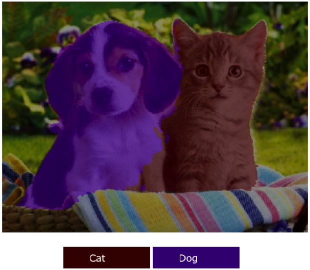
CSC 411: Lecture 10: Neural Networks I

Richard Zemel, Raquel Urtasun and Sanja Fidler

University of Toronto

Motivating Examples



Inspiration: The Brain

- Many machine learning methods inspired by biology, e.g., the (human) brain
- Our brain has $\sim 10^{11}$ neurons, each of which communicates (is connected) to $\sim 10^4$ other neurons

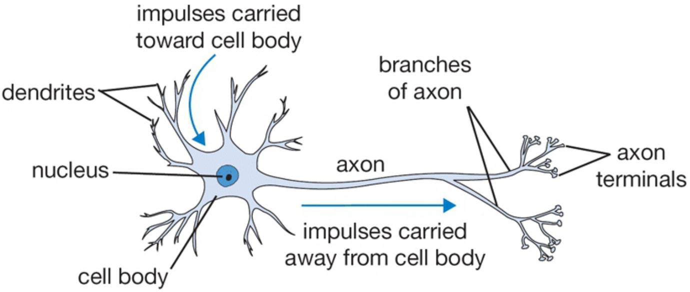


Figure : The basic computational unit of the brain: Neuron

[Pic credit: http://cs231n.github.io/neural-networks-1/]

Mathematical Model of a Neuron

- Neural networks define functions of the inputs (hidden features), computed by neurons
- Artificial neurons are called units

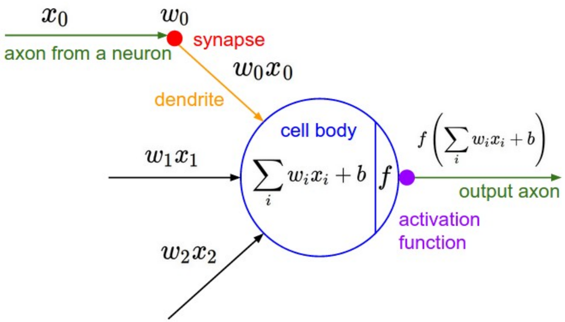


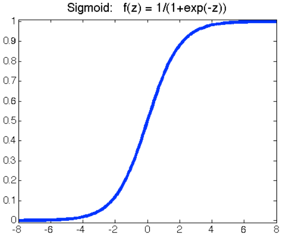
Figure : A mathematical model of the neuron in a neural network

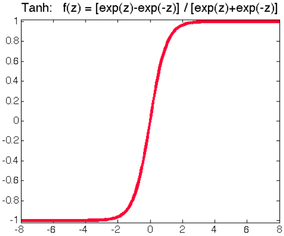
[Pic credit: http://cs231n.github.io/neural-networks-1/]

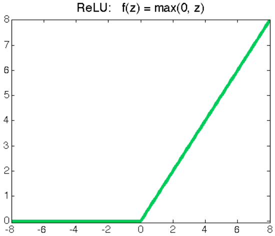
Activation Functions

Most commonly used activation functions:

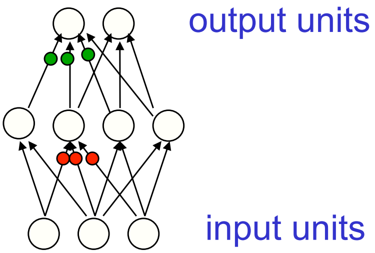
- Sigmoid: $\sigma(z) = \frac{1}{1+\exp(-z)}$
- Tanh: $\tanh(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}$
- ReLU (Rectified Linear Unit): $\mathrm{ReLU}(z) = \max(0, z)$







- Network with one layer of four hidden units:



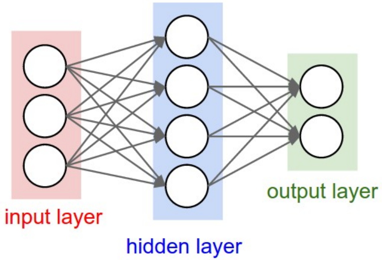


Figure : Two different visualizations of a 2-layer neural network. In this example: 3 input units, 4 hidden units and 2 output units

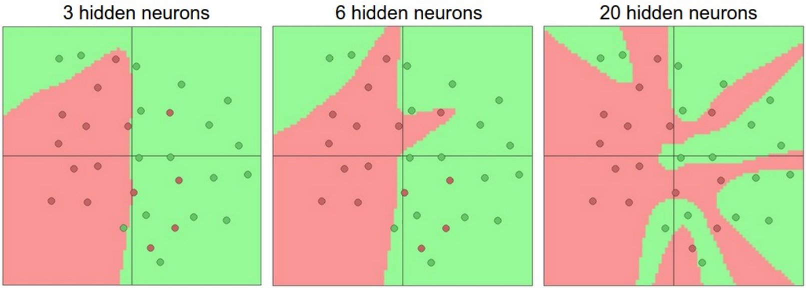
- Each unit computes its value based on linear combination of values of units that point into it, and an activation function

[http://cs231n.github.io/neural-networks-1/]

Representational Power

- Neural network with at **least one hidden layer** is a universal approximator (can represent any function).

