:

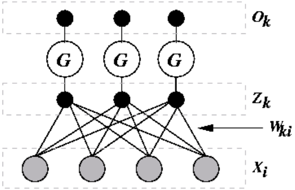
$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial E}{\partial o_k} \frac{\partial o_k}{\partial z_k} \frac{\partial z_k}{\partial w_{ki}}$$

- Error gradient is computable for any continuous activation function $g()$, and any continuous error function

Gradient descent for single layer network

- Assuming the error function is mean-squared error (MSE), on a single training example n, we have

$$\frac{\partial E}{\partial o_k^{(n)}} = o_k^{(n)} - t_k^{(n)}$$



Using logistic activations

$$\begin{aligned} o_k^{(n)} &= g(z_k^{(n)}) = (1 + \exp(z_k^{(n)}))^{-1} \\ \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} &= o_k^{(n)}(1 - o_k^{(n)}) \end{aligned}$$

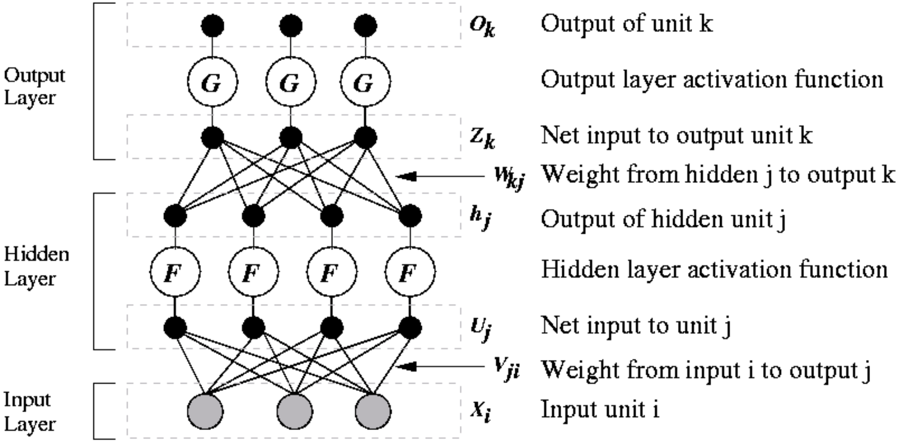
- The error gradient is then:

$$\frac{\partial E}{\partial w_{ki}} = \sum_{n=1}^{N} \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial w_{ki}} = \sum_{n=1}^{N} (o_k^{(n)} - t_k^{(n)}) o_k^{(n)} (1 - o_k^{(n)}) x_i^{(n)}$$

- The gradient descent update rule is given by:

$$w_{ki} \leftarrow w_{ki} - \eta \frac{\partial E}{\partial w_{ki}} = w_{ki} - \eta \sum_{n=1}^{N} (o_k^{(n)} - t_k^{(n)}) o_k^{(n)} (1 - o_k^{(n)}) x_i^{(n)}$$

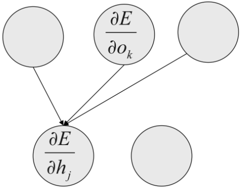
Multi-layer neural network



Back-propagation: sketch on one training case

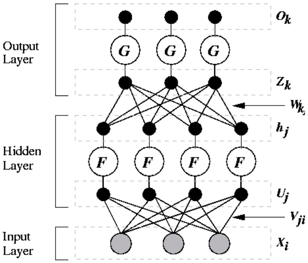
- Convert discrepancy between each output and its target value into an error derivative
$$E = \frac{1}{2}\sum_k (o_k - t_k)^2; \qquad \frac{\partial E}{\partial o_k} = o_k - t_k$$

- Compute error derivatives in each hidden layer from error derivatives in layer above. [assign blame for error at k to each unit j according to its influence on k (depends on w_{kj})]



- Use error derivatives w.r.t. activities to get error derivatives w.r.t. the weights.