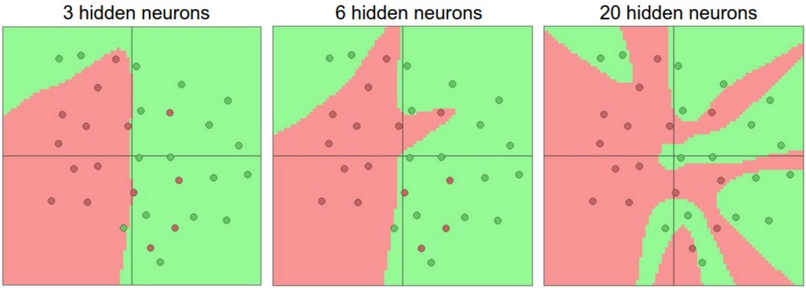
  Proof in: Approximation by Superpositions of Sigmoidal Function, Cybenko, paper



- The capacity of the network increases with more hidden units and more hidden layers

Representational Power

- Neural network with at **least one hidden layer** is a universal approximator (can represent any function).
  Proof in: Approximation by Superpositions of Sigmoidal Function, Cybenko, paper



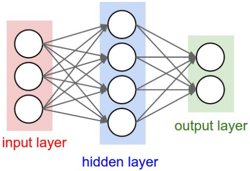
- The capacity of the network increases with more hidden units and more hidden layers
- Why go deeper? Read e.g.,: Do Deep Nets Really Need to be Deep? Jimmy Ba, Rich Caruana, Paper: paper]

[http://cs231n.github.io/neural-networks-1/]

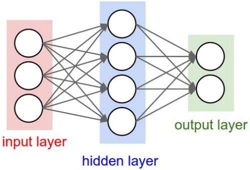
Neural Networks

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

Forward Pass: What does the Network Compute?



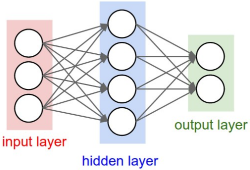
Forward Pass: What does the Network Compute?



- Output of the network can be written as:

$$h_j(\mathbf{x}) \quad = \quad f(v_{j0} + \sum_{i=1}^{D} x_i v_{ji})$$

Forward Pass: What does the Network Compute?



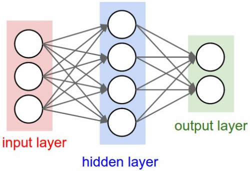
- Output of the network can be written as:

$$h_j(\mathbf{x}) = f(v_{j0} + \sum_{i=1}^{D} x_i v_{ji})$$

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

(j indexing hidden units, k indexing the output units, D number of inputs)

Forward Pass: What does the Network Compute?



- Output of the network can be written as:

$$h_j(\mathbf{x}) = f(v_{j0} + \sum_{i=1}^{D} x_i v_{ji})$$

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

(j indexing hidden units, k indexing the output units, D number of inputs)

- Activation functions f, g: sigmoid/logistic, tanh, or rectified linear (ReLU)

$$\sigma(z) = \frac{1}{1+\exp(-z)}, \quad \tanh(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}, \quad \mathrm{ReLU}(z) = \max(0, z)$$

Training Neural Networks

- Find weights:

$$\mathbf{w}^* = \operatorname*{argmin}_{\mathbf{w}} \sum_{n=1}^{N} \mathrm{loss}(\mathbf{o}^{(n)}, \mathbf{t}^{(n)})$$

where $\mathbf{o} = f(\mathbf{x}; \mathbf{w})$ is the output of a neural network

Training Neural Networks

- Find weights:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{n=1}^{N} \text{loss}(\mathbf{o}^{(n)}, \mathbf{t}^{(n)})$$

  where $\mathbf{o} = f(\mathbf{x}; \mathbf{w})$ is the output of a neural network

- Define a loss function, eg:
  - Squared loss: $\sum_k \frac{1}{2}(o_k^{(n)} - t_k^{(n)})^2$
  - Cross-entropy loss: $-\sum_k t_k^{(n)} \log o_k^{(n)}$

Training Neural Networks

- Find weights:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\text{argmin}} \sum_{n=1}^{N} \text{loss}(\mathbf{o}^{(n)}, \mathbf{t}^{(n)})$$

  where $\mathbf{o} = f(\mathbf{x}; \mathbf{w})$ is the output of a neural network

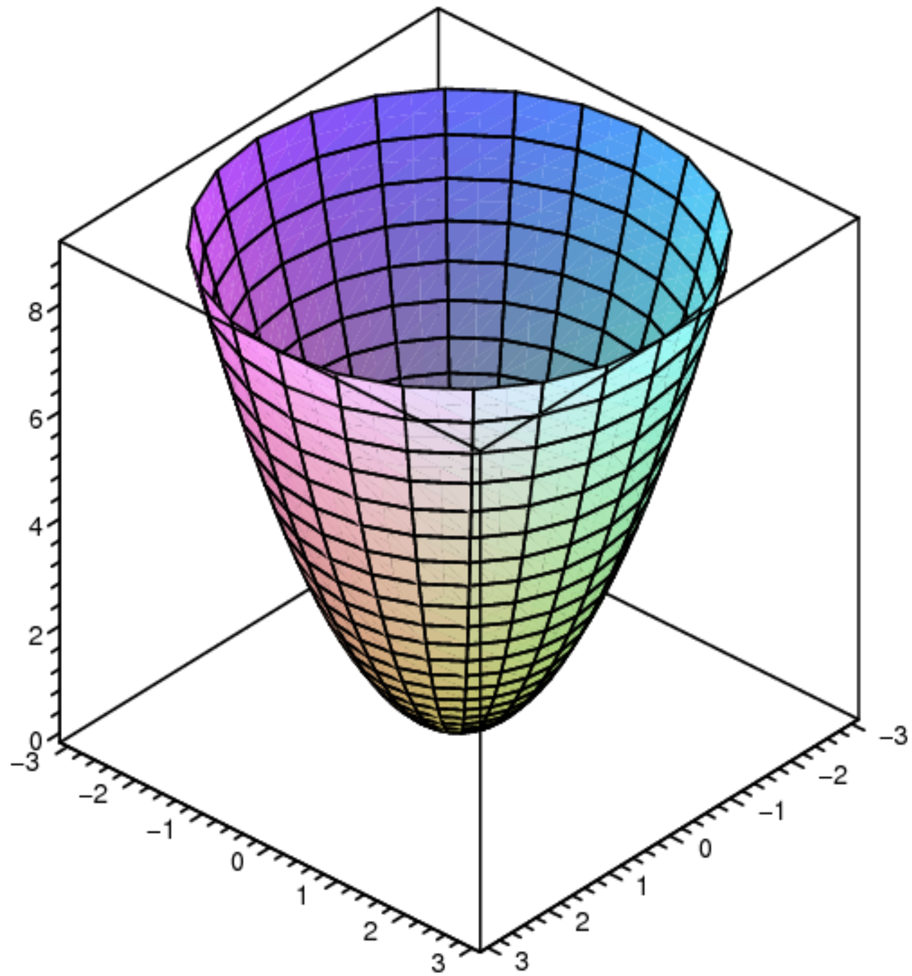
- Define a loss function, eg:
  - Squared loss: $\sum_k \frac{1}{2}(o_k^{(n)} - t_k^{(n)})^2$
  - Cross-entropy loss: $-\sum_k t_k^{(n)} \log o_k^{(n)}$

- Gradient descent:

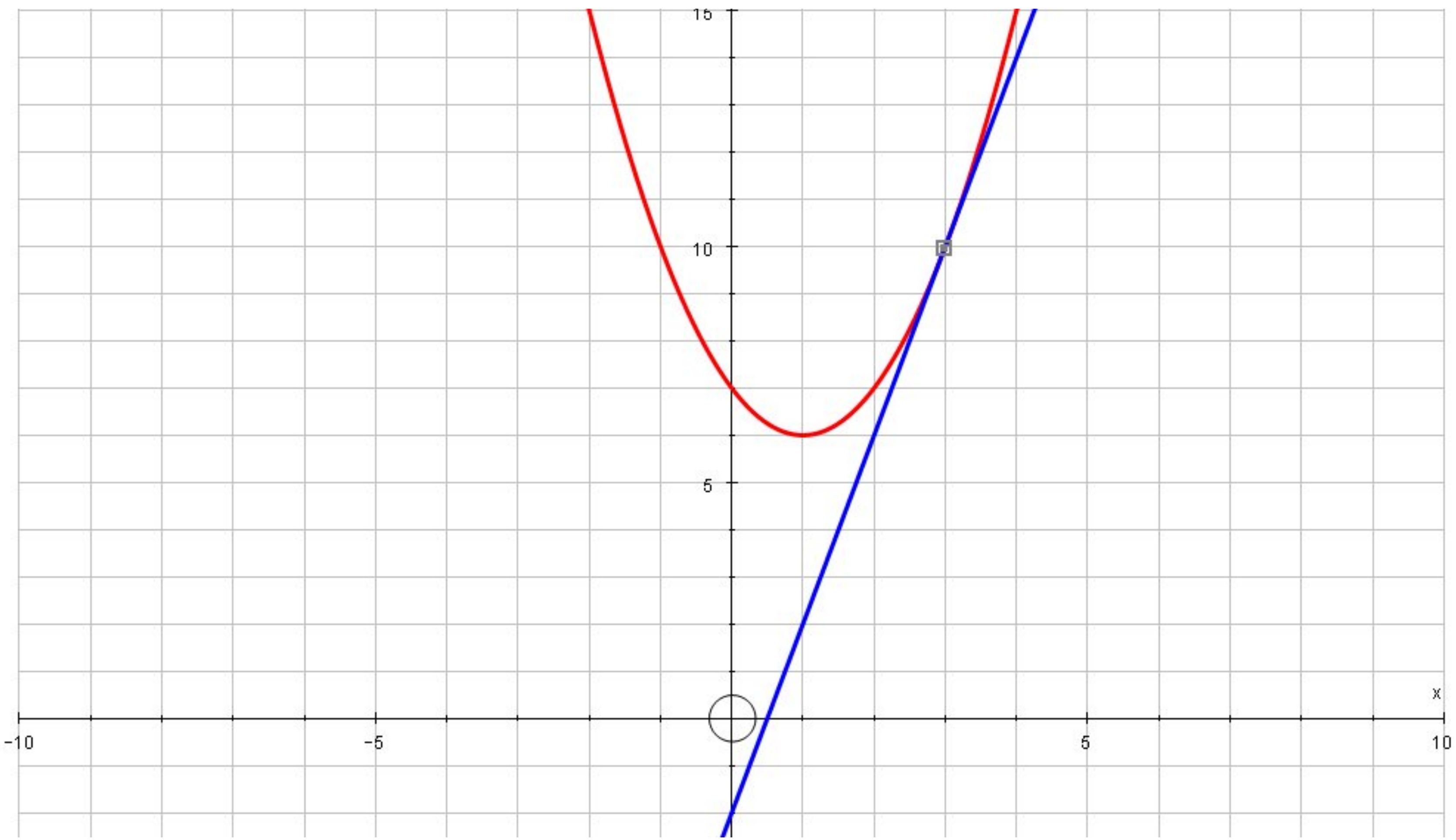
$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \frac{\partial E}{\partial \mathbf{w}^t}$$

  where η is the learning rate (and E is error/loss)