Gradient descent for multi-layer network



- The output weight gradients for a multi-layer network are the same as for a single layer network

$$\frac{\partial E}{\partial w_{kj}} = \sum_{n=1}^{N} \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial w_{kj}} = \sum_{n=1}^{N} \delta_k^{(n)} h_j^{(n)}$$

where δ_k is the error w.r.t. the net input for unit k

- Hidden weight gradients are then computed via back-prop:

$$\frac{\partial E}{\partial h_j^{(n)}} = \sum_{k} \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial h_j^{(n)}} = \sum_{k} \delta_k^{(n)} w_{kj}$$

$$\frac{\partial E}{\partial v_{ji}} = \sum_{n=1}^{N} \frac{\partial E}{\partial h_j^{(n)}} \frac{\partial h_j^{(n)}}{\partial u_j^{(n)}} \frac{\partial u_j^{(n)}}{\partial v_{ji}} = \sum_{n=1}^{N} \left(\sum_{k} \delta_k^{(n)} w_{kj} \right) f'(u_j^{(n)}) x_i^{(n)} = \sum_{n=1}^{N} \bar{\delta}_j^{(n)} x_i^{n}$$

Choosing activation and cost functions

- When using a neural network as a function approximator (regressor) sigmoid activation and MSE as loss function work well
- For classification, if it is a binary (2-class) problem, then cross-entropy error function often does better (as we saw with logistic regression)

$$E = -\sum_{n=1}^{N} t^{(n)} \log o^{(n)} + (1 - t^{(n)}) \log(1 - o^{(n)})$$

$$o^{(n)} = (1 + \exp(-z^{(n)})^{-1}$$

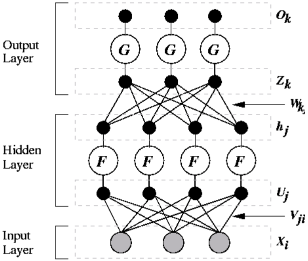
- We can then compute via the chain rule

$$\frac{\partial E}{\partial o} = o - t$$

$$\frac{\partial o}{\partial z} = o(1 - o)$$

$$\frac{\partial E}{\partial z} = \frac{\partial E}{\partial o}\frac{\partial o}{\partial z} = (o - t)o(1 - o)$$

Multi-class classification



- For multi-class classification problems, use the softmax activation

$$E = -\sum_n \sum_k t_k^{(n)} \log o_k^{(n)}$$

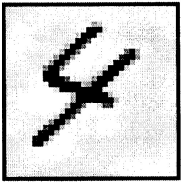
$$o_k^{(n)} = \frac{\exp(z_k^{(n)})}{\sum_j \exp(z_j^{(n)})}$$

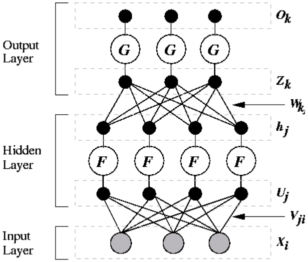
- And the derivatives become

$$\frac{\partial o_k}{\partial z_k} = o_k(1 - o_k)$$

$$\frac{\partial E}{\partial z_k} = \sum_j \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial z_k} = (o_k - t_k) o_k (1 - o_k)$$

Example Application





- Now trying to classify image of handwritten digit: 32x32 pixels
- 10 output units, 1 per digit
- Use the softmax function:

$$o_k = \frac{\exp(z_k)}{\sum_j \exp(z_j)}$$

$$z_k = w_{k0} + \sum_{j=1}^{J} h_j(\mathbf{x}) w_{kj}$$

- What is J ?

Ways to use weight derivatives

- How often to update
  - after a full sweep through the training data

$$w_{ki} \leftarrow w_{ki} - \eta \frac{\partial E}{\partial w_{ki}} = w_{ki} - \eta \sum_{n=1}^{N} (o_k^{(n)} - t_k^{(n)}) o_k^{(n)} (1 - o_k^{(n)}) x_i^{(n)}$$

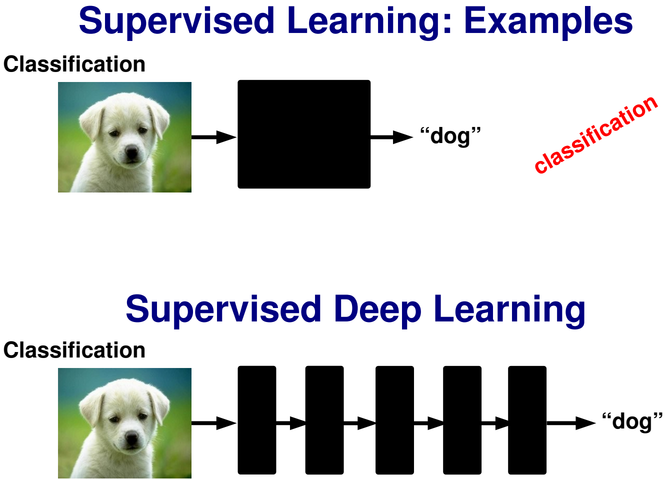
  - after each training case
  - after a mini-batch of training cases
- How much to update
  - Use a fixed learning rate
  - Adapt the learning rate
  - Add momentum