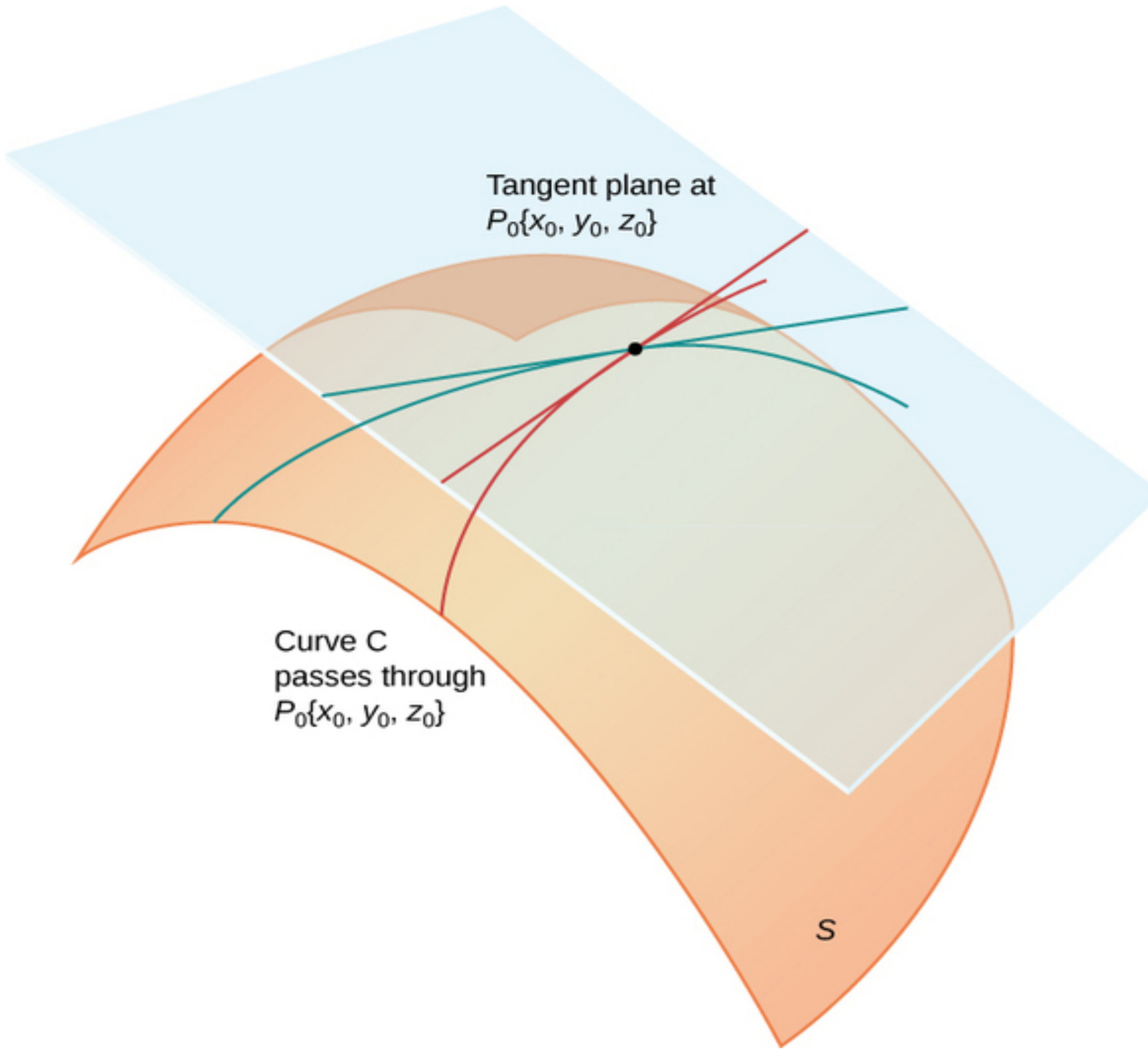
$$\ell(\mathbf{w}) = {w_0}^2 + {w_1}^2$$



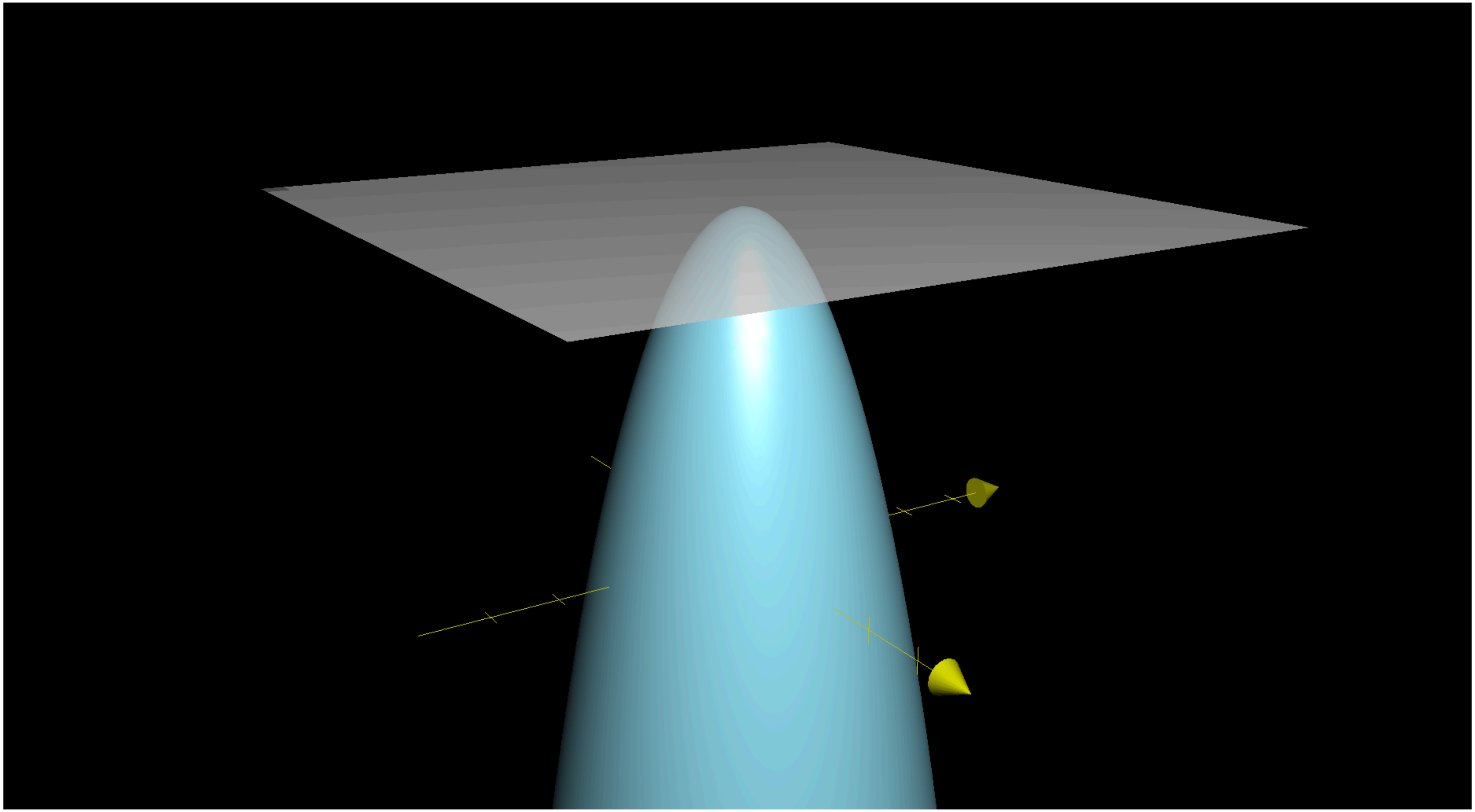
Tangent Line



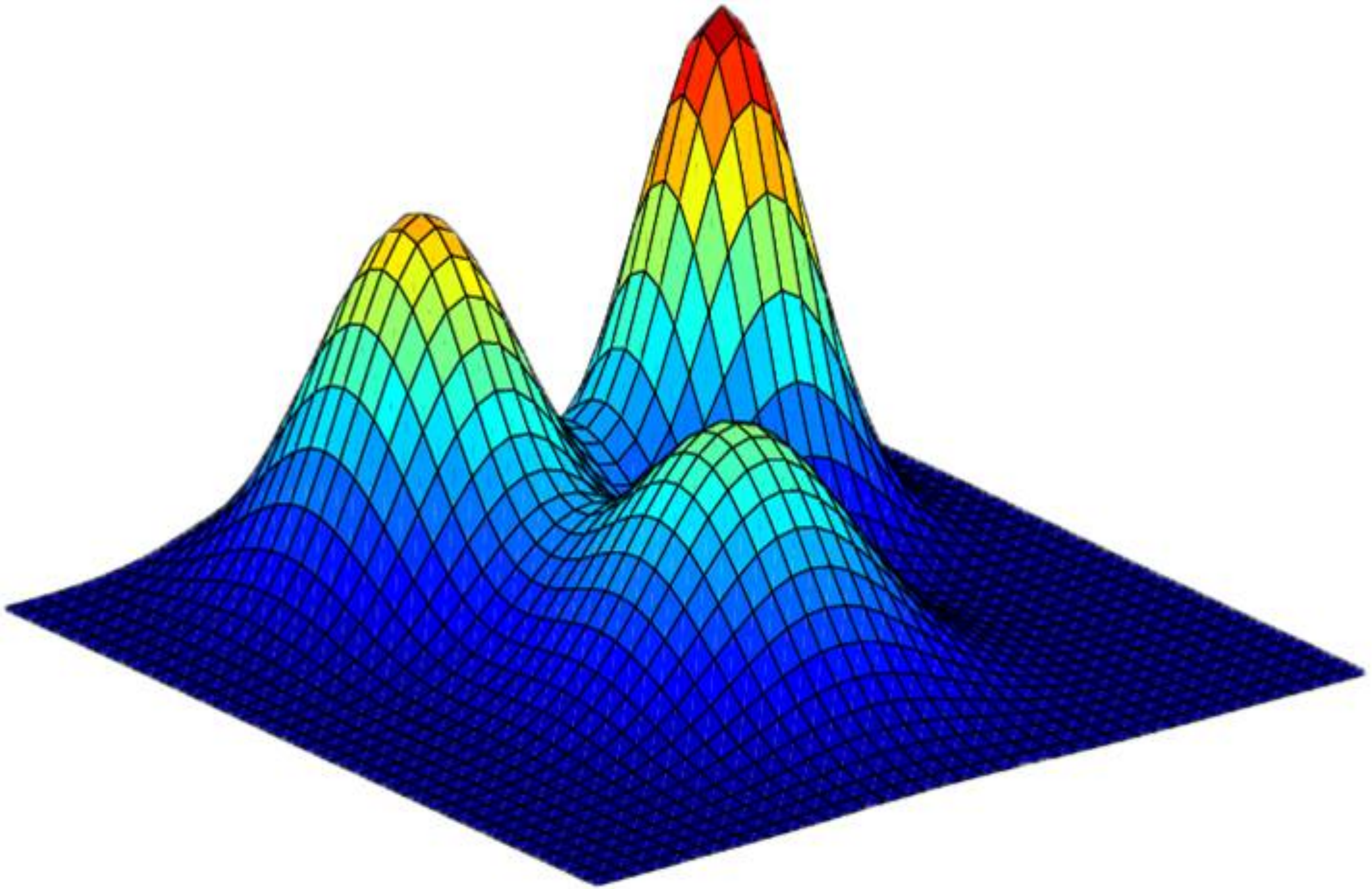
Tangent Plane



Tangent Plane at a Maximum



Mixture of three Gaussians



Training Neural Networks: Back-propagation

- Back-propagation: an efficient method for computing gradients needed to perform gradient-based optimization of the weights in a multi-layer network