$$\begin{aligned} w_{ki} &\leftarrow w_{ki} - v \\ v &\leftarrow \gamma v + \eta \frac{\partial E}{\partial w_{ki}} \end{aligned}$$

Deep Neural Networks

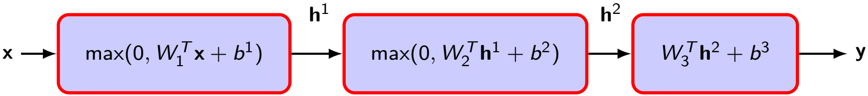
- We only need to know two algorithms
  - Forward pass: performs inference
  - Backward pass: performs learning

- Neural nets are now called deep learning. Why?



[Picture from M. Ranzato]

- Deep learning uses composite of simple functions (e.g., ReLU, sigmoid, tanh, max) to create complex non-linear functions
- Note: a composite of linear functions is linear!
- Example: 2 layer NNet (now matrix and vector form!)



  - $\mathbf{x}$ is the input
  - $\mathbf{y}$ is the output (what we want to predict)
  - $\mathbf{h}^i$ is the i-th hidden layer
  - W^i are the parameters of the i-th layer

Evaluating the Function

- Assume we have learn the weights and we want to do inference
- Forward Propagation: compute the output given the input

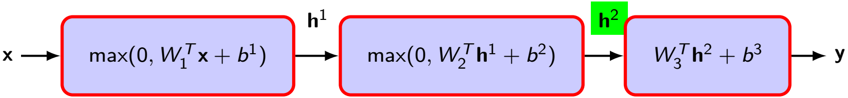
$$\mathbf{x} \longrightarrow \boxed{\max(0, W_1^T \mathbf{x} + b^1)} \xrightarrow{\mathbf{h}^1} \boxed{\max(0, W_2^T \mathbf{h}^1 + b^2)} \xrightarrow{\mathbf{h}^2} \boxed{W_3^T \mathbf{h}^2 + b^3} \longrightarrow \mathbf{y}$$

- Fully connected layer: Each hidden unit takes as input all the units from the previous layer
- The non-linearity is called a ReLU (rectified linear unit), with $\mathbf{x} \in \Re^D$, $b^i \in \Re^{N_i}$ the biases and $W^i \in \Re^{N_i \times N_{i-1}}$ the weights
- Do it in a compositional way,

$$\mathbf{h}^1 = \max(0, W^1 \mathbf{x} + b^1)$$

Evaluating the Function

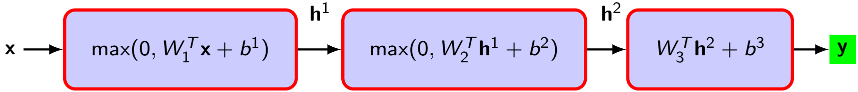
- Assume we have learn the weights and we want to do inference
- Forward Propagation: compute the output given the input



- Fully connected layer: Each hidden unit takes as input all the units from the previous layer
- The non-linearity is called a ReLU (rectified linear unit), with $\mathbf{x} \in \Re^D$, $b^i \in \Re^{N_i}$ the biases and $W^i \in \Re^{N_i \times N_{i-1}}$ the weights
- Do it in a compositional way

$$\begin{aligned}\mathbf{h}^1 &= \max(0, W^1 \mathbf{x} + b^1) \\ \mathbf{h}^2 &= \max(0, W^2 \mathbf{h}^1 + b^2)\end{aligned}$$