Training neural nets:

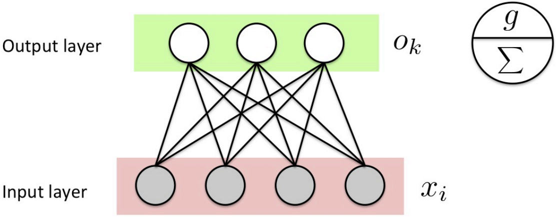
Loop until convergence:

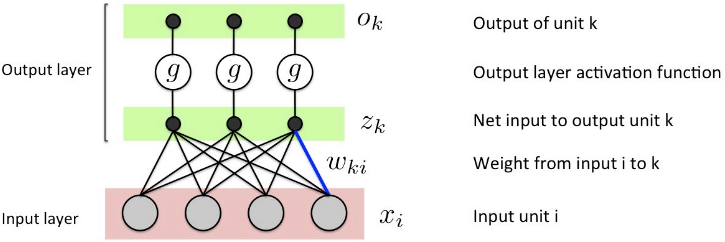
- for each example n
  1. Given input $\mathbf{x}^{(n)}$, propagate activity forward ($\mathbf{x}^{(n)} \rightarrow \mathbf{h}^{(n)} \rightarrow o^{(n)}$) (**forward pass**)
  2. Propagate gradients backward (**backward pass**)
  3. Update each weight (via gradient descent)

- Given any error function E, activation functions $g()$ and $f()$, just need to derive gradients

Computing Gradients: Single Layer Network

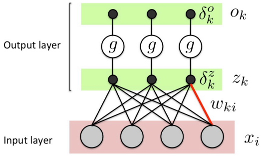
- Let's take a single layer network and draw it a bit differently





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

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



Using logistic activation functions:

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

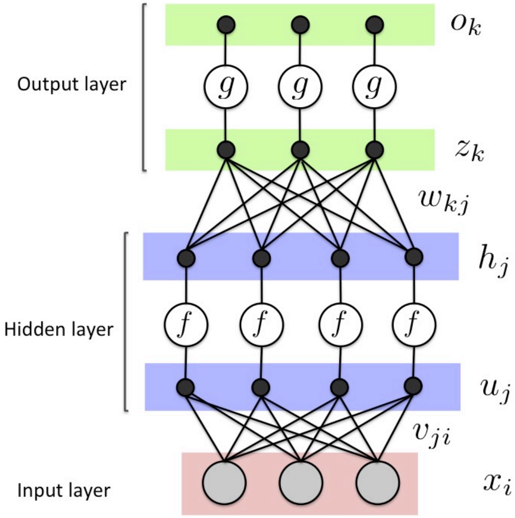
- The error gradient is then:

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

- The gradient descent update rule is given by:

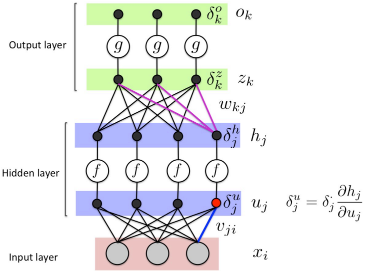
$$w_{ki} \leftarrow w_{ki} - \eta \frac{\partial E}{\partial w_{ki}} = w_{ki} - \eta \sum_{n=1}^{N} (o_k^{(n)} - t_k^{(n)}) o_k^{(n)} (1 - o_k^{(n)}) x_i^{(n)}$$

Multi-layer Neural Network



o_k	Output of unit k
	Output layer activation function
z_k	Net input to output unit k
w_{kj}	Weight from hidden unit j to output k
h_j	Output of hidden unit j
	Hidden layer activation function
u_j	Net input to unit j
v_{ji}	Weight from input i to j
x_i	Input unit i