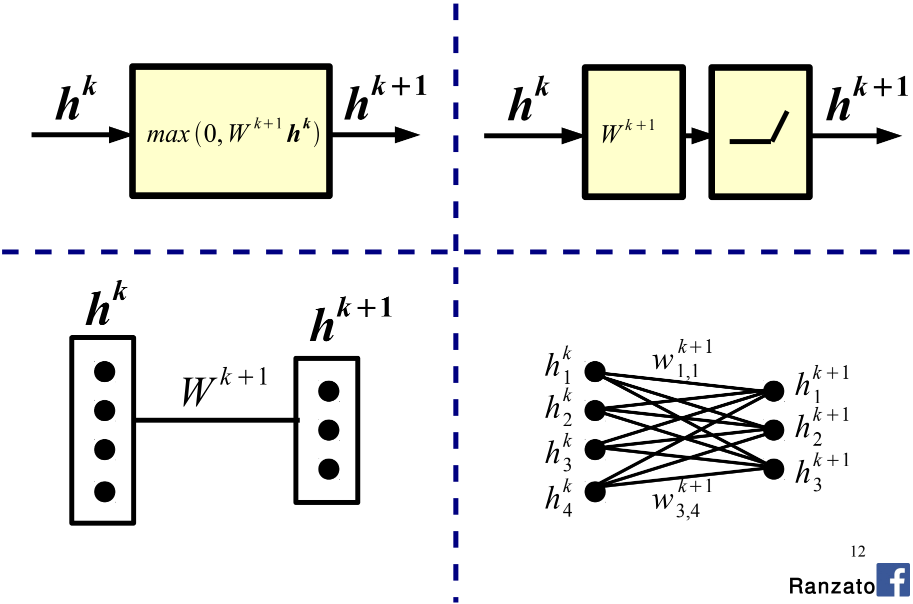
- Assume we have learn the weights and we want to do inference
- Forward Propagation: compute the output given the input



- Fully connected layer: Each hidden unit takes as input all the units from the previous layer
- The non-linearity is called a ReLU (rectified linear unit), with $\mathbf{x} \in \Re^D$, $b^i \in \Re^{N_i}$ the biases and $W^i \in \Re^{N_i \times N_{i-1}}$ the weights
- Do it in a compositional way

$$\begin{aligned}
\mathbf{h}^1 &= \max(0, W^1 \mathbf{x} + b^1) \\
\mathbf{h}^2 &= \max(0, W^2 \mathbf{h}^1 + b^2) \\
\mathbf{y} &= \max(0, W^3 \mathbf{h}^2 + b^3)
\end{aligned}$$

Alternative Graphical Representation



$$\mathbf{x} \rightarrow \boxed{\max(0, W_1^T\mathbf{x} + b^1)} \xrightarrow{\mathbf{h}^1} \boxed{\max(0, W_2^T\mathbf{h}^1 + b^2)} \xrightarrow{\mathbf{h}^2} \boxed{W_3^T\mathbf{h}^2 + b^3} \rightarrow \mathbf{y}$$

- We want to estimate the parameters, biases and hyper-parameters (e.g., number of layers, number of units) such that we do good predictions
- Collect a training set of input-output pairs $\{\mathbf{x}, t\}$
- Encode the output with 1-K encoding $t = [0, \cdots, 1, \cdots, 0]$
- Define a loss per training example and minimize the empirical risk

$$\mathcal{L}(\mathbf{w}) = \frac{1}{N}\sum_i \ell(\mathbf{w}, \mathbf{x}^{(i)}, t^{(i)})$$

with N number of examplesand $\mathbf{w}$ contains all parameters

Loss Functions

$$\mathcal{L}(\mathbf{w}) = \frac{1}{N} \sum_i \ell(\mathbf{w}, \mathbf{x}^{(i)}, t^{(i)})$$

- Probability of class k given input (softmax):

$$p(c_k = 1 | \mathbf{x}) = \frac{\exp(y_k)}{\sum_{j=1}^{C} \exp(y_j)}$$

- Cross entropy is the most used loss function for classification

$$\ell(\mathbf{x}, t, \mathbf{w}) = -\sum_i t^{(i)} \log p(c_i | \mathbf{x})$$

- Use gradient descent to train the network

$$\min_{\mathbf{w}} \frac{1}{N} \sum_i \ell(\mathbf{w}, \mathbf{x}^{(i)}, t^{(i)})$$

Backpropagation

- Efficient computation of the gradients by applying the chain rule

$$\mathbf{x} \rightarrow \boxed{\max(0, W_1^T\mathbf{x} + b^1)} \xrightarrow{\mathbf{h}^1} \boxed{\max(0, W_2^T\mathbf{h}^1 + b^2)} \xrightarrow{\mathbf{h}^2} \boxed{W_3^T\mathbf{h}^2 + b^3} \xleftarrow{\frac{\partial \ell}{\partial y}} \mathbf{y}$$

$$\begin{aligned} p(c_k = 1|\mathbf{x}) &= \frac{\exp(y_k)}{\sum_{j=1}^{C}\exp(y_j)} \\ \ell(\mathbf{x}, t, \mathbf{w}) &= -\sum_i t^{(i)} \log p(c_i|\mathbf{x}) \end{aligned}$$

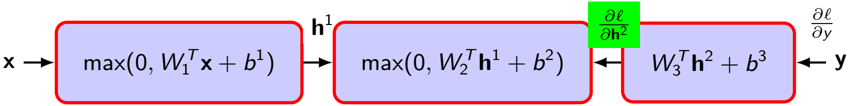
- Compute the derivative of loss w.r.t. the output

$$\frac{\partial \ell}{\partial y} = p(c|\mathbf{x}) - t$$

- Note that the forward pass is necessary to compute $\frac{\partial \ell}{\partial y}$

Backpropagation

- Efficient computation of the gradients by applying the chain rule



- We have computed the derivative of loss w.r.t the output

$$\frac{\partial \ell}{\partial y} = p(c|\mathbf{x}) - t$$

- Given $\frac{\partial \ell}{\partial y}$ if we can compute the Jacobian of each module

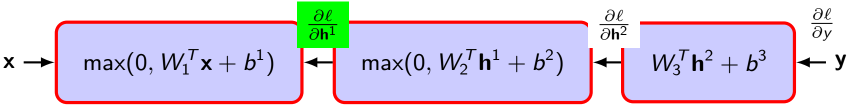
$$\frac{\partial \ell}{\partial W^3} = \frac{\partial \ell}{\partial y} \frac{\partial y}{\partial W^3} = (p(c|\mathbf{x}) - t)(\mathbf{h}^2)^T$$

$$\frac{\partial \ell}{\partial \mathbf{h}^2} = \frac{\partial \ell}{\partial y} \frac{\partial y}{\partial \mathbf{h}^2} = (W^3)^T (p(c|\mathbf{x}) - t)$$

- Need to compute gradient w.r.t. inputs and parameters in each layer

Backpropagation

- Efficient computation of the gradients by applying the chain rule



$$\frac{\partial \ell}{\partial \mathbf{h}^2} = \frac{\partial \ell}{\partial y} \frac{\partial y}{\partial \mathbf{h}^2} = (W^3)^T (p(c|\mathbf{x}) - t)$$

- Given $\frac{\partial \ell}{\partial \mathbf{h}^2}$ if we can compute the Jacobian of each module

$$\frac{\partial \ell}{\partial W^2} = \frac{\partial \ell}{\partial \mathbf{h}^2} \frac{\partial \mathbf{h}^2}{\partial W^2}$$

$$\frac{\partial \ell}{\partial \mathbf{h}^1} = \frac{\partial \ell}{\partial \mathbf{h}^2} \frac{\partial \mathbf{h}^2}{\partial \mathbf{h}^1}$$

Toy Code (Matlab): Neural Net Trainer

```
% F-PROP
for i = 1 : nr_layers - 1
  [h{i}  jac{i}]  =  nonlinearity(W{i} * h{i-1} +  b{i});
end
h{nr_layers-1}  =  W{nr_layers-1} * h{nr_layers-2}  +   b{nr_layers-1};
prediction  =  softmax(h{l-1});


% CROSS ENTROPY LOSS
loss  =  -  sum(sum(log(prediction)  .*  target)) / batch_size;


% B-PROP
dh{l-1}  =  prediction  -  target;
for i = nr_layers - 1 : -1 : 1
  Wgrad{i}  =  dh{i} * h{i-1}';
  bgrad{i}  =  sum(dh{i}, 2);
  dh{i-1}  =  (W{i}' * dh{i})  .*  jac{i-1};
end


% UPDATE
for i = 1 : nr_layers - 1
  W{i}  =  W{i}  -  (lr / batch_size)  *  Wgrad{i};
  b{i}  =  b{i}  -  (lr / batch_size)  *  bgrad{i};
end
```

28

Ranzato