Gradient Descent for Multi-layer Network



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

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

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

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

$$\frac{\partial E}{\partial h_j^{(n)}} = \sum_k \frac{\partial E}{\partial o_k^{(n)}} \frac{\partial o_k^{(n)}}{\partial z_k^{(n)}} \frac{\partial z_k^{(n)}}{\partial h_j^{(n)}} = \sum_k \delta_k^{z,(n)} w_{kj} := \delta_j^{h,(n)}$$

$$\frac{\partial E}{\partial v_{ji}} = \sum_{n=1}^{N} \frac{\partial E}{\partial h_j^{(n)}} \frac{\partial h_j^{(n)}}{\partial u_j^{(n)}} \frac{\partial u_j^{(n)}}{\partial v_{ji}} = \sum_{n=1}^{N} \delta_j^{h,(n)} f'(u_j^{(n)}) \frac{\partial u_j^{(n)}}{\partial v_{ji}} = \sum_{n=1}^{N} \delta_j^{u,(n)} x_i^{(n)}$$

Neural Networks

- 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 hidden layer NNet (now matrix and vector form!) with ReLU as nonlinearity

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

  - $\mathbf{x}$ 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

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

- Do it in a compositional way,

$$\mathbf{h}^1 = \max(0, W_1^T\mathbf{x} + b^1)$$

Evaluating the Function

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

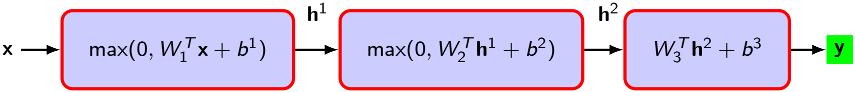
$$\mathbf{x} \longrightarrow \boxed{\max(0, W_1^T\mathbf{x} + b^1)} \xrightarrow{\mathbf{h}^1} \boxed{\max(0, W_2^T\mathbf{h}^1 + b^2)} \xrightarrow{\mathbf{h}^2} \boxed{W_3^T\mathbf{h}^2 + b^3} \longrightarrow \mathbf{y}$$

- Do it in a compositional way

$$\begin{aligned}\mathbf{h}^1 &= \max(0, W_1^T\mathbf{x} + b_1)\\ \mathbf{h}^2 &= \max(0, W_2^T\mathbf{h}^1 + b_2)\end{aligned}$$

Evaluating the Function

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



- Do it in a compositional way

$$
\begin{aligned}
\mathbf{h}^1 &= \max(0, W_1^T\mathbf{x} + b_1) \\
\mathbf{h}^2 &= \max(0, W_2^T\mathbf{h}^1 + b_2) \\
\mathbf{y} &= W_3^T\mathbf{h}^2 + b_3
\end{aligned}
$$

Learning

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

- 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 $\{\mathbf{x}^{(n)}, \mathbf{t}^{(n)}\}$
- For classification: Encode the output with 1-K encoding $\mathbf{t} = [0, .., 1, .., 0]$
- Define a loss per training example and minimize the empirical risk

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

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

Loss Function: Classification

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

- 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

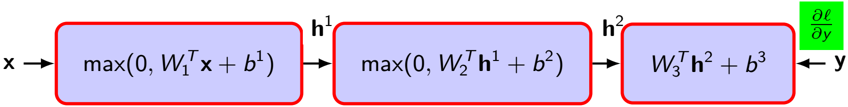
$$\ell(\mathbf{w}, \mathbf{x}^{(n)}, \mathbf{t}^{(n)}) = -\sum_k t_k^{(n)} \log p(c_k|\mathbf{x})$$

- Use gradient descent to train the network

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

Backpropagation

- Efficient computation of the gradients by applying the chain rule



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

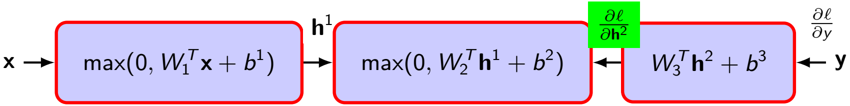
- Compute the derivative of loss w.r.t. the output

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

- 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

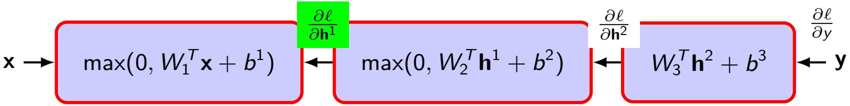
$$\frac{\partial \ell}{\partial W_3} = \frac{\partial \ell}{\partial y}\frac{\partial y}{\partial W_3} = (p(c|\mathbf{x}) - t)(\mathbf{h}^2)^T$$

$$\frac{\partial \ell}{\partial \mathbf{h}^2} = \frac{\partial \ell}{\partial y}\frac{\partial y}{\partial \mathbf{h}^2} = (W_3)^T(p(c|\mathbf{x}) - t)$$

- 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

$$\frac{\partial \ell}{\partial W_2} = \frac{\partial \ell}{\partial \mathbf{h}^2}\frac{\partial \mathbf{h}^2}{\partial W_2}$$

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